Sun™ ONE Studio 5
J2EE™ Application Tutorial

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 817-2322-10
June 2003 Revision A

Send comments about this document to: docfeedback@sun.com

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at http://www.sun.com/patents and one or more additional patents or pending patent applications in the U.S. and in other countries.

This document and the product to which it pertains are distributed under licenses restricting their use, copying, distribution, and decompilation. No part of the product or of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun logo, Forte, Java, NetBeans, iPlanet, docs.sun.com, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

Federal Acquisitions: Commercial Software - Government Users Subject to Standard License Terms and Conditions.

DOCUMENTATION IS PROVIDED “AS IS” AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. a les droits de propriété intellectuels relatants à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et sans la limitation, ces droits de propriété intellectuels peuvent inclure un ou plus des brevets américains énumérés à http://www.sun.com/patents et un ou les brevets plus supplémentaires ou les applications de brevet en attente dans les Etats - Unis et dans les autres pays.

Ce produit est un document protege par un copyright et distribue avec des licenses qui est en restreignent l’utilisation, la copie, la distribution et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, parquelque moyen que ce soit, sans l’autorisation préalable et écrite de Sun et de ses bailleurs de licence, s’il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, Forte, Java, NetBeans, iPlanet, docs.sun.com, et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d’autres pays.

Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d’autres pays. Les produits protant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

UNIX est une marque enregistree aux Etats-Unis et dans d’autres pays et licenciée exclusivement par X/Open Company Ltd.

LA DOCUMENTATION EST FOURNIE “EN L’ÉTAT” ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L’APTITUDE A UNE UTILISATION PARTICULIERE OU A L’ABSENCE DE CONTREFAÇON.

Please
Recycle

Contents

Before You Begin 13

1. Getting Started 21

Obtaining and Installing the Required Software 21

Starting the Software 23

Starting the IDE 23

Starting the Application Server 23

Confirming Sun ONE Application Server 7 as the Default Server 29

Setting Up Database Connectivity 29

Enabling the JDBC Driver 30

Setting JDBC Resources (Microsoft Windows Superusers) 30

Setting JDBC Resources (All Other Users) 31

Tutorial Database Table Descriptions 32

2. Introduction to the Tutorial 35

Functionality of the Tutorial Application 35

Application Scenarios 36

Application Functional Specification 36

User’s View of the Tutorial Application 37

Architecture of the Tutorial Application 40

Application Elements 41

EJB Tier Details 41

Overview of Tasks for Creating the Tutorial Application 42

Creating the EJB Components 42

Creating the Tutorial's Web Service 44

Installing and Using the Provided Client 45

End Comments 46

3. Building the EJB Tier of the DiningGuide Application 47

Overview of the Tutorial's EJB Tier 47

The Entity Beans 48

The Session Bean 49

The Detail Classes 49

Summary of Steps 51

Creating Entity Beans With the EJB Builder 52

Creating the Restaurant and Customerreview Entity Beans 52

Creating Create Methods for CMP Entity Beans 60

Creating Finder Methods on Entity Beans 63

Creating Business Methods for Testing Purposes 65

Creating Detail Classes to View Entity Bean Data 68

Creating the Detail Classes 68

Creating the Detail Class Properties and Their Accessor Methods 69

Creating the Detail Class Constructors 70

Creating Business Methods on the Entity Beans to Fetch the Detail Classes 71

Testing the Entity Beans 72

Creating a Test Client for the Restaurant Bean 73

Providing the Sun ONE Application Server 7 Plugin With Database Information 75

Deploying and Executing the Restaurant Bean's Test Application 77

Using the Test Client to Test the Restaurant Bean 78

Checking the Additions to the Database 81

Creating a Test Client for the Customerreview Bean 82

Deploying and Executing the Customerreview Bean's Test Application 84

Testing the Customerreview Entity Bean 84

Checking the Additions to the Database 86

Creating a Session Bean With the EJB Builder 87

Coding a Session Bean's Create Method 88

Creating Business Methods to Get the Detail Data 91

Creating a Business Method to Create a Customer Review Record 95

Creating Business Methods That Return Detail Class Types 96

Adding EJB References 98

Testing the Session Bean 100

Creating a Test Client for a Session Bean 100

Adding Entity Bean References to the EJB Module 101

Providing the Sun ONE Application Server 7 Plugin With Database Information 102

Deploying and Executing the Test Application 104

Using the Test Client to Test a Session Bean 106

Checking the Additions to the Database 109

Comments on Creating a Client 109

4. Creating the DiningGuide Application's Web Service 111

Overview of the Tutorial's Web Service 111

The Web Service 112

The Runtime Classes 112

The Client Files 112

Creating the Tutorial's Web Service 113

Creating the Logical Web Service 113

Generating the Web Service's Runtime Classes 115
Testing the Web Service 117
Creating a Test Client and Test Application 117
Adding the Web Service to the J2EE Application 118
Deploying the Test Application 119
Using the Test Application to Test the Web Service 121
Making Your Web Service Available to Other Developers 129
Generating the WSDL File 129
Generating Client Files From the WSDL File 130

5. Creating a Client for the Tutorial Application 133
Creating the Client With the Provided Code 133
Running the Tutorial Application 134
Examining the Client Code 137
Displaying Restaurant Data 137
Displaying Customer Review Data for a Selected Restaurant 138
Creating a New Customer Review Record 141

A. DiningGuide Source Files 145
RestaurantBean.java Source 146
RestaurantDetail.java Source 149
CustomerreviewBean.java Source 154
CustomerreviewDetail.java Source 157
DiningGuideManagerBean.java Source 160
RestaurantTable.java Source 164
CustomerReviewTable.java Source 168

B. DiningGuide Database Script 173
Script for a PointBase Database 174
Script for an Oracle Database 175

C. Creating the Tutorial with an Oracle Database 177

Setting up Database Connectivity with the Oracle Database 177

Enabling the Oracle Type 4 JDBC Driver 178

Connecting the IDE to the Oracle Server 179

Creating a JDBC Connection Pool 180

Creating a JDBC Data Source 182

Creating a JDBC Persistent Manager 182

Creating the Database Tables 183

Creating EJB Components with an Oracle Database 185

Creating the Web Service with an Oracle Database 186

Index 187

Figures

FIGURE 2-1 DiningGuide Application Architecture 40

FIGURE 3-1 Function of a Detail Class 50

Tables

TABLE 1-1 Admin Server Property Values 25

TABLE 1-2 DiningGuide Database Tables 33

TABLE 1-3 Restaurant Table Records 33

TABLE 1-4 CustomerReview Table Records 33

TABLE C-1 Oracle-Specific Changes to Chapter 3 185

TABLE C-2 Oracle-Specific Changes to Chapter 4 186

Before You Begin

Welcome to the Sun™ ONE Studio 5 J2EE application tutorial. This tutorial shows you how to use the following features of the Sun ONE Studio 5 integrated development environment (IDE):

- EJB™ 2.0 Builder—for creating and developing Enterprise JavaBeans™ components based on the *Enterprise JavaBeans Specification, Version 2.0*.
- EJB module assembly—for assembling the EJB™ components into an EJB module, which you export into an EJB Java Archive (JAR) file
- Test application facility—for testing enterprise beans without having to create a client manually, using the Sun ONE Application Server 7, Standard Edition software as the application server.
- Web Services features—for building a SOAP web service from the existing EJB component, and generating JSP™ pages viewable from a web browser
- Deploying to the Sun ONE Application Server—for testing the tutorial application

See the release notes for a list of environments in which you can create the example in this book. The release notes are available on this web page:

`http://forte.sun.com/ffj/documentation/index.html`

Screen shots vary slightly from one platform to another. Although almost all procedures use the interface of the Sun ONE Studio 5 software, occasionally you might be instructed to enter a command at the command line. Here too, there are slight differences from one platform to another. For example, a Microsoft Windows command might look like this:

```
c:>cd MyWorkDir\MyPackage
```

A UNIX command might look like this:

```
% cd MyWorkDir/MyPackage
```

Before You Read This Book

This tutorial creates an application that conforms to the architecture documented in Java 2 Platform, Enterprise Edition (J2EE™) Blueprints. If you want to learn how to use the features of Sun ONE Studio 5, Standard Edition to create, develop, and deploy a J2EE compliant application, you will benefit from working through this tutorial.

Before starting, you should be familiar with the following subjects:

- Java programming language
- Enterprise JavaBeans concepts
- Java™ Servlet syntax
- JDBC™ enabled driver syntax
- JavaServer Pages™ syntax
- HTML syntax
- Relational database concepts (such as tables and keys)
- How to use the chosen database
- J2EE application assembly and deployment concepts

This book requires a knowledge of J2EE concepts, as described in the following resources:

- Java 2 Platform, Enterprise Edition Blueprints
  `http://java.sun.com/j2ee/blueprints`
- *Java 2 Platform, Enterprise Edition Specification*
  `http://java.sun.com/j2ee/download.html#platformspec`
- *The J2EE Tutorial*
  `http://java.sun.com/j2ee/tutorial`
- *Java Servlet Specification Version 2.3*
  `http://java.sun.com/products/servlet/download.html#specs`
- *JavaServer Pages Specification Version 1.2*
  `http://java.sun.com/products/jsp/download.html#specs`

Familiarity with the Java API for XML-Based RPC (JAX-RPC) is helpful. For more information, see this web page:

`http://java.sun.com/xml/jaxrpc`

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document and does not endorse and is not responsible or liable for any content, advertising, products, or other materials on or available from such sites or resources. Sun will not be responsible or liable for any damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services available on or through any such sites or resources.

How This Book Is Organized

This manual is designed to be read from beginning to end. Each chapter in the tutorial builds upon the code developed in earlier chapters.

Chapter 1 lists the software requirements for the DiningGuide tutorial, explains how to start the Sun ONE Studio 5 IDE and the Sun ONE Application Server, how to get the IDE and the application server to recognize each other and both communicating with an Oracle database, how to create the tutorial database tables, and then create a database schema in the IDE based on those tables.

Chapter 2 describes the functionality and architecture of the tutorial application.

Chapter 3 provides step-by-step instructions for creating the EJB tier of the tutorial application, and how use the IDE's test application facility to test each bean.

Chapter 4 describes how to use the IDE to generate the tutorial's web service from its EJB tier, and how to test the web service.

Chapter 5 explains how a provided Swing client accesses the output generated from the Web Services module in Chapter 4, and how to run the tutorial application.

Appendix A provides complete source files for the tutorial application.

Appendix B provides the database script for the tutorial application.

Appendix C describes how to adapt the tutorial to create and run the application using an Oracle database.

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.cvspass` file. Use `DIR` to list all files. `Search is complete.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`>` **`login`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide.* These are called *class* options. You *must* save your changes.
AaBbCc123	Command-line variable; replace with a real name or value	To delete a file, type **`DEL`** *filename.*

Related Documentation

Sun ONE Studio 5 documentation includes books delivered in Acrobat Reader (PDF) format, release notes, online help, readme files for example applications, and Javadoc™ documentation.

Documentation Available Online

The documents described in this section are available from the `docs.sun.com`℠ web site and from the documentation page of the Sun ONE Studio Developer Resources portal (`http://forte.sun.com/ffj/documentation`).

The `docs.sun.com` web site (`http://docs.sun.com`) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index installed with the product on your local system or network.

- Release notes (HTML format)

  Available for each Sun ONE Studio 5 edition. Describe last-minute release changes and technical notes.

- *Sun ONE Studio 5, Standard Edition Release Notes* - part no. 817-2337-10

- Getting Started guides (PDF format)

  Describe how to install the Sun ONE Studio 5 integrated development environment (IDE) on each supported platform and include other pertinent information, such as system requirements, upgrade instructions, application server information, command-line switches, installed subdirectories, database integration, and information on how to use the Update Center.

  - *Sun ONE Studio 5, Standard Edition Getting Started Guide* - part no. 817-2318-10
  - *Sun ONE Studio 4, Mobile Edition Getting Started Guide* - part no. 817-1145-10

- Sun ONE Studio 5 Programming series (PDF format)

  This series provides in-depth information on how to use various Sun ONE Studio 5 features to develop well-formed J2EE applications.

  - *Building Web Components* - part no. 817-2334-10

    Describes how to build a web application as a J2EE web module using JSP pages, servlets, tag libraries, and supporting classes and files.

  - *Building J2EE Applications* - part no. 817-2327-10

    Describes how to assemble EJB modules and web modules into a J2EE application, and how to deploy and run a J2EE application.

  - *Building Enterprise JavaBeans Components* - part no. 817-2330-10

    Describes how to build EJB components (session beans, message-driven beans, and entity beans with container-managed or bean-managed persistence) using the Sun ONE Studio 5 EJB Builder wizard and other components of the IDE.

  - *Building Web Services* - part no. 817-2324-10

    Describes how to use the Sun ONE Studio 5 IDE to build web services, to make web services available to others through a UDDI registry, and to generate web service clients from a local web service or a UDDI registry.

  - *Using Java DataBase Connectivity* - part no. 817-2332-10

    Describes how to use the JDBC productivity enhancement tools of the Sun ONE Studio 5 IDE, including how to use them to create a JDBC application.

- Sun ONE Studio 5 tutorials (PDF format)

  These tutorials demonstrate how to use the major features of Sun ONE Studio 5, Standard Edition.

  - *Sun ONE Studio 5 Web Application Tutorial* - part no. 817-2320-10

    Provides step-by-step instructions for building a simple J2EE web application.

  - *Sun ONE Studio 5 J2EE Application Tutorial* - part no. 817-2322-10

    Provides step-by-step instructions for building an application using EJB components and Web Services technology.

- *Sun ONE Studio 4, Mobile Edition Tutorial* - part no. 816-7873-10

  Provides step-by-step instructions for building a simple application for a wireless device, such as a cellular phone or personal digital assistant (PDA). The application will be compliant with the Java 2 Platform, Micro Edition (J2ME™ platform) and conform to the Mobile Information Device Profile (MIDP) and Connected, Limited Device Configuration (CLDC).

You can also find the completed tutorial applications at:
`http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

Online Help

Online help is available inside the Sun ONE Studio 5 IDE. You can open help by pressing the help key (F1 in Microsoft Windows and Linux environments, Help key in the Solaris environment), or by choosing Help → Contents. Either action displays a list of help topics and a search facility.

Examples

You can download examples that illustrate a particular Sun ONE Studio 5 feature, as well as completed tutorial applications, from the Sun ONE Studio Developer Resources portal at:

`http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

The site includes the applications used in this document.

Javadoc Documentation

Javadoc documentation is available within the IDE for many Sun ONE Studio 5 modules. Refer to the release notes for instructions on installing this documentation.

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table.

Type of Documentation	Format and Location of Accessible Version
Books and tutorials	HTML at `http://docs.sun.com`
Mini-tutorials	HTML at `http://forte.sun.com/ffj/tutorialsandexamples.html`
Integrated example readmes	HTML in the example subdirectories of ***s1studio-install-directory***`/examples`
Release notes	HTML at `http://docs.sun.com`

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:

`http://www.sun.com/service/contacting`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Email your comments to Sun at this address:

`docfeedback@sun.com`

Please include the part number (817-2322-10) of your document in the subject line of your email.

CHAPTER 1

Getting Started

This chapter describes what you must do before starting the Sun ONE Studio 5 J2EE Application tutorial. The topics covered in this chapter are:

- "Obtaining and Installing the Required Software," which follows
- "Starting the Software" on page 23
- "Setting Up Database Connectivity" on page 29
- "Tutorial Database Table Descriptions" on page 32

Note – There are several references in this book to the *DiningGuide application files.* These files include a completed version of the tutorial application, a readme file describing how to run the completed application, and SQL script files for creating the required database tables. These files are compressed into a zip file you can download from the Sun ONE Studio 5 Developer Resources portal at `http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

Obtaining and Installing the Required Software

The following items are required to create and run the tutorial.

- Sun ONE Studio 5, Standard Edition software, which includes:
  - The Sun ONE Studio 5, Standard Edition integrated development environment (IDE)

- The Sun ONE Application Server 7 software

  The Sun ONE Studio 5, Standard Edition installer installs both products unless it detects a supported version of the Sun ONE Application Server 7 already installed. The Solaris™ 9 Update 2 Operating Environment, for example, includes an installation of Sun ONE Application Server 7.

  You can obtain the Sun ONE Studio 5, Standard Edition software from:

  - The Sun ONE Studio 5, Standard Edition CD
  - The Sun ONE Portal (`http://wwws.sun.com/software/sundev/jde/`)
  - The Sun ONE Developer Resources portal (`http://forte.sun.com/ffj/`)

- Java™ 2 Software Development Kit (the J2SE™ SDK), version 1.4.1_02 or higher

  The Sun ONE Studio 5, Standard Edition installer will not run if you do not have the J2SE SDK on your system. If you do have a J2SE SDK on your system, the installer will start and then verify whether the J2SE SDK you are using is a version required by the IDE and the Sun ONE Application Server 7 for your platform. If you do not have the required version, the installer will quit, displaying a message that you must install the correct version before proceeding. You can obtain the J2SE SDK from the same locations as the IDE.

- PointBase Network Server database software

  This tutorial uses the PointBase database. PointBase is installed with the Sun ONE Studio 5, Standard Edition software, in the subdirectory that contains the Sun ONE Application Server 7 software. If your IDE and application server were installed separately, your application server may or may not include PointBase software. If it does not, you must download and install PointBase software manually. Instructions are provided in the *Sun ONE Application Server 7 Getting Started Guide.* Alternatively, Appendix C describes how to create the tutorial application with an Oracle database.

- The tutorial database tables

  The tutorial database tables are created for you by the Sun ONE Studio 5 installer in the PointBase database in your user directory. The tables are described in "Tutorial Database Table Descriptions" on page 32. Appendix C describes how to install the tutorial tables in an Oracle database.

- A web browser

  You need a web browser to view the tutorial application pages. This can be either Netscape Communicator™ or Microsoft's Internet Explorer.

You can access general system requirements from the release notes or from the Sun ONE Studio 5 Developer Resources portal's Documentation page at `http://forte.sun.com/ffj/documentation/`.

Starting the Software

This section describes how to start the Sun ONE Studio 5 IDE and Sun ONE Application Server 7 after the software has been installed.

Starting the IDE

There are several ways to start the Sun ONE Studio 5 IDE. Only one is described here. For more options, see the *Sun ONE Studio 5, Standard Edition Getting Started Guide*.

To start the IDE:

- **Start the Sun ONE Studio 5 IDE by running the program executable.**
  - On Microsoft Windows, choose Start → Programs → Sun Microsystems → Sun ONE Studio 5 SE → Sun ONE Studio 5 SE
  - On Solaris, UNIX, and Linux environments, run the `runide.sh` script in a terminal window, as follows:

```
$ sh s1studio-install-directory/bin/runide.sh
```

The *s1studio-install-directory* variable stands for the IDE's home directory, which is by default `$HOME/studio5_se` (UNIX standard user) or `/opt/studio5_se` (UNIX superuser).

Starting the Application Server

Before starting this section, you must have write access to an application server domain. The default domain is created during installation and requires superuser privileges (administrator privileges on Microsoft Windows systems or root privileges in Solaris or Linux environments) to access. If your userid has superuser privileges, you can start the application server using all default settings, as described in the next section. Standard users (without superuser privileges) must use the procedures described in "Starting the Admin Server (Standard User)" on page 25.

Starting the Admin Server (Superuser)

If you have started the admin server previously, confirm whether it is running. See "Confirming Sun ONE Application Server 7 as the Default Server" on page 29 for information. If this is the first time you have started the admin server, start with this section.

To start the admin server:

1. **In the IDE, select the Runtime tab of the Explorer.**

   The Runtime pane of the Explorer displays the Server Registry node, among other nodes. This node contains subnodes for all the installed web and application servers, and a node showing which servers are the default servers.

2. **Select the Server Registry node.**

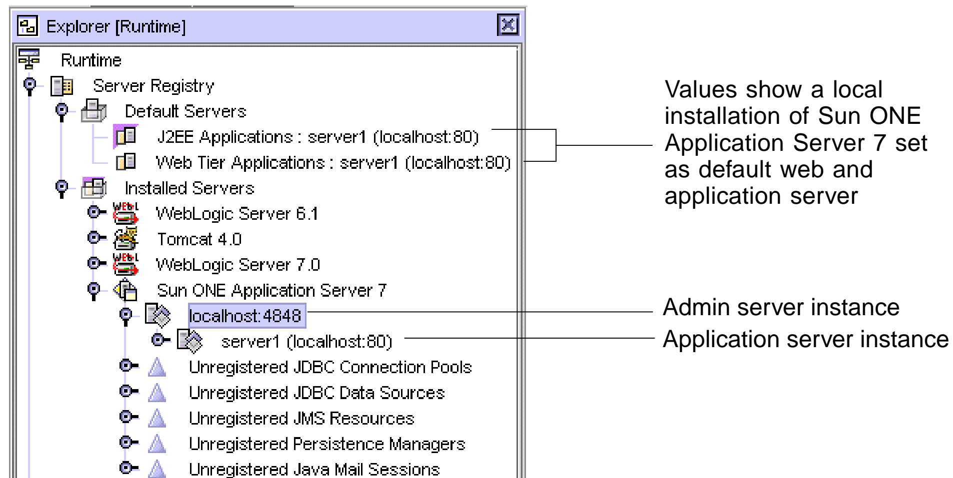
   A query window pops up, asking whether you want to start the admin server. This refers to the default domain's admin server, which can only be run by a privileged user.

3. **Click OK to start the default admin server.**

   The IDE starts the default admin server and configures Sun ONE Application Server 7 as the IDE's default application server. It also creates a server instance, server1.

4. **Expand the Server Registry node, the Installed Servers node, and the Sun ONE Application Server 7 node.**

   The Server Registry in the Explorer looks like this:



Now, start the server instance, as described in "Starting the Application Server Instance" on page 28.

Starting the Admin Server (Standard User)

If your userid does not have superuser privileges, a superuser must create a domain for you before you start this section. The procedures are described in the *Sun ONE Studio 5, Standard Edition Getting Started Guide*.

If you have started the IDE previously, created an admin server, and started it, confirm now whether it is running by using the procedures described in "Confirming Sun ONE Application Server 7 as the Default Server" on page 29. If this is the first time you have started the admin server, continue with this section.

Before starting the procedures described in this section, you will need values of several properties of your domain. Your administrator can provide these for you. Use the following table to record your values to these properties.:

TABLE 1-1 Admin Server Property Values

Admin Server Properties	Your Value
Admin Server Host	
Admin Server Port	
User Name	
User Password	
Domain	

To start the admin server:

1. **In the IDE, select the Runtime tab of the Explorer.**

   The Runtime pane of the Explorer displays the Server Registry node, among other nodes. This node contains subnodes for all the installed web and application servers, and a node showing which servers are the default servers.

2. **Select the Server Registry node.**

   A query window pops up, asking whether you want to start the admin server. This refers to the admin server of the default domain, which can only be run by a privileged user.

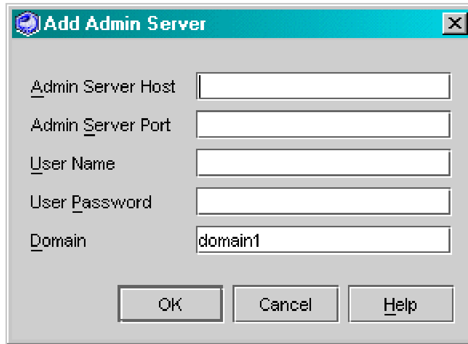
   If you click OK, this action creates and starts an admin server that you cannot use. Click Cancel.

3. **Add your admin server to the IDE.**

   a. **Expand the Server Registry node and the Installed Servers node.**

   b. **Right-click the Sun ONE Application Server 7 node and choose Add Admin Server.**

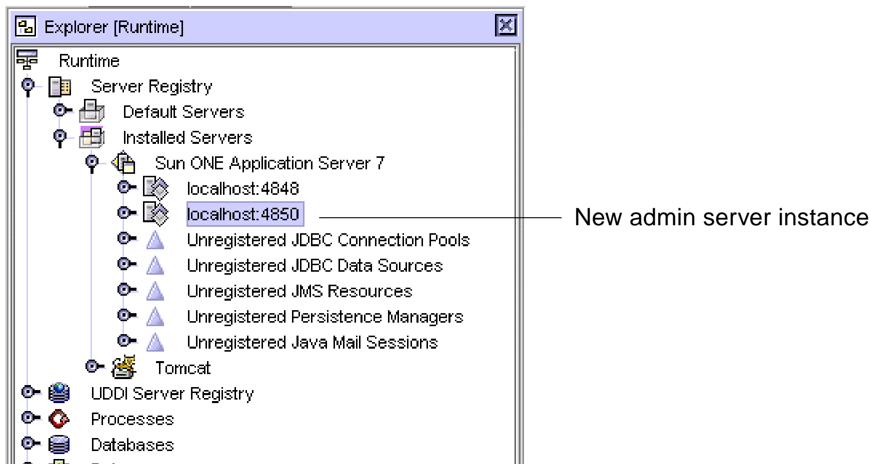
      The Add Admin Server dialog box is displayed.



c. **Type in your values (see** TABLE 1-1**) and click OK.**

If an error message appears, stating that the IDE could not locate the admin server, but will start it if it is local, click OK to close the error window. A progress window appears, showing the admin server process starting up.

A new admin server node is generated in the Explorer. In the following screen shot, for example, a new admin server's host is localhost and port number is 4849.



4. **Create an application server instance.**

a. **Right-click the new admin server node and choose Create a Server Instance.**

The Enter Server Instance Values dialog box is displayed.

b. **Type in a name and an available port number.**

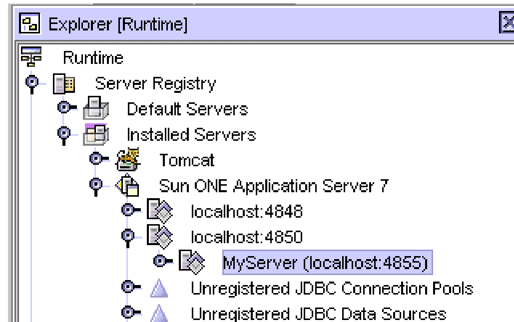
For example, you could type `MyServer` and `4855`.

Note – On Solaris and Linux systems, port numbers below 1024 are reserved. Use a port number above 1023. On all systems, do not use the port number used by the default application server, which is 80.

---

c. **Click OK.**

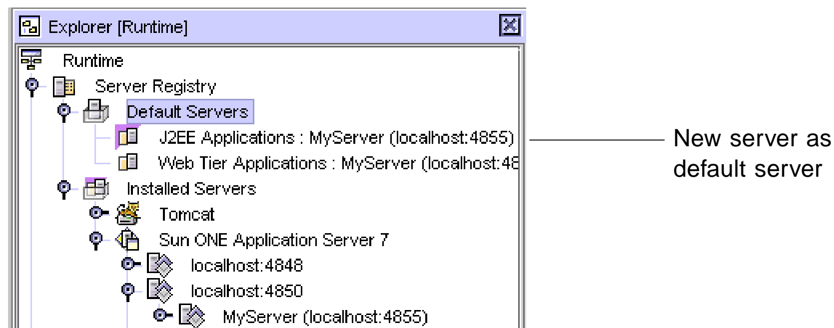
This action starts the admin server, which you can verify by messages in the output window and status bar. The new server instance is created in the IDE.



5. **Set the default application and web server by right-clicking the new server instance and choosing Set As Default.**

6. **Expand the Default Servers node to verify this action.**

The default servers for J2EE applications and web tier applications show the new server as the default.



Now start the server instance, as described in the next section.

Starting the Application Server Instance

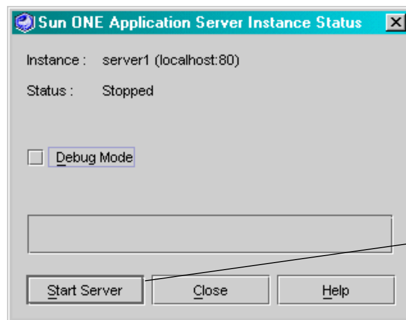
When you are test deploying and deploying applications during development, the IDE starts the application server instance automatically as long as the admin server is running. In this section, you start the application server instance manually in order to perform some operations described later in this chapter.

All users start the server instance as follows:

1. **Right-click the application server node and choose Status.**

Note – If this node is not displayed, select the admin server instance node and choose Refresh.

The Sun ONE Application Server Instance Status dialog box is displayed, as shown (your instance label may be different).



2. **Click the Start Server button.**

   (If the dialog box has a Stop Server button, the server is already running.)

   On Microsoft Windows systems, a command window appears, displaying progress messages.

   The server is started when the Server Instance Status window displays Status: Running.

3. **Click Close on the instance status dialog box.**

   Now, proceed to “Setting Up Database Connectivity” on page 29.

Confirming Sun ONE Application Server 7 as the Default Server

If you have started Sun ONE Application Server 7 before, this is how you confirm that it is still the default server:

1. **In the IDE, select the Runtime tab of the Explorer.**
2. **Expand the Server Registry node and its Default Servers subnode.**

   If the J2EE Applications node's label is *server-instance*`(`*server-hostname*`:`*server-port-number*`)`, as shown, then Sun ONE Application Server 7 is the default application server. Go to the next section. Otherwise, continue with the next step.

Value shows a local installation of Sun ONE Application Server 7 set as default application server

3. **Find your application server instance under the Installed Servers node, right-click it, and choose Set As Default.**

   Your server is set as the default server for J2EE and Web Tier applications.

Setting Up Database Connectivity

An enterprise application uses the Java Database Connectivity (JDBC™) API to interact with a database. Before you can deploy and execute an enterprise application with Sun ONE Application Server 7, the following required JDBC-related actions must be performed in the application server environment. These include:

- Enabling the database's JDBC driver
- Creating a connection pool

  Enterprise applications require pooling of database connections so that the business objects in the system can share database access.
- Creating a JDBC data source

  A JDBC data source (also called a JDBC resource) lets you make connections to a database with the `getConnection()` method.
- Creating a JDBC persistent manager

  A persistent manager is a component responsible for the persistence of the entity beans installed in the container.

Under most circumstances, the Sun ONE Studio 5 IDE performs all these actions for you, providing the values required by the tutorial. Depending on the privileges of your userid and the platform you are using, these are performed with different methods, which are described in "Setting JDBC Resources (Microsoft Windows Superusers)" on page 30 and "Setting JDBC Resources (All Other Users)" on page 31.

Exceptions to automatic enabling of the JDBC driver, however, are not dependent on your userid, as the next section describes.

Enabling the JDBC Driver

The PointBase JDBC driver is installed automatically with the standard installation, where the Sun ONE Studio 5 IDE and Sun ONE Application Server 7 are installed together.

Note – If the Sun ONE Studio 5 IDE and Sun ONE Application Server 7 were installed separately, there are actions you must take. These are described in the *Sun ONE Studio 5, Standard Edition Getting Started Guide*.

Setting JDBC Resources (Microsoft Windows Superusers)

The Sun ONE Studio 5 installer automatically creates the three connectivity resources for the Microsoft Windows user with administrator privileges.

Note – If the Sun ONE Studio 5 IDE and Sun ONE Application Server 7 were installed separately, you must configure the JDBC resources with the procedure described in "Setting JDBC Resources (All Other Users)" on page 31.

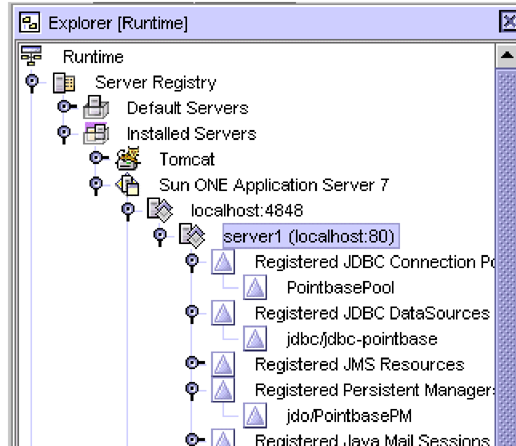
To view the resources in the IDE:

1. **In the Runtime pane of the Explorer, expand the Server Registry and Installed Servers nodes.**
2. **Expand the default admin server node (localhost:4848) and the default application server node (server1(localhost:80)).**
3. **Expand the registered resources for JDBC Connection Pools, JDBC Data Sources, and Persistent Managers.**

You should see the following nodes under these:

- JDBC Connection Pools: PointbasePool
- JDBC DataSources: jdbc/jdbc-pointbase
- Persistent Managers: jdo/PointbasePM

The Explorer looks like this:



Setting JDBC Resources (All Other Users)

Note – Before starting this procedure, make sure both the admin server and the application server are running (refer to "Starting the Software" on page 23).

To create the JDBC connection resources for this tutorial:

1. **In the Runtime pane of the Explorer, expand the Server Registry and Installed Servers nodes.**

2. **Locate your application server instance.**

   It is labeled *app-server-name* (*app-server-host*:*app-server-port*), for example, MyServer (localhost:4855).

3. **Right-click the application server instance node and choose Preconfigure PointBase JDBC Resources.**

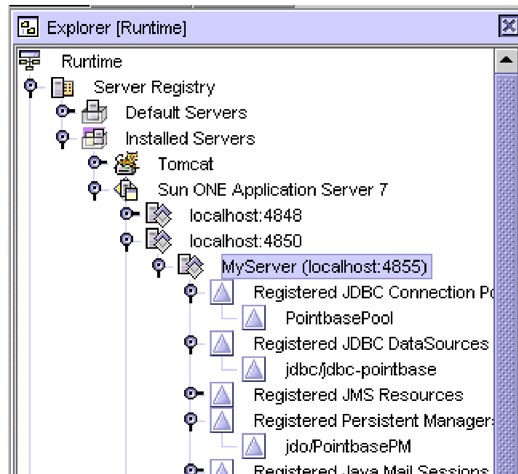
   A timer icon (such as an hour glass) appears. When the process is complete, the cursor reverts to its usual icon.

4. **Expand the registered resources for JDBC Connection Pools, JDBC Data Sources, and Persistent Managers.**

   You should see the following nodes under these:

   - JDBC Connection Pools: PointbasePool
   - JDBC DataSources: jdbc/jdbc-pointbase
   - Persistent Managers: jdo/PointbasePM

   The Explorer looks like this:



   Now proceed to the tutorial's database tables.

Tutorial Database Table Descriptions

The DiningGuide tutorial uses two database tables, Restaurant and CustomerReview. These tables are automatically created for you in the PointBase default (sample) database with the standard installation, where Sun ONE Studio 5 software and Sun ONE Application Server 7 are installed together.

However, if Sun ONE Studio 5 software and Sun ONE Application Server 7 were not installed together, you must follow the procedures described in "Enabling the JDBC Driver" on page 30. This action not only enables the PointBase JDBC driver, but copies the default sample database to your user directory, which makes these tables available to you.

The database schemas shown in TABLE 1-2 are used in this tutorial. They are created in the PointBase database by the Sun ONE Studio 5 installer.

TABLE 1-2 DiningGuide Database Tables

Table Name	Columns	Primary Key	Other
Restaurant	restaurantName	yes	
	cuisine		
	neighborhood		
	address		
	phone		
	description		
	rating		
CustomerReview	restaurantName	yes	Compound primary key with CustomerName; references Restaurant(restaurantName)
	customerName	yes	
	review		

The Restaurant table contains the records shown in TABLE 1-3.

TABLE 1-3 Restaurant Table Records

restaurant-Name	cuisine	neighborhood	address	phone	description	rating
French Lemon	Mediterranean	Rockridge	1200 College Avenue	510 888 8888	Very nice spot.	5
Bay Fox	Mediterranean	Piedmont	1200 Piedmont Avenue	510 888 8888	Excellent.	5

The CustomerReview table contains the records shown in TABLE 1-4.

TABLE 1-4 CustomerReview Table Records

restaurantName	customerName	comment
French Lemon	Fred	Nice flowers.
French Lemon	Ralph	Excellent Service

Now you are ready to start the tutorial application. Either continue to Chapter 2 to get an overview of the application you will build, or go directly to Chapter 3 and start building it.

CHAPTER 2

Introduction to the Tutorial

In the process of creating the tutorial example application, you will learn how to build a simple J2EE application using Sun ONE Studio 5, Standard Edition features.

This chapter describes the application you will build, first describing its requirements, and then presenting an architecture that fulfills the requirements. The final section describes how you use Sun ONE Studio 5, Standard Edition features—the EJB Builder, the test application facility, and the New Web Service wizard—to create the application.

This chapter is organized into the following sections:

- "Functionality of the Tutorial Application," which follows
- "User's View of the Tutorial Application" on page 37
- "Architecture of the Tutorial Application" on page 40
- "Overview of Tasks for Creating the Tutorial Application" on page 42

Functionality of the Tutorial Application

The tutorial application, DiningGuide, is a simple dining guide application that enables users to view a list of available restaurants and their features. The user can also view a list of a selected restaurant's customer reviews, and add a review to a restaurant's record. The restaurant features include the restaurant name, its cuisine type, its neighborhood, address, and phone number, a brief description of the restaurant, and a rating number (1 - 5).

The user interacts with the application's interface as follows:

- The user views a complete list of restaurants
- The user requests a list of customer reviews for a particular restaurant
- The user writes a review and adds it to the restaurant's list of reviews

Application Scenarios

The interaction of DiningGuide begins when the user executes a client page listing all the restaurant records in the database. The interaction ends when the user quits the application's client. A simple Swing client is provided to illustrate how a user can interact with the application's features. However, other types of clients, such as a web client or another application, could access the business methods of the DiningGuide application.

The following scenarios illustrate interactions that happen within the application, and the application's requirements.

1. The user executes the application's `RestaurantTable` class.

   The application displays the DiningGuide Restaurant Listing window, which displays a list of all restaurants, their names, cuisine type, location, phone number, a short review comment, and a rating from 1 to 5. On the page is a button labeled View Customer Comments.

2. The user selects a restaurant record in the list and clicks the View Customer Comments button for a given restaurant.

   The application displays a All Customer Reviews By Restaurant Name window with a list of all the reviews submitted by customers for the selected restaurant.

3. On the customer review window, the user types text into the Customer Name and Review fields and clicks the Submit Customer Review button.

   The application adds the customer's name and review text to the CustomerReview database table, and redisplays the All Customer Reviews By Restaurant Name window with the new record added.

4. The user returns to the Restaurant Listing window, selects another restaurant, and clicks the View Customer Comments button.

   The application displays a new All Customer Reviews By Restaurant Name window showing all the reviews for the selected restaurant.

Application Functional Specification

The following items list the main functions for a user interface of an application that supports the application scenarios.

- A master view of all restaurant data through a displayed list
- A button on the master restaurant list window for retrieving all customer review data for a given restaurant
- A master view of all customer review data for a given restaurant
- A button on the customer review list window for adding a new review

- Text entry fields on the customer review list window for typing in a new customer name and new customer review for the current restaurant
- A button on the customer review list window for submitting the finished review data to the database

User's View of the Tutorial Application

The user's view of the application illustrates how the scenarios and the functional specification, described in "Functionality of the Tutorial Application" on page 35 are realized.

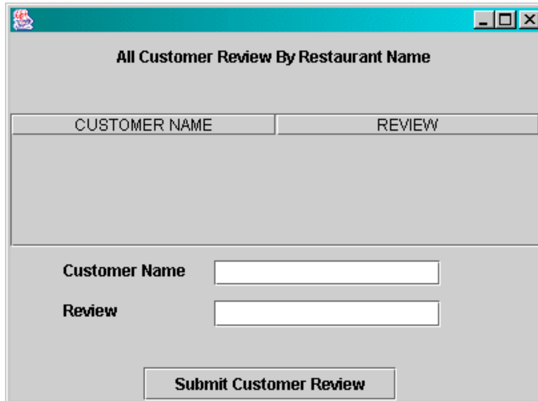
1. **In the Sun ONE Studio 5 Explorer, right-click the RestaurantTable node and choose Execute.**

   The IDE switches to Runtime mode. A Restaurant node appears in the execution window. Then, the RestaurantTable window is displayed, as shown:

   This window displays the data from the Restaurant table described in "Tutorial Database Table Descriptions" on page 32.

2. **To view the customer reviews for a given restaurant, select the restaurant name and click the View Customer Comments button.**

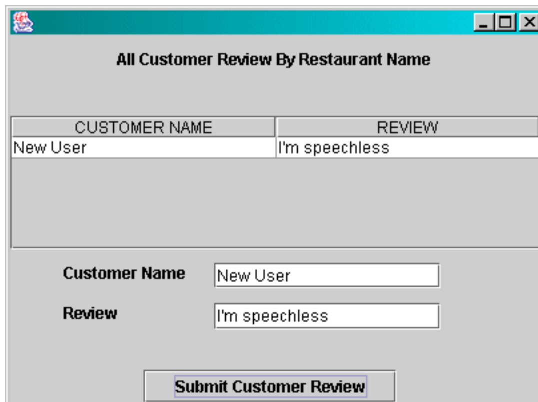
   For example, select the Bay Fox restaurant. The CustomerReviewTable window is displayed.



In this case, no records are shown, because none are in the database. Refer to TABLE 1-4.

3. **To add a review, type in a customer name and some text for the review and click the Submit Customer Review button.**

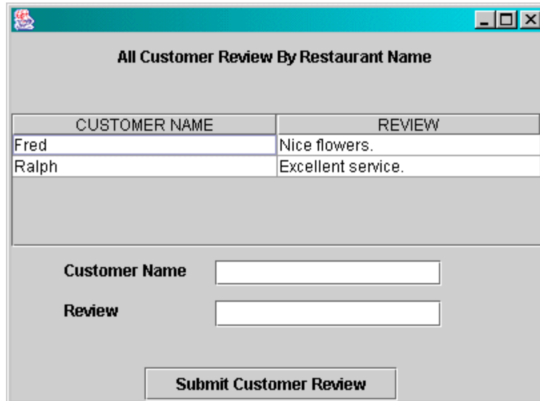
   For example, type in `New User` for the name and `I'm speechless!` for the review. The application redisplays the customer review window, as shown:



Now, display the reviews of the other restaurant.

4. **On the Restaurant List window, select French Lemon and click the View Customer Comments button.**

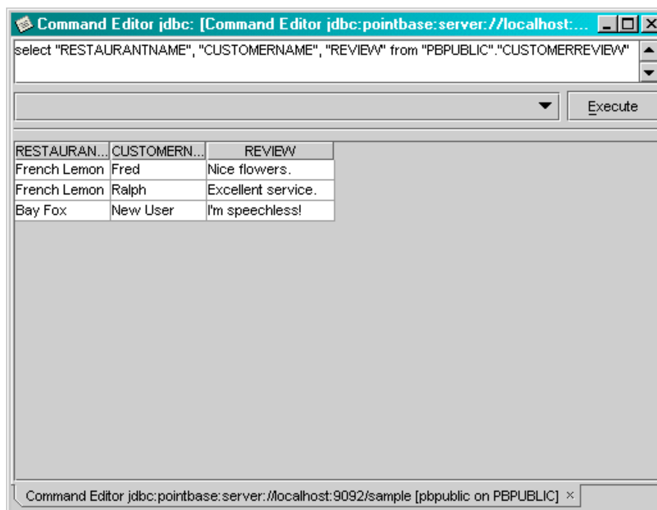
   A new customer review list window is displayed, showing the comments for the French Lemon restaurant.



Two customer review records are displayed. Refer to TABLE 1-4 for confirmation.

5. **Continue to add and view customer review records.**
6. **When you are done, quit the application by closing any of the application's windows.**
7. **To verify that the new customer review records were written to the database, in the IDE, select the Runtime tab of the Explorer.**
8. **Expand the Databases node, the PointBase connection node, and the Tables node under it.**
9. **Right-click the CUSTOMERREVIEW table and choose View Data.**

   A command editor window is displayed, showing any new CustomerReview records you entered in Step 3, for example:



Architecture of the Tutorial Application

The heart of the tutorial application is the EJB tier that contains two entity type enterprise beans, two detail classes, and a session bean. The entity beans represent the two DiningGuide database tables (Restaurant and CustomerReview); the two detail classes mirror the entity bean fields and include getter and setter methods for each field. The detail classes are used to reduce the number of method calls to the entity beans when retrieving database data. The session bean manages the interaction between the client (by way of the web service) and the entity beans.

FIGURE 2-1 shows the DiningGuide application architecture.

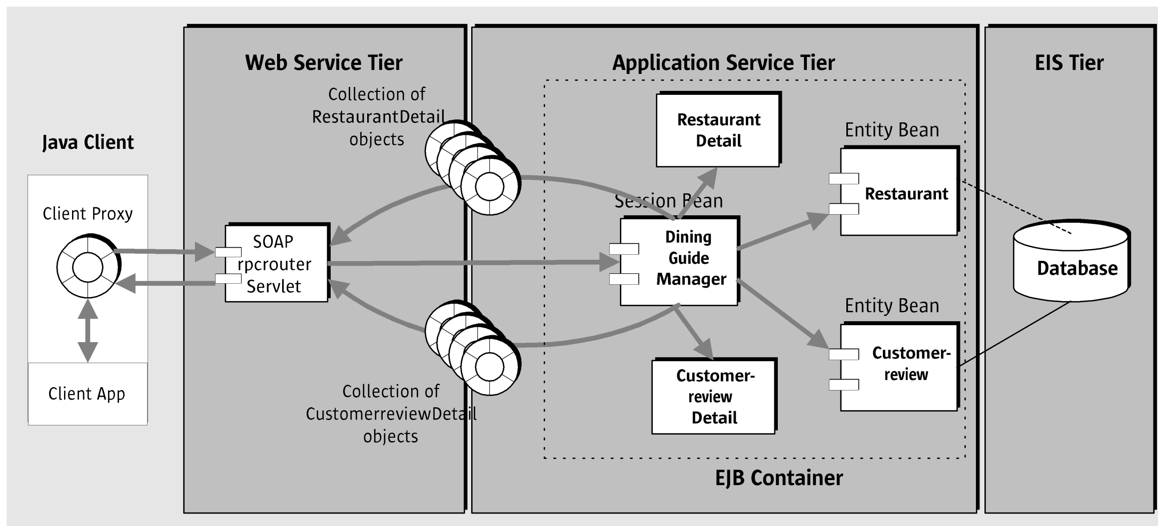


FIGURE 2-1 DiningGuide Application Architecture

In FIGURE 2-1, the client includes a client proxy, which uses the SOAP runtime system to communicate with the SOAP runtime system on the web server. Requests are passed as SOAP messages. The web service translates the SOAP messages into calls on the EJB tier's session bean's methods. The session bean passes its responses back to the web service, which translates them into SOAP messages to give to the client proxy and ultimately get translated into a display of data or action.

A SOAP request is an XML wrapper that contains a method call on the web service and input data in serialized form.

Application Elements

The elements shown in FIGURE 2-1 are:

- An application service tier (an EJB tier)

  You build and test the EJB tier before you build anything else in the tutorial. The EJB tier consists of:

  - Two entity enterprise beans that use container-managed persistence (CMP) to represent the two database tables of the application
  - Two detail classes to hold returned database records
  - A stateless session enterprise bean to manage the requests from the client and to format the objects returned to the client.

- The web service tier
  - A web module containing Servlets and JSP pages for exercising the session bean's methods

    This is automatically created when a test application is built for the session bean.

  - A web service logical node that represents the entire web service and enables modification and configuration of the web service
  - A client proxy that is generated when the web service is deployed
  - A WSDL (Web Services Descriptive Language) file that described the web service for a client

- The client

  The client component is a Swing client that displays the application pages. In Chapter 5, you copy code from provided client pages that instantiate the client proxy created in the web service in Chapter 4.

EJB Tier Details

The EJB tier of the DiningGuide application contains two entity-type enterprise beans, two detail classes, and a session bean used to manage the interaction between the client and the entity beans.

- Restaurant CMP EJB component

  The Restaurant bean is an entity bean that uses container-managed persistence (CMP) to represent the data of the Restaurant database table.

- Customerreview CMP EJB component

  Also a CMP-type entity bean, the Customerreview entity bean represents the data from the CustomerReview database table.

- RestaurantDetail class

  This component has the same fields as the Restaurant entity bean, plus getter and setter methods for each field for retrieving this data from the entity bean's remote reference. Its constructor instantiates an object that represents the restaurant data. This object can then be formatted into a JSP page, HTML page, or Swing component for the client to view.

- CustomerreviewDetail class

  This component serves the same function for the Customerreview entity bean that the RestaurantDetail class serves for the Restaurant entity bean.

- DiningGuideManager session EJB component

  This component is a stateless session bean that is used to manage the interaction between the client and the entity beans.

Overview of Tasks for Creating the Tutorial Application

The tutorial building process is divided into three chapters. In the first (Chapter 3), you create the EJB tier and use the IDE's test application facility to test each enterprise bean as you work. Then you create a session bean to manage traffic. This is a common model when creating the web services and the client manually.

In the second chapter (Chapter 4), you create a web service and specify which of the EJB tier's business methods to reference. You deploy the web service, which generates a client proxy, and then you test the client proxy.

In the final chapter (Chapter 5), you install two provided Swing classes into the application and execute them to test the application.

Creating the EJB Components

In Chapter 3 you learn how to use Sun ONE Studio 5 features to:

- Build entity and session beans quickly with the EJB Builder
- Generate classes (with getter and setter methods) from a database schema
- Use the test application facility to assemble a test J2EE application from enterprise beans
- Add EJB references to a J2EE application

- Deploy the test application to the J2EE Reference Implementation application server
- Exercise enterprise bean methods from the test client page created by the test application facility.

Using the EJB Builder

The EJB Builder wizard automatically creates the various components that make up an enterprise bean, whether it's a stateless or stateful session bean, or an entity bean with container-managed persistence (CMP) or bean-managed persistence (BMP). In Chapter 3, you create two CMP entity beans based on existing database tables, and a stateless session bean.

When you create the entity beans, you learn how to connect to a database during the creation process, and then generate an entity bean whose fields represent the table's columns. The basic parts of the bean are generated into the Sun ONE Studio 5 Explorer with Java code already generated for the home interface, remote interface, bean class, and (if applicable) the primary key class. You learn how to edit and modify the bean properly by using the logical bean node, which represents the bean as a whole. You learn to add create, finder, and business methods using the EJB Builder's GUI features.

Creating the Detail Classes

The detail classes must have the same fields as the entity beans. You create two classes and add the appropriate bean properties to them. While adding each property, you activate an option that automatically generates accessor methods for the property. This way, you obtain the getter and setter methods the application requires. Then you code each class's constructor to instantiate the properties. Finally, you add code to each of the two entity beans to return an instance of its corresponding detail class.

Using the Test Application Facility

The Sun ONE Studio 5 IDE includes a facility for testing enterprise JavaBean components without your having to create a client for this purpose. This facility uses Sun ONE Application Server 7 as the application server and deploys the enterprise bean as part of a J2EE application that includes a web module and client JSP pages. An HTML page coordinates these JSP pages so that, from a web browser, you can create an instance of the bean and then exercise its business methods.

You create test applications for all three of the enterprise beans separately. For the entity beans, the test application generates a J2EE application that contains a web module, which contains the automatically generated JSP pages for the client's use from a web browser, and an EJB module for the entity bean. The session bean's EJB module must also contain the EJB modules of the entity beans, because it calls methods on those entity beans. You add the entity bean references to the session bean's EJB module using commands in the IDE. The EJB module created while creating the test application is referenced later by the web service.

When you test the session bean in a web browser, you can exercise all the application's business methods. At the end of Chapter 3 are guidelines for using the test client apparatus to guide you if you want to create your own web service and client manually.

Creating the Tutorial's Web Service

In Chapter 4 you learn how to use Sun ONE Studio 5 features to:

- Create a logical web service
- Specify which session business methods are to be referenced by the web service
- Create a J2EE application to contain the web service
- Generate the web service's runtime classes and client pages
- Generate the web service's client proxy

Creating a Web Service

A web service is a logical entity that represents the entire set of objects in the web service, and facilitates modifying and configuring the web service. You create a web service in the Explorer using the New wizard to define its name and package location. As you create the web service, the wizard prompts you to specify the business methods you want the web service to reference.

You inform the web service of the location of the JAX-RPC runtime by specifying its URL as a property of the web service. You then generate the web service's runtime classes, which are EJB components that implement the web service.

Creating a Test Client for the Tutorial

You create a test client that consists of front-end client and a back-end J2EE application. You then add references to the session bean's EJB module and to the web service. This action makes the web service's WAR and EJB JAR files available, so you can customize their properties. One property that you customize is the Web Context property. This completes the DiningGuide's J2EE application, and you are ready to deploy it.

Deploying the Web Service and Creating a Test Client

When you deploy the J2EE application that contains the web service, the IDE automatically generates a client proxy and supporting files. The supporting files include a JSP page for each referenced method, a JSP error page, and a welcome page.

Testing the Web Service

You use an IDE command to deploy the DiningGuide application. This starts the application server and displays the test client's welcome page that displays all the operations on one page. The generated JSP pages contain input fields when an input parameter is required, and an Invoke button to execute the operation. You use these means to test how the web service calls each of the session bean's methods.

Making a Web Service Available to Other Developers

Although this tutorial does not describe how to publish the web service to a UDDI registry, it does describe an informal method for enabling other developers to use the web service for testing purposes. You learn how to generate a WSDL file, which you can then make available, either by placing it on a server, or by distributing it some other way, such as by email. The target developers can generate a client proxy from this file and discover which methods are available on your web service. They can then build a client accordingly, and, if you provide them with the URL of your deployed web service, they can test their client against your web service.

The Sun ONE Studio 5 IDE also provides a single-user internal UDDI registry for testing purposes. The StockApp example, available from the Examples and Tutorials page of the Sun ONE Studio 5 Developer's portal, demonstrates how to publish a web service using this device. The Examples and Tutorials page is at:

```
http://forte.sun.com/ffj/documentation/tutorialsandexamples.html
```

See *Building Web Services* in the Sun ONE Studio 5 Programming series for complete information about publishing a web service to a UDDI registry.

Installing and Using the Provided Client

Code for a simple Swing client that demonstrates the functionality of the DiningGuide application is provided in Appendix A. This client consists of a Swing class for each of the database tables. You create two classes and then replace their default code with the provided code. Then, you simply execute the main class.

You learn by examining the provided code how a client accesses the application's methods. First, the client must instantiate the client proxy. This makes the client proxy's methods available to the client. These methods (see FIGURE 2-1) are used by the SOAP runtime to access the methods of the application's EJB tier.

End Comments

This tutorial application is designed to be a running application that illustrates the main features of Sun ONE Studio 5, Standard Edition, while still brief enough for you to create in a short time (perhaps a day). This places certain restrictions on its scope, for example:

- There is no error handling
- There are no debugging procedures
- Publishing the web service is not described

Although the tutorial application described in this book is designed to be a simple application that you can complete quickly, you might want to import the entire application, view the source files, or copy and paste method code into methods you create. The DiningGuide application is accessible from the Examples and Tutorials page of the Sun ONE Studio 5 Developer's portal at:

`http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

CHAPTER 3

Building the EJB Tier of the DiningGuide Application

This chapter describes, step by step, how to create the EJB tier of the DiningGuide tutorial application. Along the way, you learn how to use the EJB Builder to create both entity and session beans, and how to use the IDE's test mechanism to test the beans. The topics covered in this chapter are:

- "Overview of the Tutorial's EJB Tier," which follows
- "Creating Entity Beans With the EJB Builder" on page 52
- "Creating Detail Classes to View Entity Bean Data" on page 68
- "Testing the Entity Beans" on page 72
- "Creating a Session Bean With the EJB Builder" on page 87
- "Testing the Session Bean" on page 100
- "Comments on Creating a Client" on page 109

By the end of this chapter, you will be able to run the whole EJB tier of the DiningGuide application as a deployed test application.

After you have created the EJB tier, you are free to create your own web services and client pages. Alternatively, you can continue on to Chapter 4, to learn how to create the application's web services using the Sun ONE Studio 5 Web Services features.

Overview of the Tutorial's EJB Tier

In this chapter, you create the module that is the heart of the tutorial application, namely, its EJB tier. As you create each component, you test it using the IDE's test application facility, which automatically creates a test web service and test client.

The EJB tier you create will include:

- a Restaurant entity bean
- a Customerreview entity bean
- a DiningGuideManager session bean
- a RestaurantDetail bean
- a CustomerreviewDetail bean

For a complete discussion of the role of the EJB tier within J2EE architecture, see *Building Enterprise JavaBeans Components* in the Sun ONE Studio 5 Programming series. That document provides full descriptions of all the bean elements, and explains how transactions, persistence, and security are supported in enterprise beans.

To examine an application that also uses an EJB tier and a web service generated from it, see the PartSupplier example on the Sun ONE Studio 5 examples page, `http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

The Entity Beans

An entity bean provides a consistent interface to a set of shared data that defines a concept. In this tutorial, there are two concepts: *restaurant* and *customer review*. The Restaurant and Customerreview entity beans that you create represent the database tables you created in Chapter 1.

Entity beans can have either container-managed persistence (CMP) or bean-managed persistence (BMP). With a BMP entity bean, the developer must provide code for mapping the bean's fields to the database table columns. With a CMP entity bean, the EJB execution environment manages persistence operations. In this tutorial, you use CMP entity beans. Using the IDE's EJB Builder wizard, you connect to the database and indicate which columns to map. The wizard creates the entity beans mapped to the database.

The EJB Builder creates the CMP entity bean's framework, including the required home interface, remote interface, and bean class. The wizard also creates a logical node to organize and facilitate customization of the entity bean.

You manually define the entity bean's create, finder, and business methods. When you define these methods, the IDE automatically propagates the method to the appropriate bean components. For example, a create method is propagated to the bean's home interface and a corresponding ejbCreate method to the bean's class. When you edit the method, the changes are propagated as well.

With finder methods, you must define the appropriate database statements to find the objects you want. The EJB 2.0 architecture defines a database-independent version of SQL, called EJB QL, which you use for your statements. At deployment, the Sun ONE Application Server plugin translates the EJB QL into the SQL appropriate for your database and places the SQL in the deployment descriptor.

The Session Bean

Entity beans represent shared data, but session beans access data that spans concepts and is not shared. Session beans can also manage the steps required to accomplish a particular task. Session beans can be stateful or stateless. A *stateful* session bean performs tasks on behalf of a client while maintaining a continued conversational state with the client. A *stateless* session bean does not maintain a conversational state and is not dedicated to one client. Once a stateless bean has finished calling a method for a client, the bean is available to service a request from a different client.

In the DiningGuide application, client requests might include obtaining data on all the restaurants in the database or finding all the customer reviews for a given restaurant. Submitting a review for a given restaurant is another client request. These requests are not interrelated, and don't require maintenance of a conversational state. For these reasons, the DiningGuide tutorial uses a stateless session bean to manage the different steps required for each request.

The session bean repeatedly builds collections of restaurant and customer review records to satisfy a client's request. This task could be accomplished by adding getter and setter methods for each field onto the entity beans, but this approach would require calling a method for every field each time the session bean has to retrieve a row of the table. To reduce the number of method calls, this tutorial uses special helper classes, called *detail* classes, to hold the row data.

The Detail Classes

A detail class is a Java class that has fields that correspond to the container-managed fields of the entity beans, plus getter and setter methods for each field. When the session bean looks up an entity bean, it uses the corresponding detail class to create an instance of each remote reference returned by the entity bean. The session bean just calls the detail class's constructor to instantiate a row of data for viewing. In this way, the session bean can create a collection of row instances that can be formatted into an HTML page for the client to view. Returning detail class instances to the client consumes less network bandwidth than returning remote references for the entity bean instances.

FIGURE 3-1 shows graphically how the detail classes work.

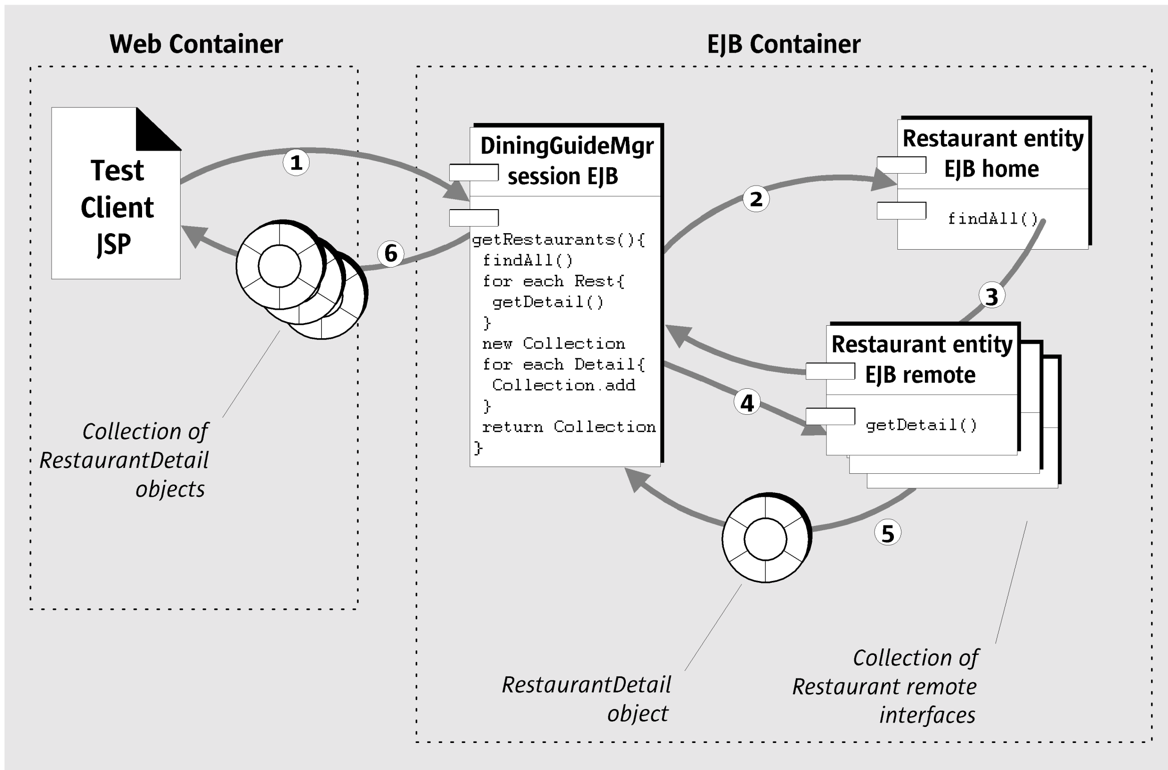


FIGURE 3-1 Function of a Detail Class

The numbered items in FIGURE 3-1 signify the following actions:

1. The web container passes a client's request for all restaurant data to the DiningGuideManager session bean.
2. The session bean calls the Restaurant entity bean's findAll method to perform a lookup on the Restaurant entity bean.
3. The findAll method obtains all available remote references to the entity bean.
4. For each remote reference returned, the session bean calls the Restaurant bean's getRestaurantDetail business method to fetch the RestaurantDetail class.
5. The getRestaurantDetail method returns a RestaurantDetail object, which is added to the collection.
6. The session bean returns a collection of all RestaurantDetail objects to the web container, which formats the data appropriately for the client to view.

Summary of Steps

Creating the EJB tier requires six steps:

1. Creating the entity beans

   First, you create database schemas in the IDE, mapped from the tutorial's database tables. Next, using the EJB Builder, you create CMP entity beans that model the tables with the EJB Builder wizard. Then, you program the entity beans by adding and coding create, finder, and business methods.

2. Creating detail classes to be passed to the client for display

   You create two JavaBeans classes with the same fields as the corresponding entity beans, and include the same accessor methods. These classes are used by the entity beans as types for parameters, fields, and return values. Returning these classes to the client is more efficient regarding network bandwidth than returning remote references for the entity bean instances.

3. Creating business methods on the entity beans to fetch the detail classes

4. Testing the entity beans' methods with the IDE's test application facility

   From each entity bean, you automatically generate a test client that, when viewed in a web browser, allows you to create an instance of the bean and then exercise the business methods on the instance.

5. Creating the session bean

   You use the EJB Builder to create a stateless session bean. You program the bean's create method to perform a lookup on the entity beans and its getter methods to construct collections of detail objects (from the detail classes for each entity bean. You create a method that creates customer review record in the database. You also create two dummy business methods required by the SOAP runtime.

6. Testing the session bean's methods

   Before testing the session bean, you add references to the CMP entity beans to the EJB module's property sheet. From the session bean, you then generate a test application, which includes an EJB module. To this module, you add the EJB modules from the entity beans' test applications. Then you use the test client to create an instance of the session bean and then exercise its methods.

Note – Before you can begin work on the tutorial application, you must first have performed all the setup steps described in Chapter 1.

Creating Entity Beans With the EJB Builder

Create two entity beans, Restaurant and Customerreview, to represent the two database tables you created in Chapter 1.

In version 2.0 of the EJB architecture, entity beans can have local interfaces, remote interfaces, or both. The criterion for deciding which to use rests on whether the client that calls the bean's methods is remote or local to the bean. In this tutorial, you create the entity beans with both remote and local interfaces, for flexibility regarding how the web service will access the beans' methods. Two possibilities are the session bean accesses the beans' methods (using local interfaces), or the web service accesses the methods directly (using remote interfaces).

Tip – For more details about working with the EJB Builder, see the Sun ONE Studio 5 help topics on EJB components.

Note – The source code for the completed entity beans is provided in Appendix A.

Creating the Restaurant and Customerreview Entity Beans

First, create a directory to contain the application, then mount the directory in the IDE. Mounting a directory puts it in the IDE's class path. Next, in the mounted directory, create a database schema from the two database tables. Finally, create a package for the EJB tier and create two entity beans in it that are modeled from the database schema.

Creating the Tutorial's Directory

Create a directory to contain the tutorial's files and mount it in the IDE's file system, as follows:

1. **Somewhere on your file system, create a directory and name it** `DiningGuide`**.**

Note – If you create this as a subdirectory of another directory, the specifications of some methods you create for this tutorial, which include the file specification, may become very long. This may cause execution problems for platforms using Microsoft Windows 2000 with a Service Pack number less than 3. To avoid this, create this directory at the top level of a disk or volume (for example `c:\DiningGuide`).

---

2. **Mount the `DiningGuide` directory.**

   Mounting an object in the Explorer puts that object into the IDE's class path.

   a. **In the Sun ONE Studio 5 IDE, choose the File → Mount Filesystem.**

      The New wizard is displayed.

   b. **Select Local Directory, and click Next.**

      The Select Directory page of the New wizard is displayed.

   c. **Use the Look In file finder to find the `DiningGuide` directory, select it, and click Finish.**

      The new directory (for example, `c:\DiningGuide`) is mounted in the Explorer.

Creating a Database Schema for the Tutorial's Tables

Now create a database schema that models the Restaurant and Customerreview database tables. The IDE reads table definitions from the database and creates the schema.

1. **Start the PointBase server by choosing Tools → PointBase Network Server → Start Server.**

   If the Start Server command is dimmed, the server is already running.

2. **Open a connection to the PointBase sample database.**

   a. **Select the Runtime tab of the Explorer.**

   b. **Expand the Databases node.**

      - If there is a square icon labeled jdbc:pointbase:server://localhost:9092/sample [PBPUBLIC on PBPUBLIC] and it is whole, skip to Step 3.
      - If there is a square icon, as above, and it is broken, right-click it and choose Connect.

        The broken square icon redisplays as a whole square when the connection is opened.

Note – If there is no square icon, your IDE and application server were installed separately from each other and the PointBase JDBC driver has not been enabled in the IDE; consult the *Sun ONE Studio 5, Mobile Edition Getting Started Guide* for instructions on connecting an external PointBase database to the IDE.

---

3. **Begin creating the schema.**

   a. **Right-click the DiningGuide node in the Explorer and choose New → All Templates.**

      The New Wizard is displayed, showing the Choose Template page.

   b. **Expand the Databases node, select Database Schema, and click Next.**

      The New Object Name page of the New wizard is displayed.

4. **Type `dgSchema` in the Name field and click Next.**

   The Database Connection page of the wizard is displayed.

5. **Specify the source database for the schema.**

   a. **Enable the Existing Connection option.**

   b. **Select from the list jdbc:pointbase:server://localhost:9092/sample [PBPUBLIC on PBPUBLIC].**

   c. **Click Next.**

      The Tables and Views page is displayed.

6. **Select the table to be modeled in the schema.**

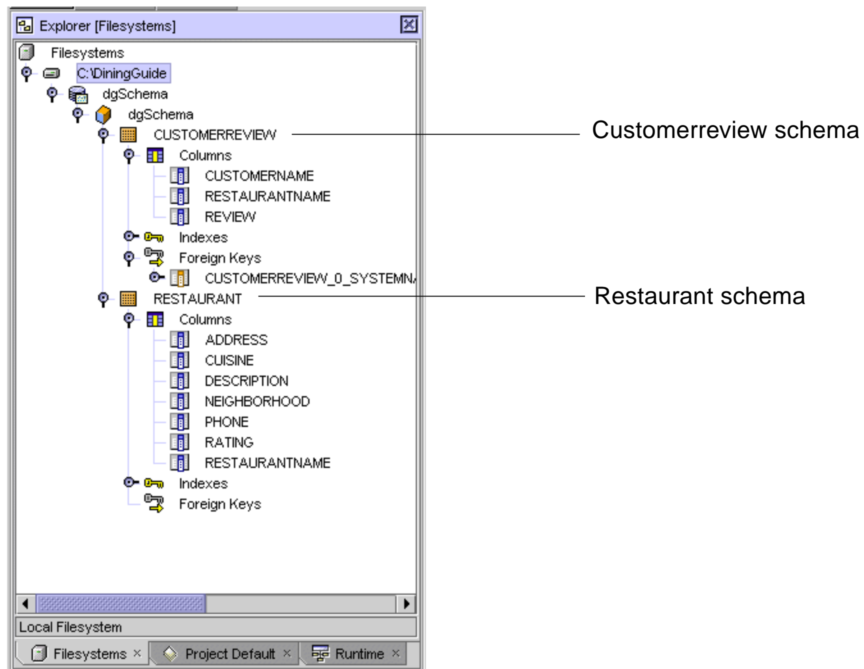
   a. **Select the CUSTOMERREVIEW table in the list of available tables and click the Add button.**

   b. **Select the RESTAURANT table in the list and click the Add button.**

      The CUSTOMERREVIEW and RESTAURANT tables appear in the list of selected tables and views.

   c. **Click Finish.**

      The new database schema appears under the `DiningGuide` directory in the Explorer. If you expand all its subnodes, it looks like this:



Creating a Java Package for the EJB Tier

Create a Java package within the mounted DiningGuide node to hold the EJB tier, which is the data of your application.

1. **Right-click the DiningGuide node and choose New → Java Package.**
2. **Name the new package** `Data` **and click Finish.**

   The new Data package appears under the DiningGuide directory.

Creating the Restaurant Entity Bean

Create an entity bean to model the Restaurant table, as follows:

1. **Begin creating the Restaurant entity bean.**

   a. **Right-click the new Data package and choose New → All Templates.**

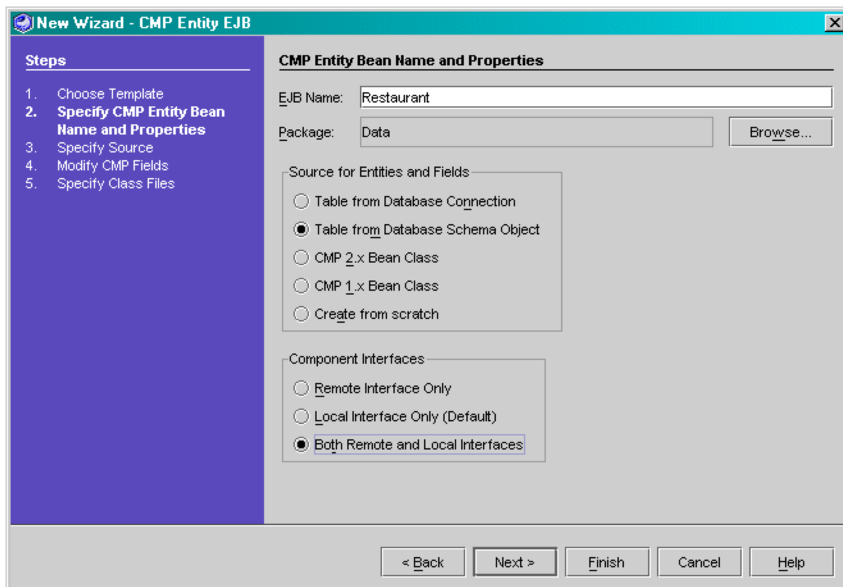
b. From the Templates wizard, expand the J2EE node, select CMP Entity EJB and click Next.

The CMP Entity Bean Name and Properties page of the New wizard (used by the EJB Builder module) is displayed. If you click the Help button on any page of the wizard, you can get context-sensitive help on creating CMP entity beans.

c. Name the new CMP entity bean `Restaurant` and select the following options:

Option Category	Option to Select
Source for Entities and Fields	Table from Database Schema Object
Component Interfaces	Both Remote and Local Interfaces

The New wizard looks like this.



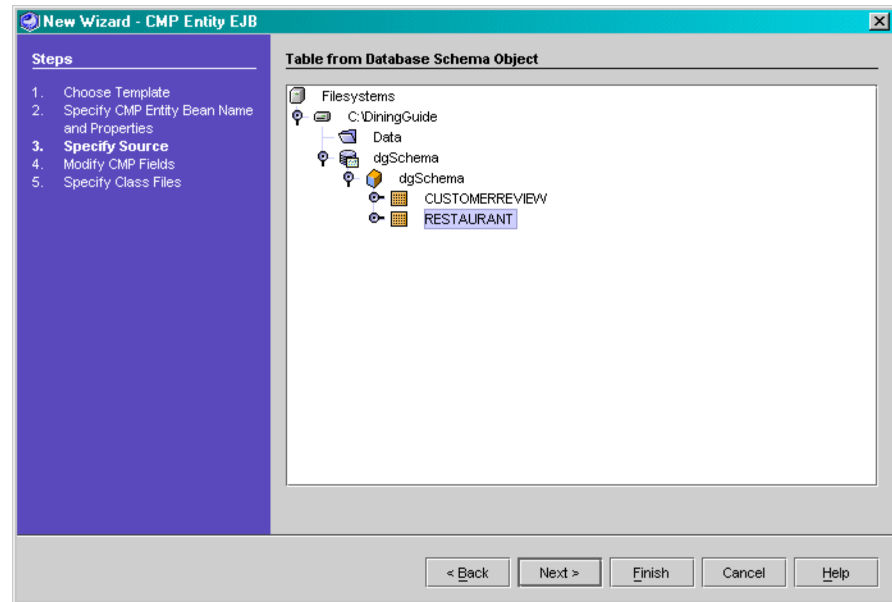
d. Click Next.

This displays the Table from Database Schema Object page.

2. Specify the RESTAURANT schema.

a. Expand the DiningGuide node and the nodes under the dgSchema node, and select the RESTAURANT table.

The page looks like this:



b. **Click Next to go to the CMP Fields page.**

3. **Customize any fields you wish to define the CMP bean.**

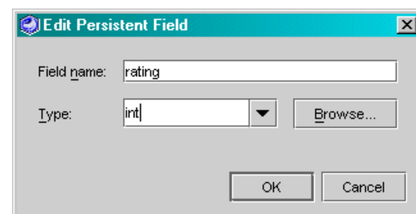
   You see a side-by-side display of the columns of the Restaurant database table and the corresponding Java fields that the columns will be mapped to when the wizard creates the Restaurant entity bean. For this tutorial, you want to change the type of the rating field.

   a. **Select the rating field and click the Edit button.**

      The Edit Persistent Field dialog box is displayed.

   b. **Delete the existing text in the Type field and type `int`.**

      The dialog box looks like this:



   c. **Click OK to close the dialog box.**

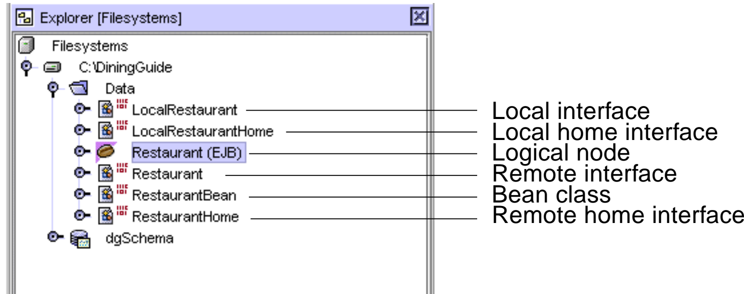
      The type of the rating Java field is changed to `int`.

d. Click Next.

The CMP Entity Bean Class Files page is displayed, listing the parts of the `Restaurant` bean that will be created. Notice that the EJB Builder wizard has automatically named the new entity bean with the same name as the database table.

4. **Finish creating the bean by clicking Finish.**

The new Restaurant entity bean and all its parts are created and displayed in the Explorer window.



Five of the parts are interfaces and one part is the bean class. The sixth part is the *logical node* that groups all the elements of the enterprise bean together and facilitates working with them.

Creating the Customerreview Entity Bean

Create the Customerreview entity bean as you did the Restaurant bean, using the following steps:

1. **Begin creating the Customerreview entity bean.**

a. Right-click the new Data package and choose New → CMP Entity EJB.

Notice that this item appears in the contextual menu. This shortcut is created for your convenience whenever you select an item from the New menu.

b. Name the new CMP bean Customerreview and select the following options:

Option Category	Option to Select
Source for Entities and Fields	Table from Database Schema Object
Component Interfaces	Both Remote and Local Interfaces

c. Click Next.

This displays the Table from Database Schema Object page.

2. Specify the CUSTOMERREVIEW schema.

a. Expand the DiningGuide node and all the nodes under the dgSchema node.

b. Select the CUSTOMERREVIEW table.

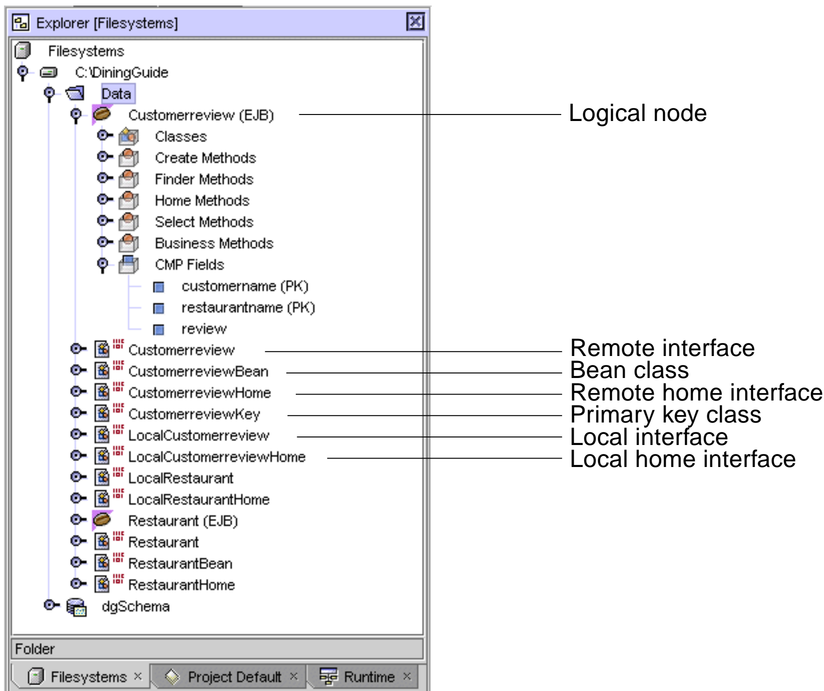
c. Click Next.

3. Finish creating the bean.

a. Click Next on the CMP Fields page.

b. Click Finish on the last page (CMP Entity Bean Class Files page).

The `Customerreview` entity bean is displayed in the Data package in the Explorer. Notice that there is also a primary key class named CustomerreviewKey. This class is automatically created when the entity bean has a composite primary key. (See TABLE 1-2 in Chapter 1 to confirm the composite primary key in this table.)

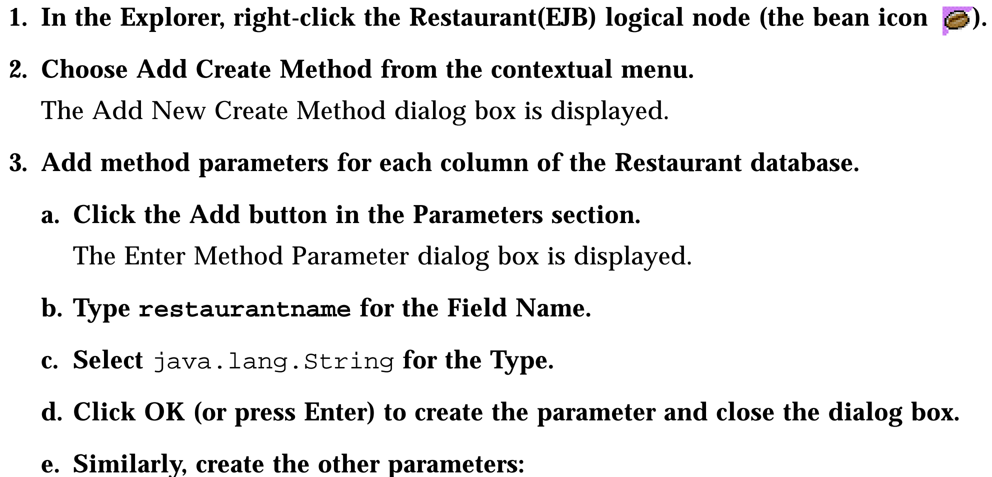


Creating Create Methods for CMP Entity Beans

Create the create methods for both entity beans, adding parameters and code to initialize the fields of the beans' instances.

Creating the Restaurant Bean's Create Method

Create the create method for the Restaurant entity bean as follows:

1. **In the Explorer, right-click the Restaurant(EJB) logical node (the bean icon ).**
2. **Choose Add Create Method from the contextual menu.**

   The Add New Create Method dialog box is displayed.
3. **Add method parameters for each column of the Restaurant database.**

   a. **Click the Add button in the Parameters section.**

   The Enter Method Parameter dialog box is displayed.

   b. **Type `restaurantname` for the Field Name.**

   c. **Select `java.lang.String` for the Type.**

   d. **Click OK (or press Enter) to create the parameter and close the dialog box.**

   e. **Similarly, create the other parameters:**

Field Name	Type
cuisine	java.lang.String
neighborhood	java.lang.String
address	java.lang.String
phone	java.lang.String
description	java.lang.String
rating	int

Note – The order in which you create these parameters becomes important when you test the bean with the test application facility. Create them in the order given here.

Keep the two exceptions created by default, and make sure the method is added to both Home and Local Home interfaces.

4. **Click OK.**

   The IDE propagates a create method under the RestaurantHome interface, another create method under the LocalRestaurantHome interface, and an ejbCreate method under the Restaurant bean class (RestaurantBean). A related ejbPostCreate method is also added to the bean class.

5. **Expand the Restaurant(EJB) logical node and the Create Methods folder, and double-click the create method.**

   The Source Editor is displayed with the cursor at the ejbCreate method of the bean.

Note – If you right-click the create method node and choose Help, you can get online help information on create methods.

---

6. **Add the following code (the bold text only) to the body of the ejbCreate method to initialize the fields of the bean instance:**

```
public String ejbCreate(java.lang.String restaurantname,
java.lang.String cuisine, java.lang.String neighborhood,
java.lang.String address, java.lang.String phone,
java.lang.String description, int rating) throws
javax.ejb.CreateException {
       if (restaurantname == null) {
// Join the following two lines in the Source Editor
              throw new javax.ejb.CreateException("The restaurant
name is required.");
       }
       setRestaurantname(restaurantname);
       setCuisine(cuisine);
       setNeighborhood(neighborhood);
       setAddress(address);
       setPhone(phone);
       setDescription(description);
       setRating(rating);

       return null;
}
```

Tip – After you enter code (either by typing or copying and pasting) into the Source Editor, press Control-Shift F to reformat it properly. If you are copying from the PDF file, join the lines that get broken by word wrap, as indicated by the code comment.

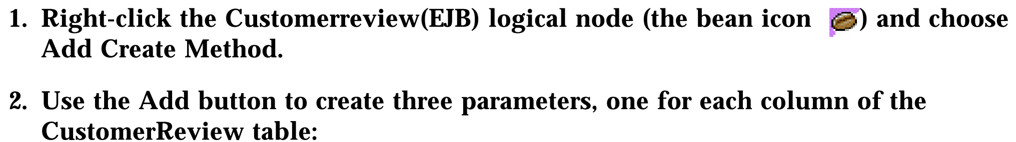
When the Restaurant entity bean's create method is called, it creates a new record in the database, based on the container-managed fields of this bean.

7. **Select the Restaurant(EJB) logical node and press F9 to compile the bean.**

   The Restaurant entity bean should compile without errors.

Creating the Customerreview Bean's Create Method

Create the create method for the Customerreview entity bean as follows:

1. **Right-click the Customerreview(EJB) logical node (the bean icon ) and choose Add Create Method.**

2. **Use the Add button to create three parameters, one for each column of the CustomerReview table:**

Field Name	Type
restaurantname	java.lang.String
customername	java.lang.String
review	java.lang.String

Note – As in Step a, create these parameters in the order given.

   Keep the two exceptions created by default, and make sure the method is added to both Home and Local Home interfaces.

3. **Click OK.**

4. **Open the Customerreview(EJB) logical node and the Create Methods folder, and double-click the create method.**

   The Source Editor opens with the cursor at the ejbCreate method of the bean.

5. **Add the following (bold) code to the body of the ejbCreate method to initialize the fields of the bean instance:**

```
public CustomerreviewKey ejbCreate(java.lang.String
restaurantname, java.lang.String customername, java.lang.String
review) throws javax.ejb.CreateException {
       if ((restaurantname == null) || (customername == null)) {
// Join the following two lines in the Source Editor
              throw new javax.ejb.CreateException("Both the
restaurant name and customer name are required.");
       }
       setRestaurantname(restaurantname);
       setCustomername(customername);
       setReview(review);

       return null;
}
```

Tip – After you enter code (either by typing or copying and pasting) into the Source Editor, press Control-Shift F to reformat it properly. If you are copying from the PDF file, join the lines that get broken by word wrap, as indicated by the code comment.

When the ejbCreate method is called, it creates a new record in the database, based on the container-managed fields of this bean.

6. **Select the Customerreview(EJB) logical node and press F9 to compile the bean.**

   The Customerreview entity bean should compile without errors.

   Now, create finder methods on both entity beans that will locate all or selected instances of each bean in the context.

Creating Finder Methods on Entity Beans

Create a findAll method on the Restaurant bean to locate all restaurant data. Also create a findByRestaurantName on the Customerreview bean to locate review data for a given restaurant.

Every finder method, except findByPrimaryKey, must be associated with a query element in the deployment descriptor. When you create the finder methods for these two entity beans, specify SQL statements using a database-independent language specified in the EJB 2.0 specification, namely EJB QL. At deployment time, the application server plugin translates the EJB QL into the SQL of the target database.

Creating the Restaurant Bean's findAll Method

To create the `Restaurant` bean's findAll method:

1. **Right-click the Restaurant(EJB) logical node and choose Add Finder Method.**

   The Add New Finder Method dialog box is displayed.

2. **Type `All` after the "find" string in the Name field.**
3. **Select** `java.util.Collection` **for the Return type.**
4. **Accept the two default exceptions.**
5. **Define the EJB QL statements, as follows:**

EJB QL Statement	Text
Select	`Object(o)`
From	`Restaurant o`

6. **Make sure the method is added to both Home and Local Home interfaces.**
7. **Click OK.**

   The new findAll method is created in the Local and Local `Home` interfaces of the `Restaurant` bean.

Note – If you right-click the Finder Methods node and choose Help, you can get online help information on finder methods.

---

8. **Select the Restaurant(EJB) logical node and press F9 to compile the bean.**

   The Restaurant entity bean should compile without errors.

Creating the Customerreview Bean's findByRestaurantName Method

To create the Customerreview bean's findByRestaurantName method:

1. **Right-click the Customerreview(EJB) logical node and choose Add Finder Method.**

   The Add New Finder Method dialog box is displayed.

2. **Type `ByRestaurantName` after the "find" string in the Name field.**
3. **Select** `java.util.Collection` **for the Return type.**

4. **Click the parameter's Add button.**

   The Enter New Parameter dialog box is displayed.

5. **Type `restaurantname` for the parameter name.**

6. **Select `java.lang.String` for the parameter type.**

7. **Click OK.**

8. **Accept the two default exceptions.**

9. **Define the EJB QL statements, as follows:**

EJB QL Statement	Text
Select	`Object(o)`
From	`Customerreview o`
Where	`o.restaurantname = ?1`

   (Which numeral you use for the Where clause depends on the position of the parameter in the finder method. In this case there's only one parameter, so the numeral is "1").

10. **Make sure the method is added to both Home and Local Home interfaces.**

11. **Click OK.**

    The new findByRestaurantName method is created in the Local and Local Home interfaces of the Customerreview bean.

12. **Select the Customerreview(EJB) logical node and press F9 to compile the bean.**

    The Customerreview entity bean should compile without errors.

Creating Business Methods for Testing Purposes

Create a business method for each entity bean that returns a value of one of its parameters. The business method enables you to test the beans later. For Restaurant, create a getRating method; for Customerreview, create a getReview method.

Creating the Restaurant Bean's getRating Method

To create the getRating business method for the Restaurant bean:

1. **Examine the Restaurant entity bean's business methods.**

   a. **Expand the Restaurant(EJB) logical node, and then expand its Business Methods node.**

   There are no business methods yet for this entity bean.

   b. **Expand the Restaurant bean's class (RestaurantBean), and then expand its Methods node.**

   Every field on the bean has accessor methods, including a getRating method.

   These methods are used by the container for synchronization with the data source. To use any of these methods in development, you have to create them as business methods.

2. **Create the getRating business method.**

   a. **Right-click the Restaurant(EJB) logical node and choose Add Business Method.**

   The Add New Business Method dialog box is displayed.

   b. **Type `getRating` in the Name field.**

   c. **Select `int` from the list of the Return Type field.**

   d. **Accept the default exception (RemoteException), and the designation that the method will be created in both Remote and Local Home interfaces.**

   e. **Click OK.**

3. **Examine the Restaurant entity bean's business methods again.**

   Under the logical node's Business Methods folder, the getRating method is now accessible as a business method. When the getRating method is used, it returns the value in the rating column of a selected restaurant record.

4. **Validate the bean by right-clicking the Restaurant(EJB) logical node and choosing Validate EJB from the contextual menu.**

   The Restaurant entity bean should compile without errors. Now, create a similar method for the Customerreview bean.

Creating the Customerreview Bean's getReview Method

To create the getReview business method for the Customerreview bean:

1. **Right-click the Customerreview(EJB) logical node and choose Add Business Method.**

   The Add New Business Method dialog box is displayed.

2. **Type `getReview` in the Name field.**

3. **Select `java.lang.String` in the Return Type field.**

   Accept the default exception (RemoteException), and the designation that the method will be created in both Remote and Local Home interfaces.

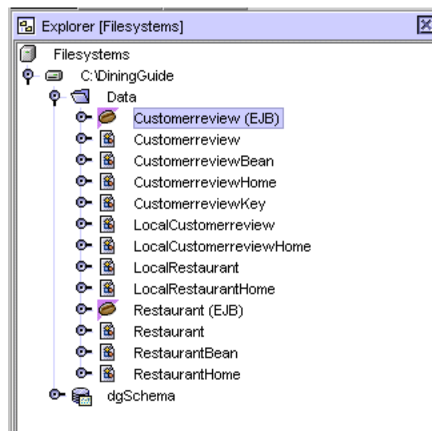
4. **Click OK.**

   The getReview method is now accessible as a business method. When the getReview method is called, it returns the value in the review column of a selected restaurant record.

5. **Right-click the Customerreview(EJB) logical node and choose Validate EJB from the contextual menu.**

   The Customerreview entity bean should compile without errors.

6. **Check that the icons in the Explorer no longer indicate that the beans are uncompiled.**



Creating Detail Classes to View Entity Bean Data

As discussed in "The Detail Classes" on page 49, this tutorial uses detail classes as a mechanism for holding row data for viewing and reducing method calls to the entity beans. These classes must have the same fields as the corresponding entity beans, access methods for each field, and a constructor that sets each field.

Note – The source code for the completed detail classes is provided in Appendix A.

Creating the Detail Classes

First, create a RestaurantDetail class and a CustomerreviewDetail class:

1. **In the Explorer, right-click the Data package and choose New → All Templates.**

   The New wizard is displayed.

2. **In the Choose Template page of the wizard, expand the Java Beans node and select Java Bean.**

3. **Click Next.**

   The New Object Name page of the wizard is displayed.

4. **Type `RestaurantDetail` in the Name field and click Finish.**

   The new bean is displayed in the Explorer.

5. **Right-click the Data package node and choose New → Java Bean.**

   Notice the IDE has created a shortcut to the Java Bean template.

6. **Type `CustomerreviewDetail` in the Name field and click Finish.**

Creating the Detail Class Properties and Their Accessor Methods

Now, duplicate the entity beans' CMP fields in the detail classes. You do this by adding these fields as bean properties. (If you look in the Bean Patterns nodes of an entity bean's bean class, you will see that the CMP fields are stored as bean properties.) While adding the fields, you can automatically create accessor methods for each field.

To create the detail class properties and methods:

1. **Expand the RestaurantDetail node and the class RestaurantDetail node.**
2. **Right-click the Bean Patterns node and choose Add → Property.**

   The New Property Pattern dialog box is displayed.
3. **Type `restaurantname` in the Name field.**
4. **Select** `String` **for the Type.**
5. **Select the** `Generate Field` **option.**
6. **Select the** `Generate Return Statement` **option.**
7. **Select the** `Generate Set Statement` **option.**
8. **Click OK.**
9. **Repeat Step 2 through Step 8 to create the following additional properties:**

   ```
   cuisine (String)
   neighborhood (String)
   address (String)
   phone (String)
   description (String)
   rating (int)
   ```
10. **Expand the RestaurantDetail bean's Methods node.**

    Observe that accessor methods have been generated for each field.
11. **Expand the CustomerreviewDetail node and the class CustomerreviewDetail node.**
12. **Repeat Step 2 through Step 8 to create the following properties in the Bean Properties node:**

    ```
    restaurantname (String)
    customername (String)
    review (String)
    ```

Creating the Detail Class Constructors

To create constructors for the detail classes that instantiate the class fields:

1. **Expand the RestaurantDetail bean, right-click the class RestaurantDetail node, and choose Add → Constructor.**

   The Edit New Constructor dialog box is displayed.

2. **Add the following method parameters and click OK:**

```
java.lang.String restaurantname
java.lang.String cuisine
java.lang.String neighborhood
java.lang.String address
java.lang.String phone
java.lang.String description
int rating
```

3. **Add the following bold code to the body of this RestaurantDetail constructor to initialize the fields:**

```
public RestaurantDetail(java.lang.String restaurantname,
java.lang.String cuisine, java.lang.String neighborhood,
java.lang.String address, java.lang.String phone,
java.lang.String description, java.lang.Integer rating){
        System.out.println("Creating new RestaurantDetail");
        setRestaurantname(restaurantname);
        setCuisine(cuisine);
        setNeighborhood(neighborhood);
        setAddress(address);
        setPhone(phone);
        setDescription(description);
        setRating(rating);
}
```

Tip – Remember, you can reformat code you paste or type into the Source Editor by pressing Control-Shift F.

---

4. **Similarly, add a constructor to the CustomerreviewDetail class with the following parameters:**

```
java.lang.String restaurantname
java.lang.String customername
java.lang.String review
```

5. **Add the following bold code to the body of this CustomerreviewDetail constructor to initialize the fields:**

```
public CustomerreviewDetail(java.lang.String restaurantname,
java.lang.String customername, java.lang.String review){
       System.out.println("Creating new CustomerreviewDetail");
       setRestaurantname(restaurantname);
       setCustomername(customername);
       setReview(review);
}
```

6. **Right-click the Data package and choose Compile All.**

   The package should compile without errors.

   Now, create `get` methods on the entity beans to retrieve instances of the detail classes.

Creating Business Methods on the Entity Beans to Fetch the Detail Classes

Create a method on each entity bean that returns an instance of its corresponding detail class.

To create these methods:

1. **In the Explorer, right-click the Restaurant (EJB) logical node and choose Add Business Method.**

   The Add New Business Method dialog box is displayed.

2. **Type `getRestaurantDetail` in the Name field.**

3. **For the return type, use the Browse button to select the** `RestaurantDetail` **class.**

Tip – Be sure to select the class ( ), not the bean's node.

Data.RestaurantDetail is displayed in the Return Type field.

4. **Make sure the option Both Remote and Local Interfaces is enabled.**

5. **Accept all other default values and click OK to create the method.**

6. **Find the new method under the logical bean's Business Methods node and double-click it to display it in the Source Editor.**

7. **Add the following bold code to the method's body:**

```
public Data.RestaurantDetail getRestaurantDetail() {
       return (new RestaurantDetail(getRestaurantname(),
getCuisine(), getNeighborhood(), getAddress(), getPhone(),
getDescription(), getRating()));
}
```

8. **Select the Restaurant (EJB) logical node and press F9 to compile the code.**
9. **In the Explorer, right-click the Customerreview (EJB) logical node and choose Add Business Method.**
10. **Type `getCustomerreviewDetail` in the Name field.**
11. **For the return type, use the Browse button to select the CustomerreviewDetail class icon.**
12. **Make sure the option Both Remote and Local Interfaces is enabled.**
13. **Accept all other default values and click OK to create the method.**
14. **Open the method in the Source Editor and add the following bold code:**

```
public Data.CustomerreviewDetail getCustomerreviewDetail() {
       return (new CustomerreviewDetail(getRestaurantname(),
getCustomername(), getReview()));
}
```

15. **Right-click the Data package and choose Compile All.**

    The entire package should compile without errors.

    You have finished creating the entity beans of the tutorial application and their detail class helpers. Your next task is to test the beans.

Testing the Entity Beans

With the Sun ONE Studio 5 IDE, you can create a test client to exercise the methods on an entity bean. The test client uses JavaServer Pages technology that allows you to create instances of the bean and exercise the bean's create, finder, and business methods in a web browser.

Use this test client to exercise the Restaurant bean's create and getRating methods.

Creating a Test Client for the Restaurant Bean

When you create a test client, the IDE generates an EJB module, a J2EE application module, and many supporting elements.

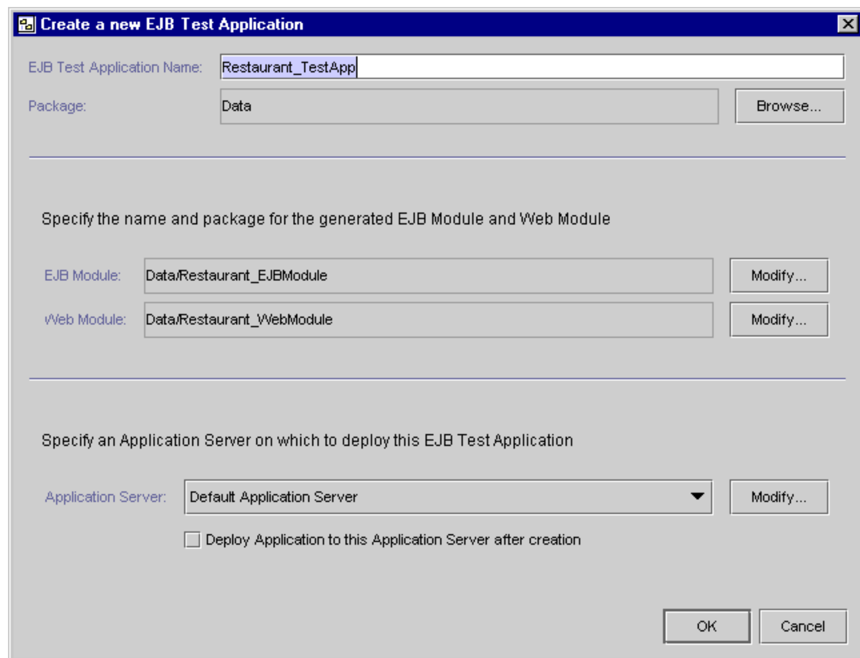
To create a test client for the Restaurant entity bean:

1. **Right-click the Restaurant(EJB) logical node and choose Create New EJB Test Application.**

   The EJB Test Application wizard is displayed.

2. **Accept all default values.**

   The wizard's window looks like this:

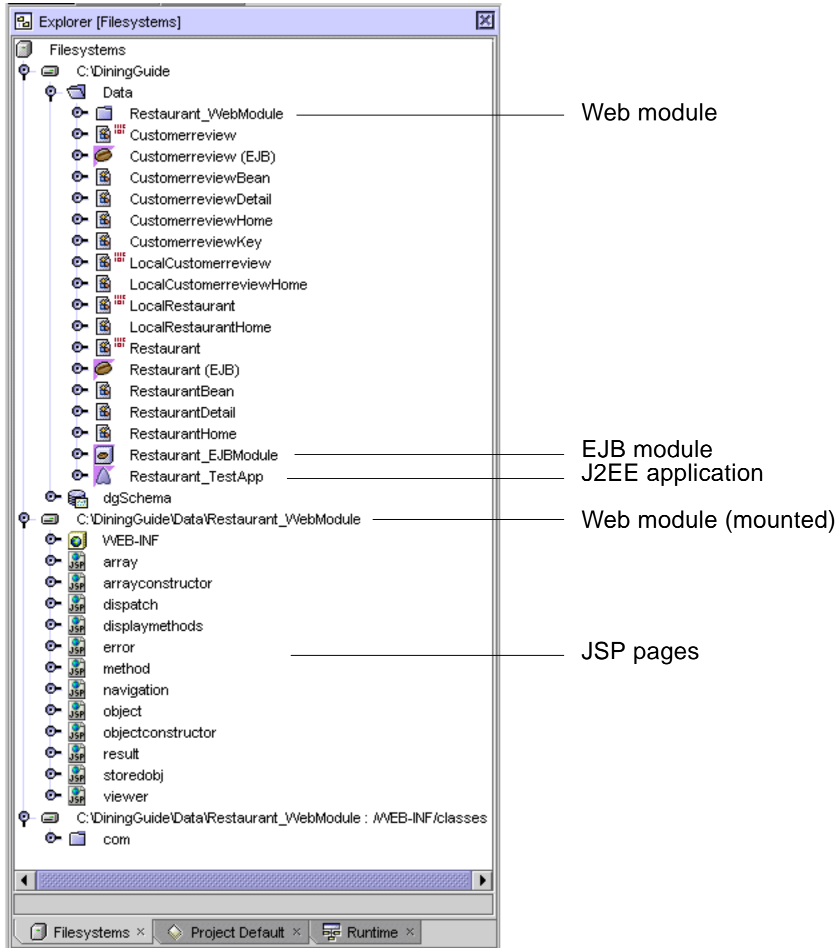


3. **Click OK.**

   A progress monitor appears briefly and then goes away when the process is complete. Another window is displayed informing you that the web module that was created is also visible in the Project tab. This window should go away automatically, also. If not, click OK to close the window.

4. **View the resulting test objects in the Explorer.**

   The IDE has created an EJB module named Restaurant_EJBModule, a web module named Restaurant_WebModule (which is also mounted separately), and a J2EE application named Restaurant_TestApp. The web module contains a number of JSP pages that support the test client. The J2EE application includes references to the EJB module and to the web module.



   The J2EE application created by the IDE contains references to the web module and the EJB module. You can see these objects by expanding the Restaurant_TestApp:

Providing the Sun ONE Application Server 7 Plugin With Database Information

In order for the test client to find the database and log onto it, you must add information about your database to the application server properties of the EJB module.

To add the required information:

1. **Expand the EJB module (Restaurant_EJBModule) in the Explorer and select the Restaurant node (a reference to the Restaurant bean) under it, to display its properties.**

   If the Properties Window is not already displayed, choose View → Properties.

2. **Select the Sun ONE AS tab of the Properties window.**

3. **Confirm that the following properties are set as follows:**

Property	Value
Mapped Fields	7 container managed fields mapped
Mapped Primary Table	RESTAURANT
Mapped Schema	dgSchema

   If these values are displayed, continue with Step 5.

4. **If the values are not displayed, remap the Restaurant reference as follows:**

   a. **Select the value field of the Mapped Fields value and click on the ellipsis button.**

      The Map to Database wizard is displayed.

   b. **Click Next to view the Select Tables page.**

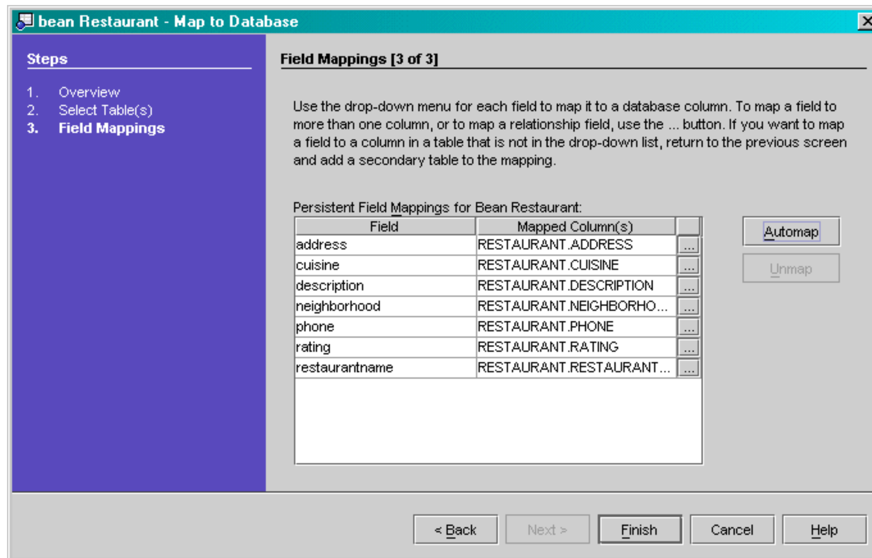
   c. **Select RESTAURANT from the drop list of the Primary Table field.**

      If RESTAURANT is not in the list, use the Browse button to find the table within the dgSchema schema.

   d. **Click Next to view the Field Mappings page.**

   e. **If the fields are unmapped, click the Automap button.**

Values for mappings appear for each field, as shown:



f. **Click Finish.**

The values should now display as in Step 3.

5. **Select the EJB module (Restaurant_EJBModule) to display its properties.**

6. **Select the Sun ONE AS tab of the properties window.**

7. **Click in the value field for the CMP Resource property to display an ellipsis button.**

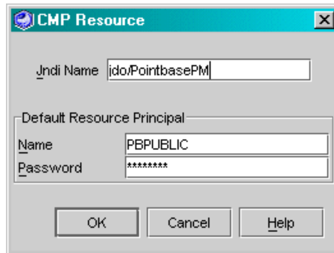
8. **Click the ellipsis button to display the CMP Resource property editor.**

9. **Type `jdo/PointbasePM` in the Jndi Name field.**

   This is the JNDI name of the JDBC Persistence Manager described in "Setting JDBC Resources (Microsoft Windows Superusers)" on page 30 or "Setting JDBC Resources (All Other Users)" on page 31.

10. **For the Name and Password fields, type the User Name and Password for your database.**

For the PointBase sample database, these are both PBPUBLIC. The editor looks similar to this:



11. **Click OK to accept the values and close the property editor.**

    You have finished configuring the test application to use your database and now you can deploy the test application.

Deploying and Executing the Restaurant Bean's Test Application

---

Note – Make sure the PointBase server is running before you deploy the test application, or any other J2EE application that accesses the database. In addition, make sure Sun ONE Application Server 7 is running and is the default application server of the IDE. See "Confirming Sun ONE Application Server 7 as the Default Server" on page 29 for information.

To deploy the Restaurant test application:

- **Right-click the Restaurant_TestApp J2EE application node and choose Execute from the contextual menu.**

  A Progress Monitor window shows the progress of the deployment process. The server instance's log file tab on the output window displays progress messages. The application is successfully deployed when you see success messages.

  The IDE starts the default web browser and displays the test application's home URL:

  `http://`*server-host*`:`*server-port-number*`/Restaurant_TestApp/dispatch.jsp`

Your browser displays the test client like this:

List of instances being tested, beginning with the home interface of the bean being tested.

Area where you can enter parameters and invoke methods.

List of objects created during the test session.

Area where results of last method invocation are shown.

Note – If you do not see the browser, look behind the windows of the IDE. You can verify whether the application is deployed by checking the server instance's log file tab on the output window. The application is successfully deployed when you see success messages.

Using the Test Client to Test the Restaurant Bean

On the test client's web page that is displayed, use the create method of the Restaurant bean's home interface to create an instance of the bean. Then test a business method (in this case, getRating) on that instance.

To test the `Restaurant` bean:

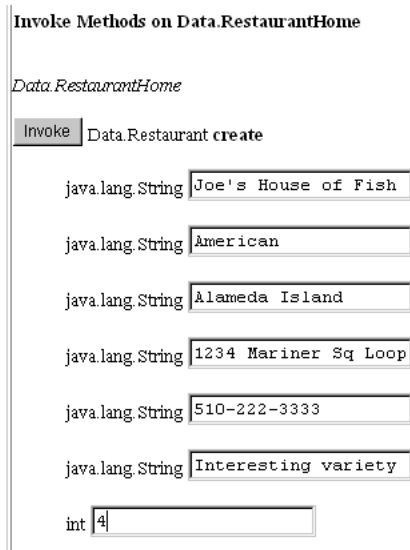
1. **Create an instance of the Restaurant bean by invoking the create method.**

   The create method is under the heading "Invoke Methods on Data.RestaurantHome." There are seven fields under it. The fields are not named, but you can deduce what they are by their order, which is the same order you created them in (see Step a under "Creating Create Methods for CMP Entity Beans" on page 60).

Note – Double-click the Restaurant.create method in the Explorer to display it in the Source Editor; the order of the fields is shown in the method's definition.

Tip – If you want the parameters to appear in a different order, right-click the Restaurant.create method node in the Explorer window and choose Customize. In the Customizer dialog box, rearrange the parameters by selecting and clicking the Up and Down buttons. Then redeploy the test application by right-clicking its node in the Explorer and choosing Deploy.

Type any data you like into the fields, for example (your field order may be different):



2. **Click the Invoke button next to the create method.**

   The deployed test application adds the records you created to the test database. The new Restaurant instance is listed by its restaurantname value in the upper left, and new data objects are listed in the upper right, as shown.

EJB Navigation

This view allows easy access to instances of the home and remote interfaces of the Enterprise Java Bean undergoing testing. The home interface will be listed at the beginning of the list, instances of the remote interface will follow. Selecting an object here allows methods to be invoked on the object instance. An instance of the tested Bean can be created by selecting the home interface and invoking the appropriate method.

- Data.RestaurantHome
- Joe's House of Fish

Stored Objects

This view provides a list of objects created during your test session which may be used as method parameters. Objects are placed on this list once created from a method invocation or when created directly using the new button provided for complex parameters. Selecting an object from this view allows methods on the object instance to be invoked.

Remove Selected | Remove All

- [ ] Data.Restaurant Joe's House of Fish
- [ ] java.lang.Integer 4
- [ ] java.lang.String Interesting variety
- [ ] java.lang.String 510-222-3333
- [ ] java.lang.String 1234 Mariner Sq Loop
- [ ] java.lang.String Alameda Island
- [ ] java.lang.String American
- [ ] java.lang.String Joe's House of Fish

Data.RestaurantHome

The results are shown in the Results area.

Results of the Last Method Invocation

Joe's House of Fish

Method Invoked: *create (java.lang.String,java.lang.String,java.lang.String,java.lang.String,java.lang.String,java.lang.String,int)*
Parameters:
Joe's House of Fish
American
Alameda Island
1234 Mariner Sq Loop
510-222-3333
Interesting variety
4

3. **Test the findAll method of the Restaurant bean by clicking the Invoke button next to it.**

   The results area should look like this:

Results of the Last Method Invocation

size = 3

Method Invoked: *findAll ( )*
Parameters:
none

Notice that three items were returned. This demonstrates that the new database record you created in Step 2 was added to the two you created in Chapter 1.

4. **Test the findByPrimaryKey method by typing in `Bay Fox` and clicking the Invoke button next to the method.**

   The results area shows that the Bay Fox record was returned.

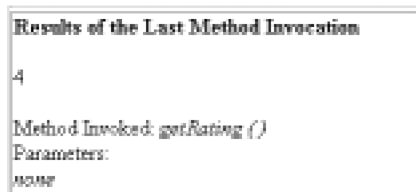
   Now, test the entity bean's business methods.

5. Select the instance for Joe's House of Fish listed under Data.RestaurantHome in the instances list (upper left).

   The getRating method is now listed under the Invoke Methods area.

6. **Click the Invoke button next to the getRating method.**

   The results of this action are listed in the Results area and should look like this:



   This demonstrates that you have created a new record in the database and used the getRating method to retrieve the value of one of its fields.

   Continue testing by selecting created objects and invoking their methods. For example, if you select one of the Data.RestaurantDetail objects, you can invoke its getter methods to view its data, or its setter methods to write new data to the database.

7. **When you are finished testing, you can quit the browser, point it to a different URL, or do nothing, as you wish**

Checking the Additions to the Database

To verify that the Restaurant_TestApp application inserted data in the database:

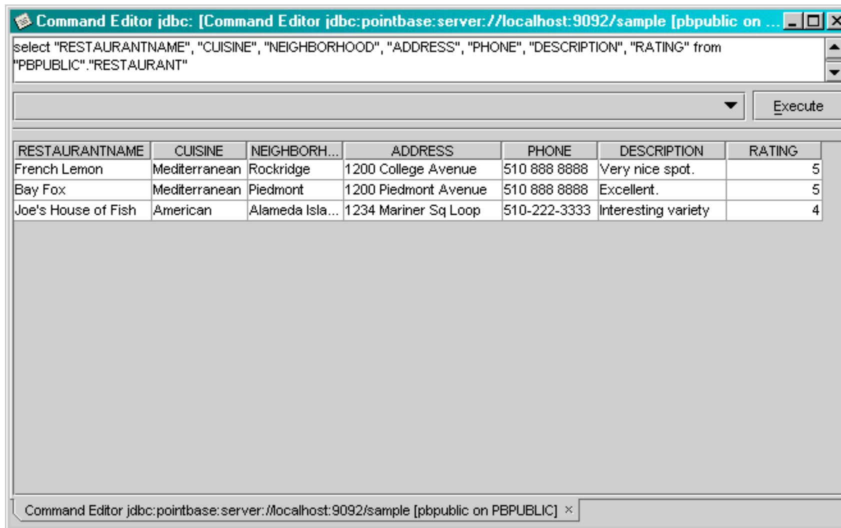
1. **In the IDE, select the Runtime tab of the Explorer.**

2. **Expand the Databases node, the PointBase connection node, and the Tables node.**

   The nodes for each table in the sample database are displayed.

3. **Right-click the RESTAURANT table and choose View Data.**

   A command editor window is displayed, showing the new Restaurant record you entered in Step 1 under "Using the Test Client to Test the Restaurant Bean" on page 78, as shown.



You are now ready to create the session bean.

Note – You do not need to stop the application server's process (which is listed in the execution window). Whenever you redeploy, the server undeploys the application and then redeploys. When you exit the IDE, a dialog box is displayed for terminating the application server's instance process. However, you can manually terminate it at any time by right-clicking the server1(*hostname*:*host-port*) node in the execution window and choosing Terminate Process.

Creating a Test Client for the Customerreview Bean

Create a test client for the Customerreview entity bean by repeating all the steps in "Creating a Test Client for the Restaurant Bean" on page 73, but using values appropriate to the Customerreview bean.

To summarize creating the test client application:

1. **Select the Customerreview(EJB) logical bean and choose Create a new EJB Test Application.**
2. **Accept all default values and click OK.**

   Add database information for the plugin, similar to the description in "Providing the Sun ONE Application Server 7 Plugin With Database Information" on page 75, using values appropriate to the Customerreview bean.

To summarize adding database information to the plugin:

1. **Expand the EJB module (Customerreview_EJBModule), select the Customerreview node under it, and display its properties.**
2. **Select the Sun ONE AS tab of the Properties window.**
3. **Confirm that the following three values for the appropriate properties:**

Property	Value
Mapped Fields	3 container managed fields mapped
Mapped Primary Table	CUSTOMERREVIEW
Mapped Schema	dgSchema

If these values are displayed, continue with Step 5.

4. **If the values are not displayed, remap the Customerreview bean as follows:**
   a. **Set the Mapped Schema to** `Data/dgSchema`**.**
   b. **Set the Mapped Primary Table to** `CUSTOMERREVIEW`**.**
   c. **Click the ellipsis button in the value field of the Mapped Fields property.**
   d. **In the wizard, click Next to view the Select Tables page.**
   e. **Select** `CUSTOMERREVIEW` **from the drop list of the Primary Table field.**
   f. **Click Next to view the Field Mappings page.**
   g. **If the fields are unmapped, click the Automap button.**
   h. **Click Finish.**
5. **Display the EJB module (Customerreview_EJBModule) properties.**
6. **Select the Sun ONE AS tab of the properties window.**
7. **Click the ellipsis button in the value field of the CMP Resource property.**
8. **In the CMP Resource property editor, type `jdo/PointbasePM` in the Jndi Name field.**
9. **For the Name and Password fields, type the User Name and Password for your database.**

   For the PointBase sample database, these are both PBPUBLIC.
10. **Click OK to accept the values and close the property editor.**

Deploying and Executing the Customerreview Bean's Test Application

---

Note – Make sure the PointBase server is running before you deploy the test application, or any other J2EE application that accesses the database. In addition, make sure Sun ONE Application Server 7 is running and is the default application server of the IDE. See "Confirming Sun ONE Application Server 7 as the Default Server" on page 29 for information.

To deploy the Customerreview test application:

- **Right-click the Customerreview_TestApp J2EE application node and choose Execute from the contextual menu.**

  A Progress Monitor window shows the progress of the deployment process. The server instance's log file tab on the output window displays progress messages. The application is successfully deployed when you see success messages.

  The IDE starts the default web browser and displays the test application's home URL:

  `http://`***server-host***`:`***server-port-number***`/Customerreview_TestApp/dispatch.jsp`

Note – If you do not see the browser, look behind the windows of the IDE. You can verify whether the application is deployed by checking the server instance's log file tab on the output window. The application is successfully deployed when you see success messages.

Testing the Customerreview Entity Bean

On the test client's web page, use the create method of the Customerreview bean's home interface to create an instance of the bean. Then test a business method (in this case, getCustomerreview) on that instance.

To test the Customerreview bean:

1. **Create an instance of the Customerreview bean by invoking the create method.**

   The create method is under the heading "Invoke Methods on Data.CustomerreviewHome." There are three fields under it.

2. **Type values in the three fields.**

   Type any data you like into the fields, for example (your field order may be different):

   Invoke Methods on Data.CustomerreviewHome

   Data.CustomerreviewHome
   Invoke Data.Customerreview **create**
   java.lang.String Bay Fox
   java.lang.String Bill Goodperson
   java.lang.String Excellent wine list.

3. **Click the Invoke button next to the create method.**

   The deployed test application adds the records you created to the test database. The new Restaurant instance is listed by its restaurantname value in the upper left, and new data objects are listed in the upper right, as shown.

   Results of the Last Method Invocation

   mytest.CustomerreviewKey@834dbf82

   Method Invoked: *create (java.lang.String,java.lang.String,java.lang.String)*
   Parameters:
   Bay Fox
   Bill Goodperson
   Excellent wine list.

4. **In the field for the findByRestaurantName method, type `French Lemon` and click the Invoke button.**

   The results look like this, showing that the French Lemon record was returned:

   Results of the Last Method Invocation

   size = 2

   Method Invoked: *findByRestaurantName (java.lang.String)*
   Parameters:
   French Lemon

5. **In the Navigation cell (upper left), select the CustomerreviewKey instance.**

6. **Find the instance's getReview method and click its Invoke button.**

   The results display the customer review of the instance you created in Step 2 and Step 3, for example:

Results of the Last Method Invocation

Excellent wine list.

Method Invoked: *getReview ()*
Parameters:
none

Continue testing by selecting created objects and invoking their methods.

7. **When you are finished testing, you can quit the browser, point it to a different URL, or do nothing, as you wish.**

Checking the Additions to the Database

Verify that the Customerreview_TestApp application inserted data in the database, as you tested the Restaurant_TestApp in "Checking the Additions to the Database" on page 81:

1. **In the IDE, select the Runtime tab of the Explorer.**
2. **Expand the Databases node, the PointBase connection node, and the Tables node.**
3. **Right-click the CUSTOMERREVIEW table and choose View Data.**

   Your new review record should be displayed in the command editor.

Creating a Session Bean With the EJB Builder

Create a stateless session bean to manage the conversation between the client (the web service you will create in Chapter 4) and the entity beans.

Note – The source code for the completed session bean is provided in Appendix A.

In version 2.0 of the EJB architecture, session beans can have local interfaces, remote interfaces, or both. In this tutorial, the session beans' methods will be called by the test application (which is local to the session bean), the web services (also local), and the client (remote). Therefore, create a session bean with both local and remote interfaces.

1. **In the Sun ONE Studio 5 Explorer, right-click the Data package and choose New → All Templates.**

   The New wizard is displayed, showing the Choose Templates page.

2. **Expand the J2EE node and select Session EJB.**

3. **Click Next.**

   The Session Bean Name and Properties page is displayed.

   i. **Type `DiningGuideManager` in the Name field and select the following options:**

Option Category	Option to Select
State	Stateless
Transaction Type	Container Managed
Component Interfaces	Both Remote and Local Interfaces

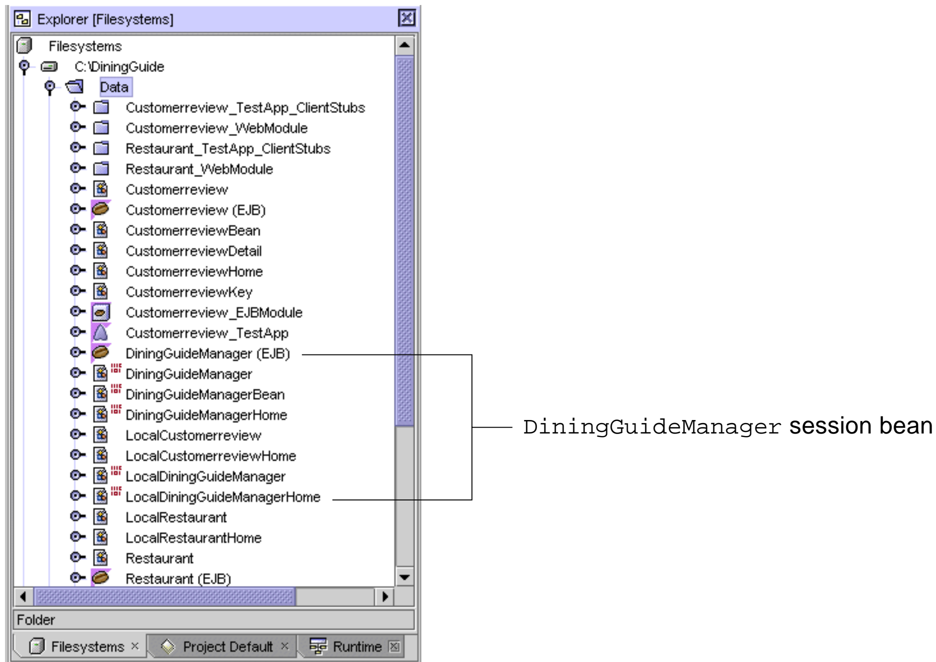
4. **Click Next.**

   The Session Bean Class Files page of the wizard is displayed, listing all the components that will be created for this session bean.

   Notice that the names of the all the components are based on DiningGuideManager.

5. **Click Finish.**

   The new DiningGuideManager session bean is displayed in the Explorer.



Now create the session bean's methods.

Coding a Session Bean's Create Method

The create method was created when you created the DiningGuideManager session bean. You will now modify it.

Create methods of stateless session beans have no arguments, because session beans do not maintain an ongoing state that needs to be initialized. The create method of the DiningGuideManager session bean must first create an initial context, which it then uses to get the required remote references.

1. **Locate the DiningGuideManager's create method within the logical bean's Create folder, and double-click it to display it in the Source Editor.**

2. **Begin coding the method with a JNDI lookup for a remote reference to the RestaurantHome interface.**

```
public void ejbCreate(){
// Join the following two lines in the Source Editor
       System.out.println("Entering
DiningGuideManagerEJB.ejbCreate()");
       Context c = null;
       Object result = null;

       if (this.myRestaurantHome == null) {
              try {
                  c = new InitialContext();
                  result =
c.lookup("java:comp/env/ejb/Restaurant");
                  myRestaurantHome =
(RestaurantHome)javax.rmi.PortableRemoteObject.narrow (result,
RestaurantHome.class);
              }
              catch (Exception e) {System.out.println("Error: "+
e); }
       }
```

Tip – Remember, you can reformat the code you enter in the Source Editor by pressing Control-Shift F. Also remember to remove line breaks when indicated by the code comments.

---

3. **Under the preceding code, add a similar JNDI lookup for the CustomerreviewHome interface.**

```
       Context crc = null;
       Object crresult = null;

       if (this.myCustomerreviewHome == null) {
               try {
                   crc = new InitialContext();
                   result =
crc.lookup("java:comp/env/ejb/Customerreview");
                   myCustomerreviewHome =
(CustomerreviewHome)javax.rmi.PortableRemoteObject.narrow(result
, CustomerreviewHome.class);
               }
               catch (Exception e) {System.out.println("Error: "+
e); }
       }
```

4. **Now add an import statement for the** `javax.naming` **package.**

   Add the import statement at the top of the file. You must import javax.naming because it contains the `lookup` method you just used.

```
import javax.naming.*;
```

5. **Declare the myRestaurantHome and myCustomerreviewHome fields**.

   Add these declarations to the definition of the `DiningGuideManagerEJB` session bean after the `import` statements.

```
public class DiningGuideManagerBean implements
javax.ejb.SessionBean {
       private javax.ejb.SessionContext Context;
       private RestaurantHome myRestaurantHome;
       private CustomerreviewHome myCustomerreviewHome;
```

6. **Select the (DiningGuideManager(EJB) logical node and press F9 to compile the bean.**

   Next, create the DiningGuideManager’s business methods.

Creating Business Methods to Get the Detail Data

The DiningGuideManager bean requires a method that retrieves all restaurant data when it receives a request from the client to see the list of restaurants. It also requires a method to retrieve review data for a specific restaurant when the client requests a list of customer reviews. Create the getAllRestaurants and getCustomerreviewsByRestaurant methods to satisfy these requirements.

Creating the getAllRestaurants Method

To create the getAllRestaurants business method:

1. **Right-click the DiningGuideManager logical node and choose Add Business Method.**

   The Add New Business Method dialog box is displayed.

2. **Type `getAllRestaurants` in the Name field.**

3. **Type `java.util.Vector` in the Return Type field.**

4. **Make sure the option Both Remote and Local Interfaces is enabled.**

   This method should be added to both remote and local interfaces.

5. **Accept all other default values and click OK.**

   The method shell is created in the DiningGuideManager session bean's business methods.

6. **Open the method in the Source Editor and add the following (bold only) code:**

```
public java.util.Vector getAllRestaurants() {
// Join the following two lines in the Source Editor
       System.out.println("Entering
DiningGuideManagerEJB.getAllRestaurants()");
       java.util.Vector restaurantList = new java.util.Vector();
       try {
              java.util.Collection rl =
myRestaurantHome.findAll();
              if (rl == null) { restaurantList = null; }
              else {
                  RestaurantDetail rd;
                  java.util.Iterator rli = rl.iterator();
                  while ( rli.hasNext() ) {
                         rd =
((Restaurant)rli.next()).getRestaurantDetail();

System.out.println(rd.getRestaurantname());
                         System.out.println(rd.getRating());
                         restaurantList.addElement(rd);
                  }
              }
       }
       catch (Exception e) {
// Join the following two lines in the Source Editor
              System.out.println("Error in
DiningGuideManagerEJB.getAllRestaurants(): " + e);
       }
// Join the following two lines in the Source Editor
       System.out.println("Leaving
DiningGuideManagerEJB.getAllRestaurants()");
       return restaurantList;
}
```

Note – Remember to remove line breaks when indicated by the code comments.

This code gets an instance of RestaurantDetail for each remote reference of the Restaurant bean in the context, adds it to a vector called restaurantList, and returns this vector.

7. **Select the DiningGuideManager(EJB) logical node and press F9 to compile the bean.**

Now, create a similar method to get a list of customer reviews.

Creating the getCustomerreviewsByRestaurant Method

To create the getCustomerreviewsByRestaurant method:

1. **Right-click the DiningGuideManager logical node and choose Add Business Method.**

   The Add New Business Method dialog box is displayed.

2. **Type `getCustomerreviewsByRestaurant` in the Name field.**
3. **Type `java.util.Vector` in the Return Type field.**
4. **Click the Add button to add a parameter with the following values:**

Parameter Name	Type
restaurantname	java.lang.String

5. **Click OK to close the dialog box and create the method parameter.**
6. **Make sure the option Both Remote and Local Interfaces is enabled.**
7. **Accept all other default values and click OK again create the business method.**

   The method is created in the `DiningGuideManager` session bean.

8. **Find the method in the Source Editor and add the following bold code:**

```
public java.util.Vector
getCustomerreviewsByRestaurant(java.lang.String
              restaurantname) {
// Join the following two lines in the Source Editor
       System.out.println("Entering
DiningGuideManagerEJB.getCustomerreviewsByRestaurant()");
       java.util.Vector reviewList = new java.util.Vector();
       try {
              java.util.Collection rl =
myCustomerreviewHome.findByRestaurantName(restaurantname);
              if (rl == null) { reviewList = null; }
              else {
                  CustomerreviewDetail crd;
                  java.util.Iterator rli = rl.iterator();
                  while ( rli.hasNext() ) {
                         crd =
((Customerreview)rli.next()).getCustomerreviewDetail();

System.out.println(crd.getRestaurantname());

System.out.println(crd.getCustomername());
                         System.out.println(crd.getReview());
                         reviewList.addElement(crd);
                  }
              }
       }
       catch (Exception e) {
// Join the following two lines in the Source Editor
              System.out.println("Error in
DiningGuideManagerEJB.getCustomerreviewsByRestaurant(): " + e);
       }
// Join the following two lines in the Source Editor
       System.out.println("Leaving
DiningGuideManagerEJB.getCustomerreviewsByRestaurant()");
       return reviewList;
}
```

Note – Remember to remove line breaks when indicated by the code comments.

Similar to the getAllRestaurants code, this method retrieves an instance of CustomerreviewDetail for each remote reference of the Customerreview bean in the context, adds it to a vector called reviewList and returns this vector.

9. **Select the DiningGuideManager(EJB) logical node and press F9 to compile the bean.**

Creating a Business Method to Create a Customer Review Record

Now create a business method that calls the Customerreview entity bean's create method to create a new record in the database.

To create the createCustomerreview method:

1. **Right-click the DiningGuideManager logical node and choose Add Business Method.**

   The Add New Business Method dialog box is displayed.

2. **Type `createCustomerreview` in the Name field.**
3. **Type `void` in the Return Type field.**
4. **Use the Add button to add three parameters with the following values:**

Parameter Name	Type
restaurantname	java.lang.String
customername	java.lang.String
review	java.lang.String

5. **Click OK to close the Add Parameter dialog box.**
6. **Make sure the option Both Remote and Local Interfaces is enabled.**
7. **Click OK again create the business method.**

   The method is created in the DiningGuideManager session bean.

8. **Find the method in the Source Editor and add the following bold code:**

```
public void createCustomerreview(java.lang.String restaurantname,
java.lang.String customername, java.lang.String review) {
// Join the following two lines in the Source Editor
       System.out.println("Entering
DiningGuideManagerEJB.createCustomerreview()");
       try {
              Customerreview customerrev =
myCustomerreviewHome.create(restaurantname, customername,
review);
       } catch (Exception e) {
// Join the following two lines in the Source Editor
              System.out.println("Error in
DiningGuideManagerEJB.createCustomerreview(): " + e);
       }
// Join the following two lines in the Source Editor
       System.out.println("Leaving
DiningGuideManagerEJB.createCustomerreview()");
}
```

Note – Make sure you eliminate the three line breaks indicated by the comments.

---

9. **Select the (DiningGuideManager(EJB) logical node again and press F9 to compile the bean.**

Creating Business Methods That Return Detail Class Types

The web service you will create in Chapter 4 is a JAX-RPC implementation of the SOAP RPC web service. SOAP (Simple Object Access Protocol) is an abstract messaging technique that allows web services to communicate with one another using HTTP and XML. The SOAP runtime must know of all the Java types employed by any methods that are called by the web service in order to map them properly into XML. Because the tutorial's web service will call session bean methods, it needs to know every type used by those methods.

One type the SOAP runtime can not have knowledge of is the type of objects that make up collections. The methods that you just created (getAllRestaurants and getCustomerreviewsByRestaurant) all return collections of the detail classes. You

must provide knowledge of these classes to the SOAP runtime by creating, for each detail class, a method that returns the class. The methods you will create are the getRestaurantDetail and getCustomerreviewDetail methods.

You created methods with the same names on the entity beans (see "Creating Business Methods on the Entity Beans to Fetch the Detail Classes" on page 71), but the methods you create now are empty, their purpose being simply to supply the required return type to the SOAP runtime.

For more information on Sun ONE Studio 5 web services and the SOAP runtime, see *Building Web Services* in the Sun ONE Studio 5 Programming series.

Creating the getRestaurantDetail Method

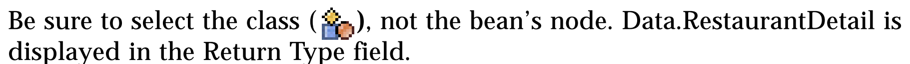
To create the getRestaurantDetail method:

1. **Right-click the DiningGuideManager logical node and choose Add Business Method.**

   The Add New Business Method dialog box is displayed.

2. **Type `getRestaurantDetail` in the Name field.**

3. **For the return type, use the Browse button to select the RestaurantDetail class.**

   Be sure to select the class ( ), not the bean's node. Data.RestaurantDetail is displayed in the Return Type field.

4. **Make sure the option Both Remote and Local Interfaces is enabled.**

5. **Accept all other default values and click OK to create the business method and close the dialog box.**

   The method is created in the DiningGuideManager session bean.

6. **Find the method in the Source Editor and add the following bold code:**

```
public Data.RestaurantDetail getRestaurantDetail() {
        return null;
}
```

Creating the getCustomerreviewDetail Method

To create the getCustomerreviewDetail method:

1. **Right-click the DiningGuideManager logical node and choose Add Business Method.**

   The Add New Business Method dialog box is displayed.

2. **Type `getCustomerreviewDetail` in the Name field.**
3. **For the return type, use the Browse button to select the** `CustomerreviewDetail` **class.**

   Data.CustomerreviewDetail is displayed in the Return Type field.
4. **Make sure the option Both Remote and Local Interfaces is enabled.**
5. **Click OK to create the business method and close the dialog box.**

   The method is created in the DiningGuideManager session bean.
6. **Find the method in the Source Editor and add the following bold code:**

```
public Data.CustomerreviewDetail getCustomerreviewDetail() {
        return null;
}
```

7. **Right-click DiningGuideManager(EJB) and choose Validate EJB.**

   The DiningGuideManager session bean should validate without errors.

Adding EJB References

When you deploy a session bean, the bean's properties must contain references to any entity beans methods called by the session bean. Add them to the session bean now; you can not add them after the bean has been assembled into an EJB module.

1. **In the Explorer, select the DiningGuideManager(EJB) logical node.**
2. **Display the bean's property sheet.**

   If the Properties window is not already visible, choose View → Properties.
3. **Select the References tab of the property window.**
4. **Click the EJB References field and then click the ellipsis (…) button.**

   The EJB References property editor is displayed.
5. **Click the Add button.**

   The Add EJB Reference property editor is displayed.
6. **Type `ejb/Restaurant` in the Reference Name field.**
7. **For the Referenced EJB Name field, click the Browse button.**

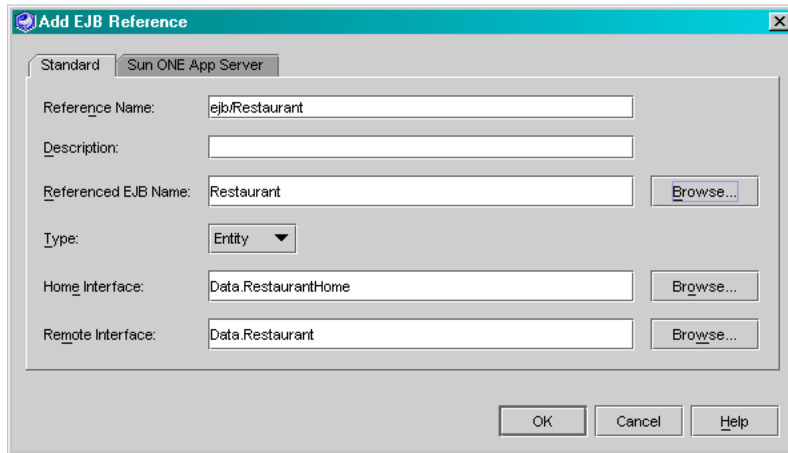
   The Select an EJB browser is displayed.

8. **Select the Restaurant (EJB) bean under the DiningGuide/Data node and click OK.**

   Notice that the Home and Remote interface fields are automatically filled.

9. **Set the Type field to `Entity`.**

   The Add EJB Reference property editor looks like this:



10. **Select the Sun ONE App Server tab.**

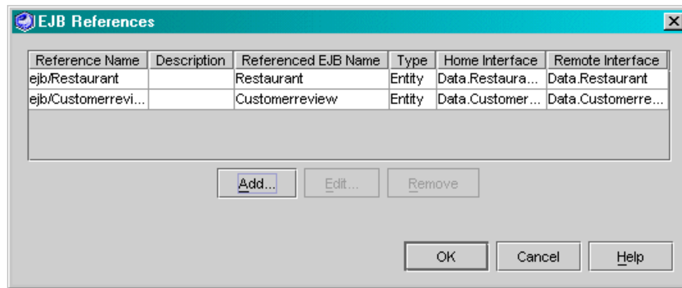
11. **Type `ejb/Restaurant` in the JNDI Name field and click OK.**

    This is the default JNDI name that was assigned to the Restaurant bean when you created it.

12. **Similarly, add a reference to the Customerreview entity bean, so that the properties have the following values:**

Tab	Property	Value
Standard	Reference Name	ejb/Customerreview
	Referenced EJB Name	Customerreview
	Type	Entity
	Home Interface	Data.CustomerreviewHome
	Remote Interface	Data.Customerreview
Sun ONE App Server	JNDI Name	ebj/Customerreview

The EJB References dialog box looks like this:



13. **Click OK to close the Property Editor window.**

    You have now completed the EJB Tier of the tutorial application and are ready to test it. As when you tested the entity beans, the IDE's test application facility creates a web tier and JSP pages that can be read by a client in a browser.

Testing the Session Bean

Use the IDE's test application facility to test the DiningGuideManager session bean. This will test the whole EJB tier, because the session bean's methods provide access to methods on all of the tier's components.

Creating a Test Client for a Session Bean

Create a test application from the DiningGuideManager bean. Then add the two entity beans to the EJB module.

To create a test client for the session bean:

1. **Right-click the DiningGuideManager logical node and choose Create New EJB Test Application.**

   The EJB Test Application wizard is displayed.

2. **Accept all default values and click OK.**

   A progress monitor appears briefly and then goes away when the process is complete. Another window is displayed informing you that the web module that was created is also visible in the Project tab. It should go away automatically, also. If not, click OK to close the window.

3. **View the resulting test objects in the Explorer.**

   The IDE has created the following objects:

   - An EJB module (DiningGuideManager_EJBModule)
   - A web module (DiningGuideManager_WebModule)
   - A J2EE application (DiningGuideManager_TestApp)

   The EJB module and the web module appear as subnodes under the Data package and also as modules contained in the J2EE application. The web module has also been mounted separately.

Adding Entity Bean References to the EJB Module

The EJB module contains only the DiningGuideManager bean, so you must add the two entity beans to it.

1. **Right-click the DiningGuideManager_EJBModule and choose Add EJB.**

   The Add EJB to EJB Module browser is displayed.

2. **Expand the DiningGuide filesystem and the Data package.**

3. **Using Control-Click, select both the Restaurant and Customerreview logical beans.**

4. **Click OK.**

   The IDE adds references to the two entity beans to the EJB module, and to its reference in the test applications. The DiningGuideManager_EJBModule should look like this:

5. **Choose File → Save All.**

Providing the Sun ONE Application Server 7 Plugin With Database Information

You must add database information to the Sun ONE Application Server 7 properties of the EJB module. You performed this task with the entity bean test client in "Providing the Sun ONE Application Server 7 Plugin With Database Information" on page 75.

To add the required information:

1. **Expand the EJB module (DiningGuideManager_EJBModule) in the Explorer and select the Restaurant node (a reference to the Restaurant bean) under it, to display its properties.**

   If the Properties Window is not already displayed, choose View → Properties.

2. **Select the Sun ONE AS tab of the Properties window.**

Note – If there is no Sun ONE AS tab on the Properties window, there is no instance of the Sun ONE Application Server 7 in the Server Registry. See "Confirming Sun ONE Application Server 7 as the Default Server" on page 29 to correct this problem.

---

3. **Confirm that the following three values for the appropriate properties:**

Property	Value
Mapped Fields	7 container managed fields mapped
Mapped Primary Table	RESTAURANT
Mapped Schema	dgSchema

   If these values are displayed, continue with Step 5.

4. **If the values are not displayed, remap the Restaurant bean as follows:**

   a. **Set the Mapped Schema property to** `Data/dgSchema`**.**

   b. **Set the Mapped Primary Table property to** `Restaurant`**.**

   c. **Click in the value field of the Mapped Fields, then click on the ellipsis button.**

      The Map to Database wizard is displayed.

   d. **Click Next to view the Select Tables page.**

   e. **Select** `RESTAURANT` **from the drop list of the Primary Table field.**

      If RESTAURANT is not in the list, use the Browse button to find the table with the dgSchema schema.

f. **Click Next to view the Field Mappings page.**

g. **If the fields are unmapped, click the Automap button.**

Values for mappings appear for each field.

h. **Click Finish.**

The values should now display as in Step 3.

5. **Repeat Step 1 through Step 4 (if required) for the Customerreview reference.**

6. **Select the EJB module (DiningGuideManager_EJBModule) to display its properties.**

7. **Select the Sun ONE AS tab of the properties window.**

8. **Click in the value field for the CMP Resource property, then click the ellipsis button.**

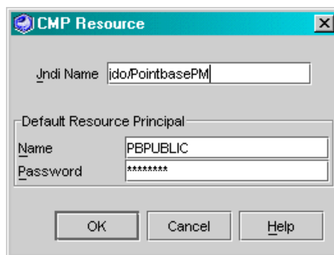
   The CMP Resource property editor is displayed.

9. **Type `jdo/PointbasePM` in the Jndi Name field.**

   This is the JNDI name of the JDBC Persistence Manager you defined in "Setting JDBC Resources (Microsoft Windows Superusers)" on page 30 or "Setting JDBC Resources (All Other Users)" on page 31.

10. **For the Name and Password fields, type the User Name and Password for your database.**

    For the PointBase sample database, these are both PBPUBLIC. The editor looks similar to this:



11. **Click OK to accept the values and close the property editor.**

    You have finished configuring the test application to use your database and now you can deploy the test application.

12. **Save your work with File → Save All.**

Deploying and Executing the Test Application

You must first undeploy the two test applications of the entity beans (if they are still deployed) before you deploy the session bean's test application. This is because they use the same JNDI lookups to the Restaurant and Customerreview beans that are used by the DiningGuideManager_TestApp application. If you fail to undeploy these applications, the DiningGuideManager test application will deploy, but will not load.

Note – Make sure your application server instance is running and is set as the default application server, before you proceed with this section.

Undeploying the Entity Bean Test Applications

To undeploy any previously deployed applications.

1. **Click the Explorer's Runtime tab to display the Runtime page.**
2. **Expand the** *servername*(*hostname*:*host-number*) **instance node under the Sun ONE Application Server 7 node under Installed Servers.**

   For example, server1(localhost:80) is the default server instance. Or your instance might have other labels, such as MyServer(localhost:82).
3. **Make sure the application server is running.**
   a. **Right-click the application server node and choose Status.**

      The Application Server Instance Status dialog box is displayed. If the Status is Running, go on to Step 4.

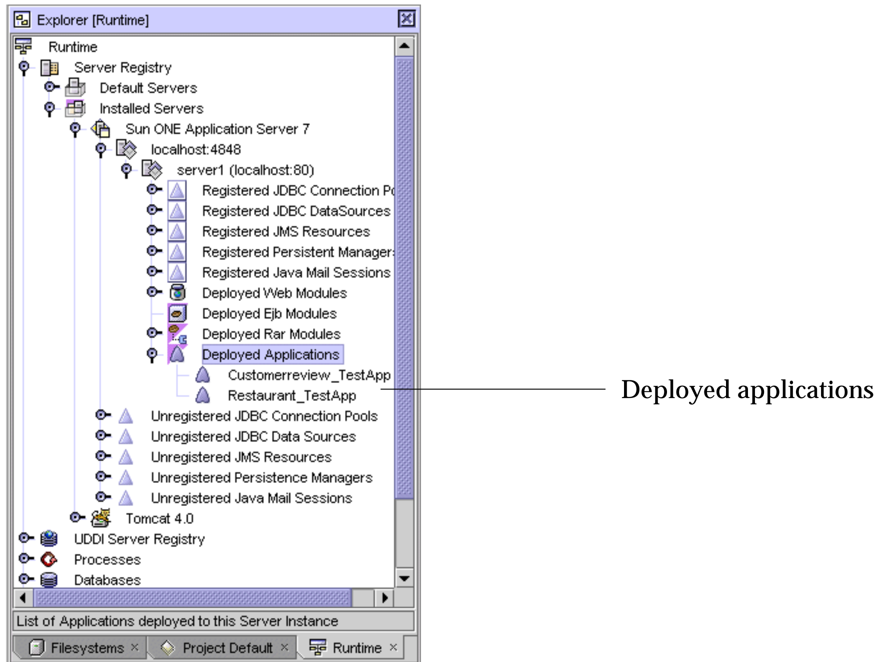
   b. **Click the Start Server button.**

      A command window appears, displaying progress messages. The server is started when this window displays the string "Application onReady complete."

   c. **Click Close on the instance status dialog box.**

      Do not close the command window, although you can minimize it if necessary.
4. **Expand the Deployed Applications node.**

   The two entity bean test applications are displayed.



If nothing is displayed, right-click the Deployed Applications node and choose Refresh List.

5. **Right-click one of the applications and choose Undeploy.**

   The application is undeployed.

6. **Repeat Step 5 to undeploy the other application.**

Deploying the DiningGuideManager Test Application

Note – Make sure the PointBase server is running before you deploy the test application, or any other J2EE application that accesses the database.

To deploy the DiningGuideManager test application:

1. **Click the Explorer's Filesystems tab to display the Filesystems page.**

2. **Right-click the DiningGuideManager_TestApp J2EE application node and choose Execute from the contextual menu.**

   A Progress Monitor window shows the progress of the deployment process. The server instance's log file tab on the output window displays progress messages. The application is successfully deployed when you see success messages.

   The IDE starts the default web browser and displays the test application's home URL, similar to `http://localhost/DiningGuideManager_TestApp/` if your application server is installed locally; it will be different if it is installed remotely.

Using the Test Client to Test a Session Bean

On the test client's web page, create an instance of the DiningGuideManager session bean by exercising the create method; then test the business methods (getRating) on that instance.

To test the DiningGuideManager bean:

1. **Create an instance of the DiningGuideManager session bean by invoking the DiningGuideManagerHome's create method.**

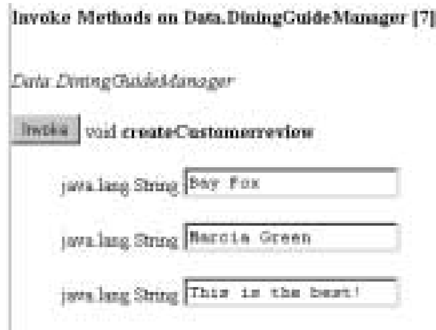
   The Data.DiningGuideManager[*x*] instance appears in the instance list. Now you can test the bean's getter methods.

2. **Select the new Data.DiningGuideManager[*x*] instance.**

   The getAllRestaurants and getCustomerreviewsByRestaurant methods are made available.

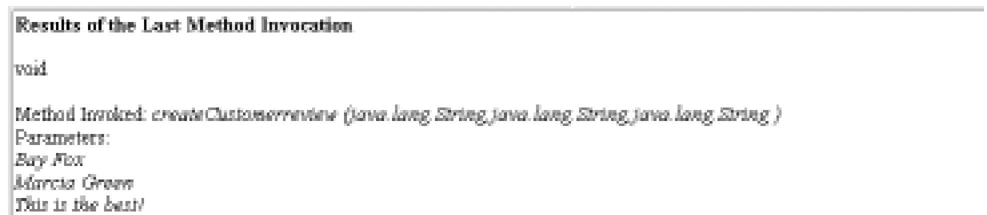
3. **Type any data you like in the createCustomerreview fields.**

For example:



4. **Click the Invoke button next to the createCustomerreview method.**

   The deployed test application adds the record you created to the database. The new parameter values are listed in the Stored Objects section (upper right), and the results are shown in the Results area:



5. **Click the Invoke button on the getAllRestaurants method.**

   If you created Joe's House of Fish in the database (in "Using the Test Client to Test the Restaurant Bean" on page 78), a vector of size 3 appears in the list of created objects (upper right), and the results of the method invocation should look as shown (actual numbers may be different). If you didn't create this record, your results might be different.

6. **Click the Invoke button on the getCustomerreviewDetail method.**

   The result is shown in the Results section.

Results of the Last Method Invocation

null

Method Invoked: *getCustomerreviewDetail ( )*
Parameters:
none

7. **Type `Joe's House of Fish` in the field for the getCustomerreviewsByRestaurant method and click the Invoke button.**

   No CustomerreviewDetail records should be returned, because there are no customer review comments in the database. Now try the French Lemon record.

8. **Type `French Lemon` in the same field and invoke the method.**

   Two CustomerreviewDetail records should be returned:

Results of the Last Method Invocation

[Data.CustomerreviewDetail@61469c, Data.CustomerreviewDetail@62bda7]

Method Invoked: *getCustomerreviewByRestaurant (java.lang.String)*
Parameters:
French Lemon

9. **Click the Invoke button on the getRestaurantDetail method.**

   The result is shown in the Results section.

Results of the Last Method Invocation

null

Method Invoked: *getRestaurantDetail ( )*
Parameters:
none

10. **When you are finished testing, stop the test client by pointing your web browser at another URL or by exiting the browser (or do nothing).**

Note – You do not need to stop the application server's process (which is listed in the execution window). Whenever you redeploy, the server undeploys the application and then redeploys. When you exit the IDE, a dialog box is displayed for terminating the application server's instance process. However, you can manually terminate it at any time by right-clicking the `server1`(***hostname*:*host-port***) node in the execution window and choosing Terminate Process.

Checking the Additions to the Database

To verify that the DiningGuideManager_TestApp application inserted data in the database, repeat the procedures described in "Checking the Additions to the Database" on page 81 and "Checking the Additions to the Database" on page 86.

You are now ready to create the web service.

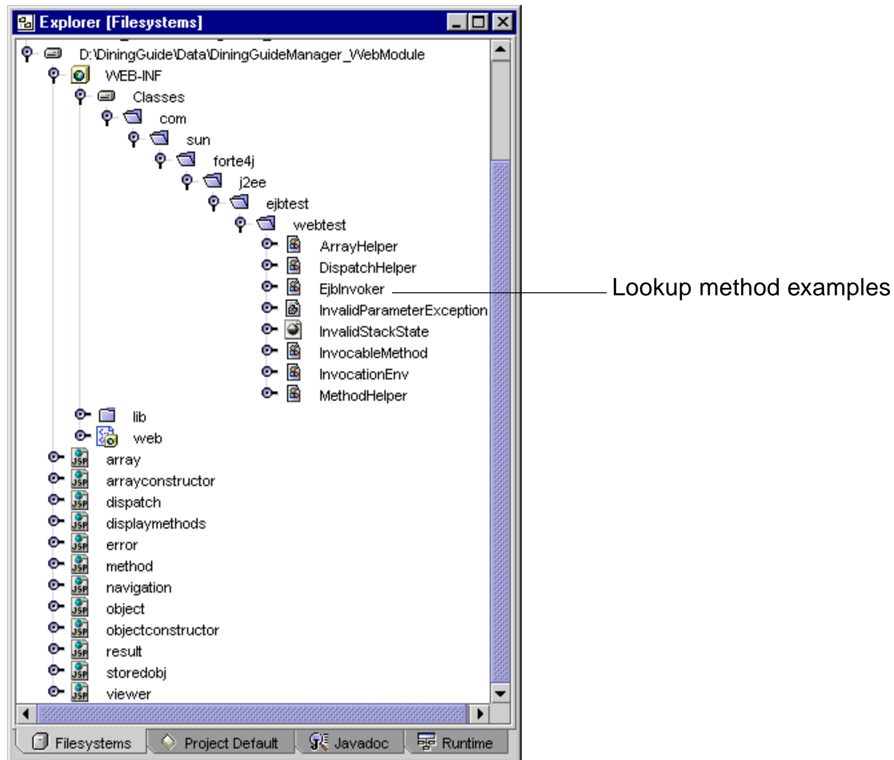
Comments on Creating a Client

Congratulations, you have successfully completed the EJB tier of the DiningGuide application. You are ready to go on to Chapter 4, to use the Sun ONE Studio 5 IDE's Web Services module to create web services for the application, and then on to Chapter 5 to install the provided Swing classes for your client.

You may, however, wish to create your own web services and client, in which case, the Sun ONE Studio 5 test application can offer some guidelines.

Web services that access a session bean like the DiningGuideManager bean must include a servlet and JSP pages with lookup methods for obtaining the Home interfaces and Home objects of the entity beans in the EJB tier. The web module created by the test application facility offers examples of the required code.

Lookup method examples are found in the EjbInvoker class under the web module. Specifically, look for this class under the `WEB-INF/Classes/com/sun/forte4j/j2ee/ejbtest/webtest` directory.



For example, the following methods offer good example lookup code:

- EjbInvoker.getHomeObject
- EjbInvoker.getHomeInterface
- EjbInvoker.resolveEjb

CHAPTER 4

Creating the DiningGuide Application's Web Service

This chapter describes how to use the Sun ONE Studio 5 IDE to create web services for the DiningGuide application. This chapter covers the following topics:

- "Overview of the Tutorial's Web Service," which follows
- "Creating the Tutorial's Web Service" on page 113
- "Testing the Web Service" on page 117
- "Making Your Web Service Available to Other Developers" on page 129

For a complete discussion of Sun ONE Studio 5 web service features, see *Building Web Services* from the Sun ONE Studio 5 Programming series. This book is available from the Sun ONE Studio 5 portal's Documentation page at `http://forte.sun.com/ffj/documentation/index.html`. For information on specific features, see the Sun ONE Studio 5 online help.

Overview of the Tutorial's Web Service

In this chapter, you will create the DiningGuide application's web service. As part of this procedure you will explicitly create a number of components and generate some others.

You will explicitly create:

- A logical web service, the DGWebService web service
- A J2EE application, which references both the session bean's EJB module and the web service

You will generate:

- Runtime classes, which are EJB components for implementing the web service
- A test client
- Test client files

The Web Service

For more complete information about web services and how to create and program them, see *Building Web Services*. See also the Sun ONE Studio 5 online help for specific web service topics and procedures.

In this tutorial, you develop your web service's functionality by creating references in the web service to the methods you want clients to be able to access.

The Runtime Classes

After creating your web service and its method references, you generate its runtime classes. You do not work directly on the runtime classes, but you will see them generated in the package containing the web service.

The Client Files

When you create a client and generate its files, supporting client pages are created. You will use these client pages for testing the web service. You can also use them as a starting point or a guide for developing a full-featured referenced method. These client files include a JSP page for each reference method, a JSP page to display errors for the web service, and an HTML welcome page to organize the method JSP pages for presentation in a web browser.

The welcome page contains one HTML form for each of the JSP pages generated for the referenced methods. If a method requires parameters, the HTML form contains the appropriate input fields. You test the methods by inputting data for each parameter, if required, and pressing the method's Invoke button. The following actions then occur:

1. The JSP page passes the request to the SOAP client file.
2. The SOAP client file passes the request to the JAX-RPC runtime system on the application server.

   A SOAP request is an XML wrapper that contains a method call on the web service and input data in a serialized form.
3. The JAX-RPC runtime system on the application server transforms the SOAP requests into a method call on the appropriate method referenced by the DiningGuide web service.
4. The method call is passed to the appropriate business method in the EJB tier.
5. The processed response is passed back up the chain to the SOAP client file.

6. The SOAP client file passes the response to the JSP page, which displays the response on a web page.

Creating the Tutorial's Web Service

Creating the tutorial's web service requires two steps:

1. Creating the logical web service for the application.

   Use the IDE's Web Service wizard to create the logical web service and specify the methods you want to reference. These are the five business methods you created for the DiningGuideManager session bean.

2. Generating the web services runtime classes.

   This task generates the supporting EJB components that are used for testing and implementing the web service.

Creating the Logical Web Service

Use the New Web Service wizard to create the logical web service. The wizard offers a choice of architectures: multitier or web-centric. The DiningGuide application's web service calls methods on the EJB tier components, so choose the multitier architecture.

The wizard also prompts you to select the methods the web service will call, so it can build references to these methods. Select the five business methods of the EJB tier's session bean.

To create the tutorial's logical web service:

1. **In the Explorer, right-click the mounted DiningGuide Filesystem and choose New → Java Package.**

   The New Package dialog box is displayed.

2. **Type `WebService` for the name and click Finish.**

   The new WebService package appears under the DiningGuide directory.

3. **Right-click the WebService package and choose New → All Templates.**

   The New wizard is displayed, showing the Choose Template page.

4. **Expand the Web Services node, select Web Service and click Next.**

   The New wizard displays the Web Service page.

5. **Type `DGWebService` in the Name field, select the following options:**

Option Category	Option to Select
Create From	Java Methods
Architecture	Multitier

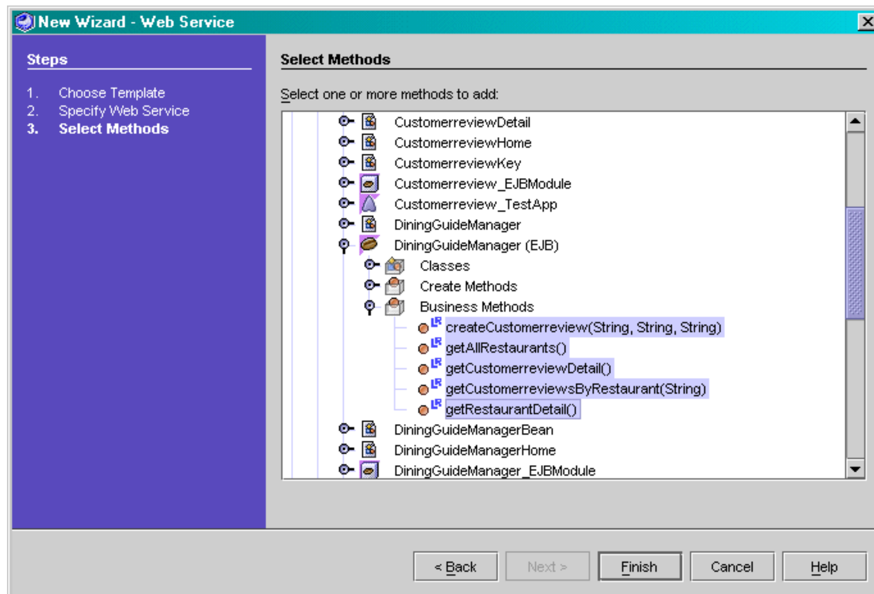
6. **Click Next.**

   The Select Methods page of the New wizard is displayed.

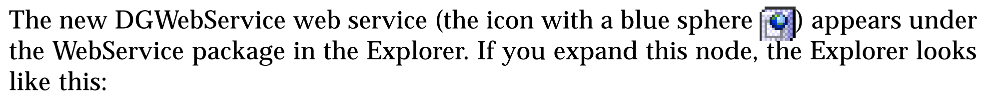
7. **Expand the Data, DiningGuideManager(EJB), and Business Methods nodes.**

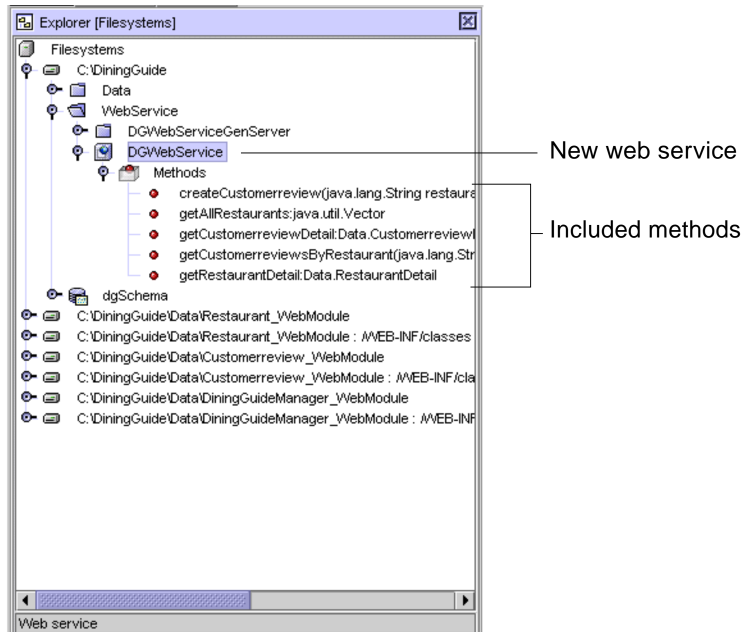
8. **Use Shift-Click to select all the DiningGuideManager's business methods:**

   The Methods page looks like this:



9. **Click Finish.**

   The new DGWebService web service (the icon with a blue sphere ) appears under the WebService package in the Explorer. If you expand this node, the Explorer looks like this:



Generating the Web Service's Runtime Classes

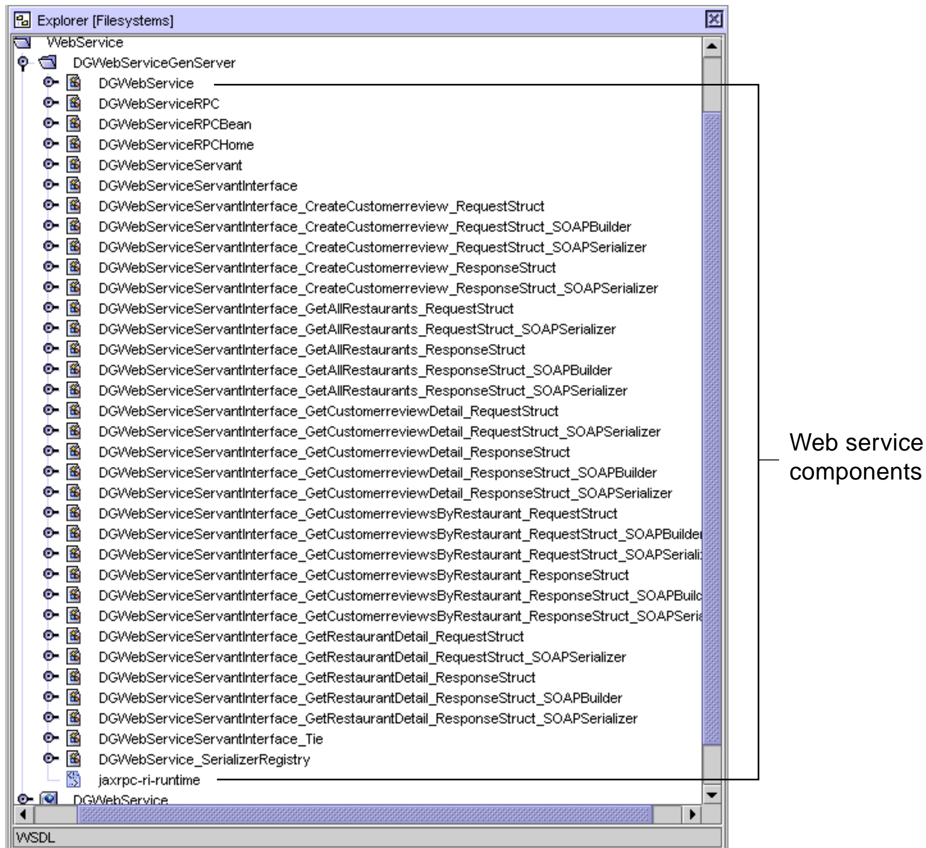
Before you can assemble the web service as a J2EE application and deploy it for testing, you must generate the web service's runtime classes. When the architecture is multitier, the IDE generates many classes to implement the web service, three of which are for a generated EJB component.

Note – Make sure your admin server is running and your application server is set as the default server.

To generate a web service's runtime classes:

1. **Right-click the DGWebService node and choose Generate Web Service Files.**

   When the operation is complete, the word "Finished" appears in the IDE's output window. Runtime classes that are EJB components for implementing the SOAP RPC web service appear in the Explorer:



2. **Display the DGWebService node's properties.**

   Either view them in the Properties window beneath the Explorer, or right-click the node and choose Properties from the contextual menu.

3. **Verify that the value of the SOAP RPC URL property reflects the hostname and portnumber of your application server.**

   For example, for an application server host name of "tech5" and a port number of "4855," the URL should be as follows:

   ```
   http://tech5:4855/DGWebService/DGWebService
   ```

Testing the Web Service

Testing your web service requires the following tasks:

1. Creating a test client that includes:
   - A test client
   - A J2EE application that references both the EJB module the web service
2. Deploying the test application.
3. Using the test application to test the web service.

The Web Services test application generates a JSP page for each XML operation in the web service, plus a welcome page to organize them for viewing in a browser. When you execute the test client, you exercise the XML operations from the welcome page.

Creating a Test Client and Test Application

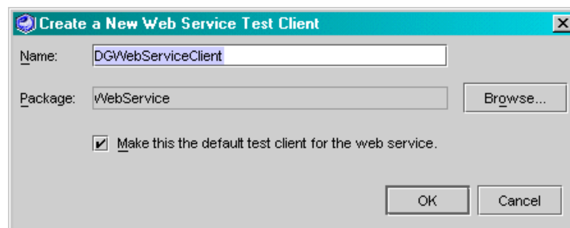
To test your web service, create a test client and a J2EE application. Add the EJB modules and the web service module to the J2EE application.

Tip – When you create the test client, make it the default test client for the web service. Then when you deploy the J2EE application, the test client is deployed as well.

To create and deploy a client application for your web service:

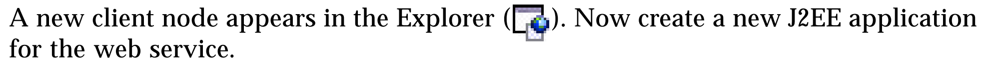
1. **In the Explorer, right-click the DGWebService node ( ) and choose Create New Web Service Test Client.**

   The Create a New Web Service Test Client dialog box is displayed.



The option to make this client the default test client for the web service is selected by default.

2. **Accept all the defaults and click OK.**

   A new client node appears in the Explorer (). Now create a new J2EE application for the web service.

3. **Right-click the WebService package and choose New → All Templates.**

4. **In the New wizard, expand the J2EE node, select Application and click Next.**

5. **Type `DGApp` in the Name field and click Finish.**

   The new J2EE application node ( ) appears under the WebService package.

Adding the Web Service to the J2EE Application

Now add the web service to the application:

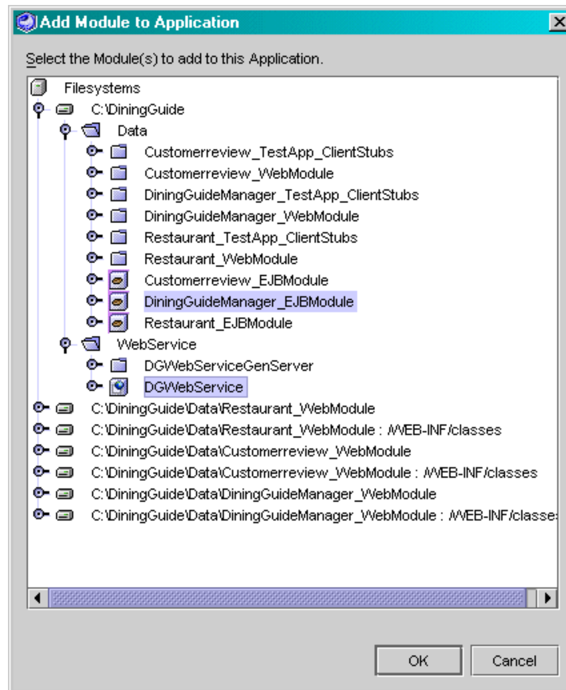
1. **Right-click the DGApp node and choose Add Module.**

   The Add Module to Application dialog box appears.

2. **In the dialog box, expand the DiningGuide filesystem and both the Data and WebService packages.**

3. **Using Control-Click, select both the DiningGuideManager_EJBModule and the DGWebService nodes.**

The dialog box looks like this:



4. **Click OK to accept the selection and close the dialog box.**

5. **In the Explorer, expand the DGApp J2EE application.**

   Both the DGWebService's WAR and EJB JAR files have been added to the application, as well as the DiningGuideManager_EJBModule:

Deploying the Test Application

You must first undeploy the any deployed test applications before you deploy the session bean's test application, for the same reason given in "Deploying and Executing the Test Application" on page 104. (Because they use the same JNDI lookups to the Restaurant and Customerreview beans that will be used by the web service test application.)

For the procedure for undeploying an application, see “Undeploying the Entity Bean Test Applications” on page 104.

Note – Make sure the PointBase server is running before you deploy the test application, or any other J2EE application that accesses the database. In addition, make sure your Sun ONE Application Server 7 instance is running and is the default application server of the IDE. See “Confirming Sun ONE Application Server 7 as the Default Server” on page 29 for information.

To deploy the DGApp application:

1. **Undeploy any deployed DiningGuide test applications.**

   Refer to “Undeploying the Entity Bean Test Applications” on page 104 for this procedure, if it is necessary. Make sure you also restart the application server.

2. **In the Explorer, right-click the DGApp node ( ) and choose Deploy.**

   A progress monitor window shows the deployment process running.

3. **Verify that the application is deployed.**

   A Progress Monitor window shows the progress of the deployment process. The server instance’s log file tab on the output window displays progress messages. The application is successfully deployed when you see success messages.

   The Execution View window of the Explorer displays a *servername*(*server-hostname*:*server-port-number*) node.

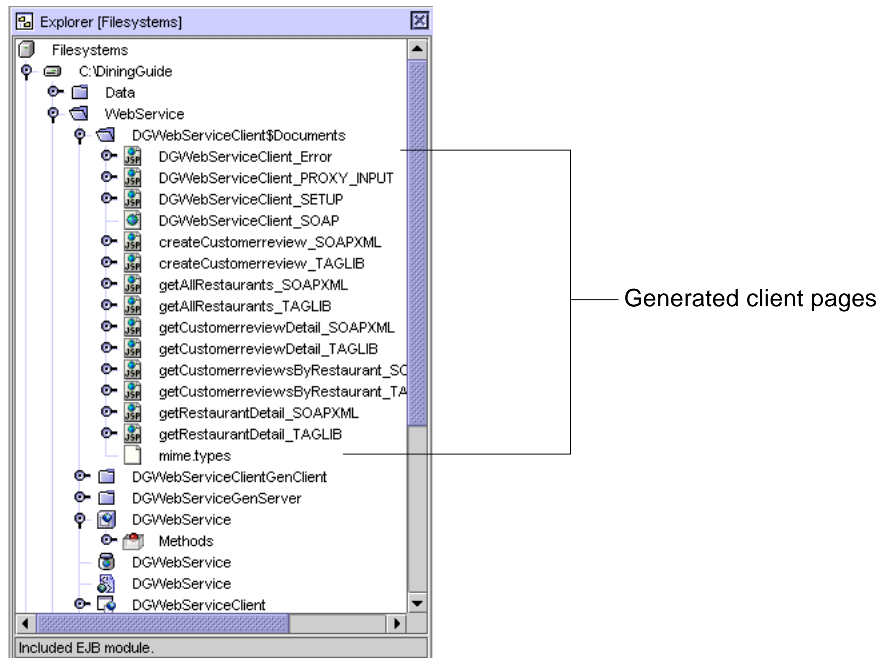
4. **Expand the DGWebServiceClient$Documents node under the WebService package.**

   The following supporting items have been created:

   - A JSP page for each method that formats the display using tag libraries
   - A JSP page for each method for displaying the returned SOAP message
   - A set of JSP pages that compose a welcome page
   - A JSP error page

   See *Building Web Services* for more information about these pages.

The Explorer looks like this:



These files are also referenced under the Generated Documents node under the DGWebServiceClient node.

Using the Test Application to Test the Web Service

For an explanation of the details of how SOAP requests and responses are passed between the client and the web service, see *Building Web Services*, available from the Sun ONE Studio 5 portal's Documentation page at `http://forte.sun.com/ffj/documentation/index.html`.

Note – Make sure the admin server and the PointBase server are running. Make sure your application server is the default application server.

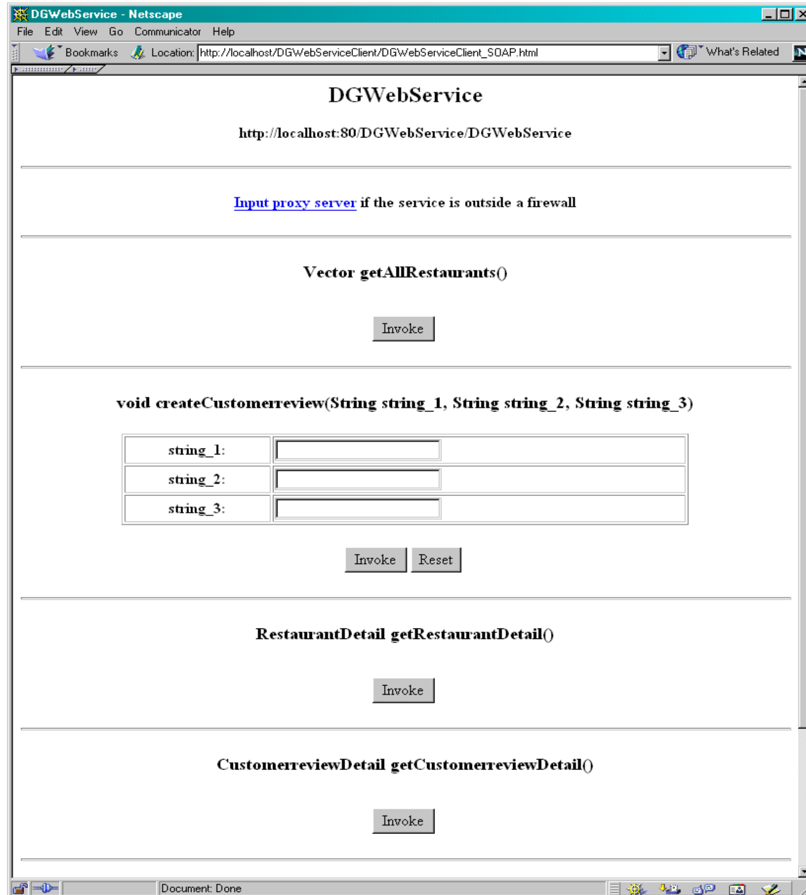
To test the web service:

1. **In the Explorer, right-click the DGWebServiceClient node ( ) and choose Deploy.**

   The DGWebServiceClient is deployed. The node is visible under your application server instance's Deployed Web Modules node in the Runtime pane of the Explorer.

2. **Right-click the same node and choose Execute.**

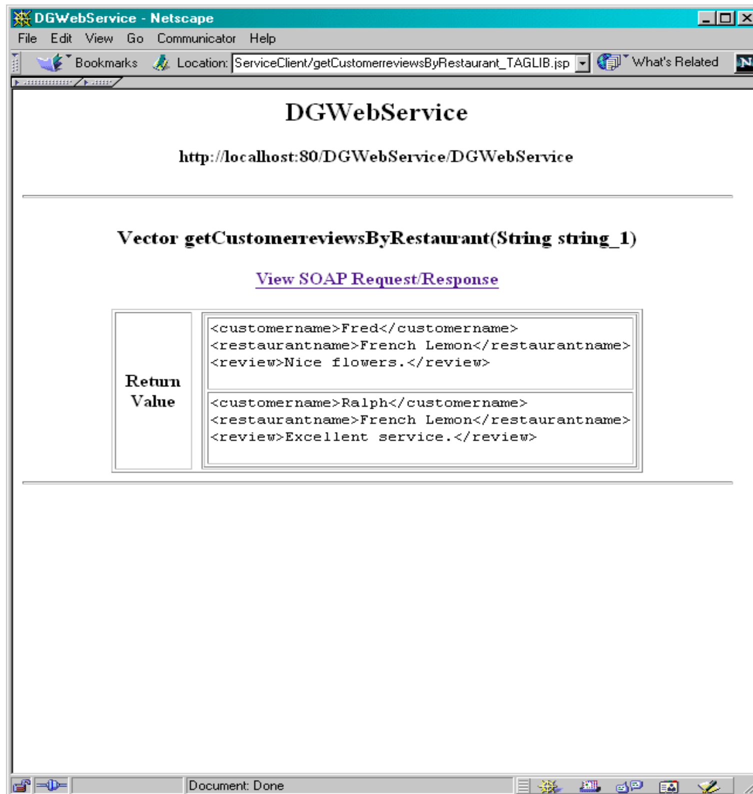
   The IDE deploys the test client, launches the default web browser, and displays the client's generated welcome page (DGWebServiceClient_SOAP.html):



   This page lets you test whether the operations work as expected.

3. **Test the getCustomerreviewsByRestaurant method by typing `French Lemon` in the text field and clicking the Invoke button.**

A SOAP message is created and sent to the application server. The DGApp web service turns the SOAP message into a method invocation of the DiningGuideManager.getCustomerreviewsByRestaurant method. This method returns a collection to the generated JSP page, getCustomerreviewsByRestaurant_TAGLIB.jsp, which displays the returned data in a formatted table, as shown.



The data includes all the records entered for the French Lemon restaurant.Verify the data by starting the Pointbase console and running the following SQL statement:

```
select * from CustomerReview;
```

The results show what CustomerReview records you have entered.

4. **To view the SOAP message, click the View SOAP Request/Response link.**

   This is displays the returned data as an XML-wrapped SOAP message.

5. **Use the Back button on your browser to return to the welcome page.**

6. **Test the createCustomerreview operation by typing `French Lemon` in the restaurantname field, and whatever you want in the other two fields.**

For example:

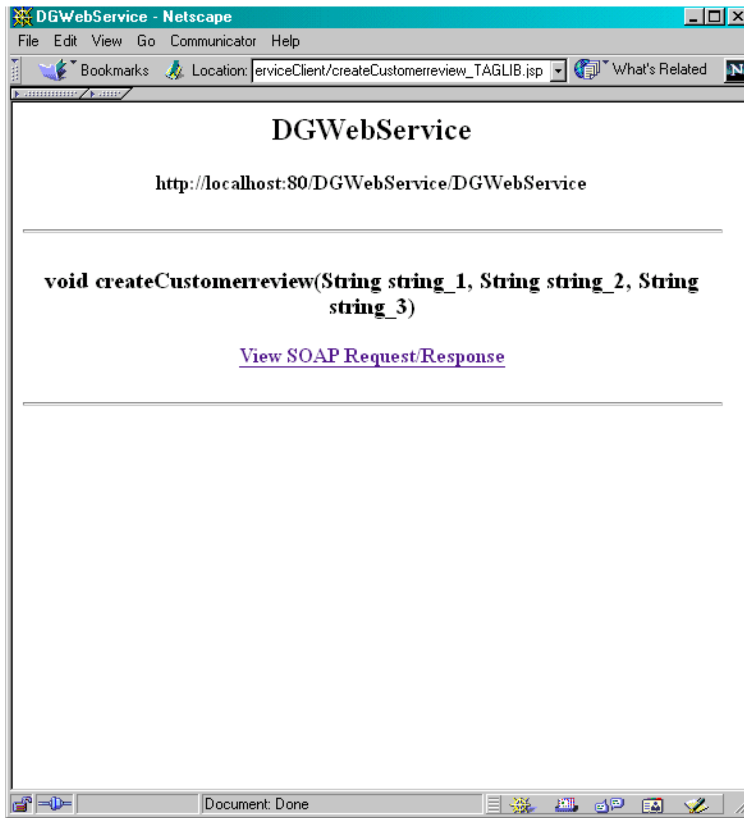
void createCustomerreview(String string_1, String string_2, String string_3)

string_1:	Bay Fox
string_2:	Dick Wagner
string_3:	Great bay views!

Invoke Reset

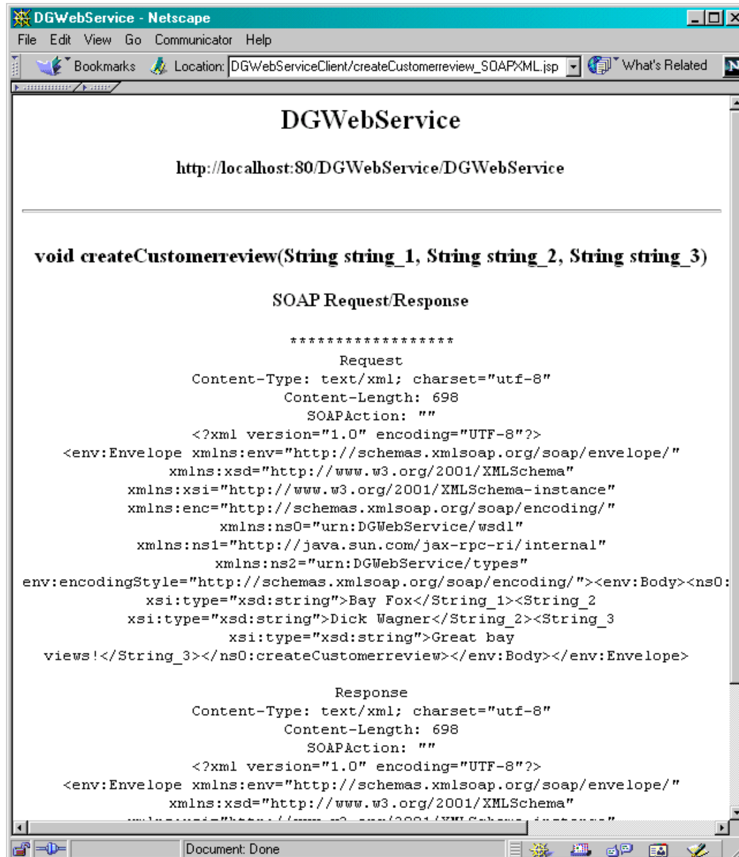
7. **Click the Invoke button.**

   This method takes a complex Java object as an input parameter. The generated JSP page, createCustomerreview_SOAP.jsp, prompts for the three inputs. These are then converted into a Customerreview object and passed as a SOAP message. This message is sent to the application server, where it is turned back into a Java method call and sent to the DiningGuideManagerEJB component. The result is displayed in the browser:



8. **Click the View SOAP Request/Response link.**

The SOAP request and response looks like this.



9. **Use the Back button on your browser to return to the welcome page.**

10. **Test the getAllRestaurants operation by clicking its Invoke button on the welcome page.**

Vector getAllRestaurants()

Invoke

This method does not require an input parameter. It returns a collection of restaurant data, which the getAllRestaurants_SOAP.jsp page displays as XML:

Notice the Restaurant record you entered when you tested the Restaurant entity bean (see "Using the Test Client to Test the Restaurant Bean" on page 78) is the last record on the page.

You have successfully created a web service for the DiningGuide tutorial. In Chapter 5, you will use a provided Swing client to run the DiningGuide application.

Making Your Web Service Available to Other Developers

You have learned a convenient method for testing web services if you are a web services developer. However, other development groups in your organization, particularly the client developers, need to test their work against your web service, as well. You can easily provide them with your web service's WSDL file. From this file, they can generate a client files from which they can build the application's client. They can then test the client against your web service, if you provide them with the URL of your deployed web service (and make sure the web server is running).

To make web services available to other developers involves these tasks:

1. The web services developer:
   - Generates a WSDL file from the web service
   - Makes the WSDL file available to the target user
   - Provides the target user with the URL of the deployed web service
2. The target user:
   - Adds the WSDL file to a mounted filesystem in the Explorer
   - Creates a web service client from this WSDL
   - Generates client files
   - Builds the client around the client files
   - Specifies the web service URL as the SOAP RPC URL property of the client

Generating the client files generates the JSP pages required for developing a real client for the application.

Generating the WSDL File

The first step in sharing access to the application's web service is to generate a WSDL file for the web service. This is performed by the developers of the web service.

To generate a WSDL file for the web service:

1. **In the Explorer, right-click the DGWebService node () and choose Generate WSDL.**

   A WSDL file (the node with a green sphere) named DGWebService is created under the WebService package.

   You can find this file, named `DGWebService.wsdl`, in the `DiningGuide/WebService` directory on your operating system's file system.

2. **Make this file available to other development teams.**

   You can attach the file to an email message or post it on a web site.

Generating Client Files From the WSDL File

The second part of sharing access to the application's web service is to generate all the web service supporting files from the WSDL file. This is performed by the developers of the client.

To generate the web service files and client files from the WSDL file:

1. **On your operating system's file system, create a directory and place the `DGWebService.wsdl` file in it.**

   For example, create the `c:\wsdlHolder` directory and paste the file in it.

2. **In the Sun ONE Studio 5 Explorer, mount this directory.**

3. **Right-click the new directory and choose New → Java Package.**

4. **Type `MyClientPackage` in the Name field and click Finish.**

   MyClientPackage is displayed in the mounted directory.

5. **In the Explorer, right-click MyClientPackage and choose New → All Templates.**

6. **In the New wizard, expand the Web Services node, select Web Service Client and click Next.**

   The Web Service Client page is displayed.

7. **Type `NewClient` in the Name field.**

8. **Make sure the package is the MyClientPackage package.**

9. **For the Source, select the Local WSDL File option and click Next.**

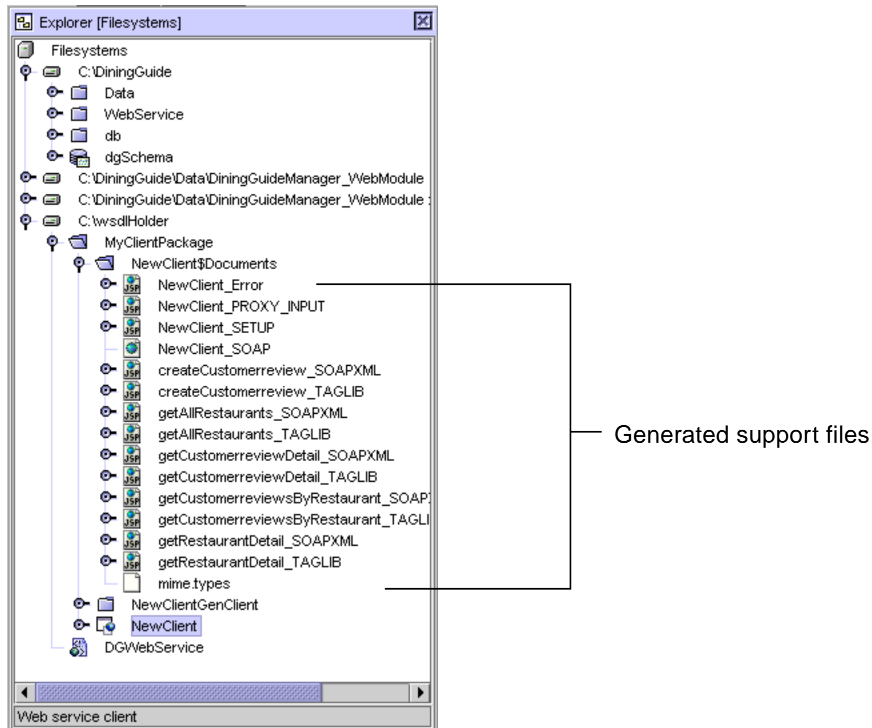
   The Select Local WSDL File page of the New wizard is displayed.

10. **Expand the directory you created in Step 1, select the DGWebService WSDL file, and click Finish.**

    A new client node ( ) appears in the Explorer.

11. **Right-click the NewClient client node and choose Generate Client Files.**

    A Generated Documents node is generated in the Explorer. The expanded Generated Documents node reveals the JSP pages and welcome page required for the client, as shown:



You can now use the client to test the web service, as described in "Using the Test Application to Test the Web Service" on page 121.

When your application is finished, you will probably publish your web service to a UDDI registry, to make it available to developers outside your immediate locale. Sun ONE Studio 5 provides a single-user internal UDDI registry to test this process, and the StockApp example, available from the Sun ONE Studio 5 portal's Examples and Tutorials page at `http://forte.sun.com/ffj/documentation/tutorialsandexamples.html` illustrates how to use this feature. For information on publishing to an external UDDI registry, see *Building Web Services*.

CHAPTER 5

Creating a Client for the Tutorial Application

This chapter shows you how to run the DiningGuide application using a provided Swing client that communicates with the web service you created in Chapter 4.

The provided client contains two Swing classes, RestaurantTable and CustomerReviewTable. You add these classes to the WebService package, then execute the RestaurantTable class to run the application.

This client is very primitive, provided only to illustrate how to access the methods of the client you have generated for the web service.

This chapter covers these topics:

- "Creating the Client With the Provided Code," which follows
- "Running the Tutorial Application" on page 134
- "Examining the Client Code" on page 137

Creating the Client With the Provided Code

The client classes are provided as Java class files in the `DiningGuide.zip` file, which you can download from the Developer Resources portal.

To copy the two provided Java client classes into the DiningGuide application:

1. **In the IDE, create the Client package within the DiningGuide folder.**

   a. **Right-click the DiningGuide folder and choose New → Java Package.**

   b. **In the New wizard, type `Client` in the Name field and click Finish.**

2. **Unzip the `DiningGuide.zip` file from the Developer Resources portal.**

a. **Download the `DiningGuide.zip` file from the Developer Resources portal.**

`http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

b. **Unzip the file to a local directory, for example, the `/MyZipFiles` directory.**

3. **Using a file system command, copy the two client files from the DiningGuide source files to the Client package, as follows:**
   - On Microsoft Windows systems, copy the files and paste them into the new `Client` folder.
   - On Solaris or Linux environments, type a command like this:

```
$ cp /MyZipFiles/DiningGuide/Client/*.java /DiningGuide/Client
```

4. **In the IDE's Explorer, expand the DiningGuide/Client package and verify that the two new classes are there.**

Note – When you create a Swing client in the IDE's Form Editor, the IDE generates a `.form` file and a `.java` file. The `.form` file enables you to edit the GUI components in the Form Editor. However, the `.form` files have not been provided for the RestaurantTable and CustomerReviewTable classes, which prevents you from modifying the GUI components in the Form Editor.

Running the Tutorial Application

Note – Make sure the PointBase server is running before you run the tutorial application. In addition, make sure Sun ONE Application Server 7 is running and is the default application server of the IDE. See "Confirming Sun ONE Application Server 7 as the Default Server" on page 29 for information.

Run the DiningGuide application by executing the RestaurantTable class, as follows:

1. **In the IDE, click the Runtime tab of the Explorer.**
2. **Expand the Server Registry, the Installed Servers, the Sun ONE Application Server 7, and its subnodes.**
3. **Right-click the Deployed Applications subnode of the server instance.**

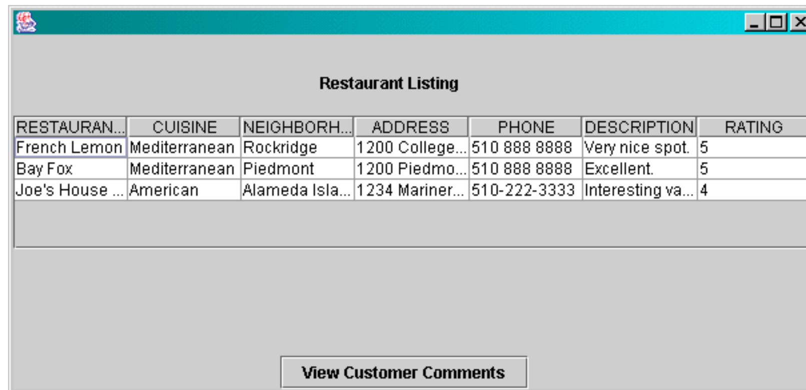
4. **Make sure the DiningGuideApp application is still deployed.**

   If it is still deployed, a DiningGuideApp node is displayed under the Deployed Applications subnode.

5. **If it is not still deployed, deploy it, as described in "Deploying the Test Application" on page 119.**

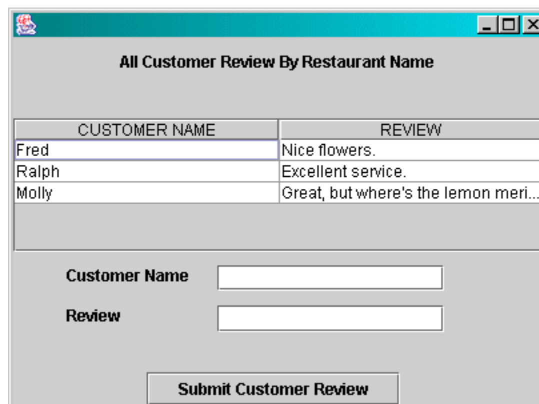
6. **In the Filesystems tab of the Explorer, right-click the RestaurantTable node and choose Execute.**

   The IDE switches to Runtime mode. A Restaurant node appears in the execution window. Then, the RestaurantTable window is displayed, as shown:



7. **Select any restaurant in the table and Click the View Customer Comments button.**

   For example, select the French Lemon restaurant. The CustomerReviewTable window is displayed. If any comments exist in the database for this restaurant, they are displayed, as shown. Otherwise, an empty table is displayed.



8. **Type a something in the Customer Name field and in the Review field and click the Submit Customer Review button.**

For example:

The record is entered in the database and is displayed on the same CustomerReviewTable window, as shown:

9. **Play around with the features, as described in "User's View of the Tutorial Application" on page 37.**

10. **Quit the application by closing any window.**

    After you quit the application, the execution window shows that the Sun ONE Application Server 7 process is still running. You need not stop the application server. If you redeploy any of the tutorial's J2EE applications or rerun the test clients (but not this Swing client), the server is automatically restarted.

    When you quit the IDE, a dialog box is displayed for terminating any process that is still running (including the application server or the web server). Select each running process and click the End Tasks button. You can also manually terminate any process at any time while the IDE is running by right-clicking its node in the execution window and choosing Terminate Process.

Examining the Client Code

The two client classes you have installed in the DiningGuide application are composed of Swing components and actions that were created in the Form Editor, and several methods that were created in the Source Editor. The methods added in the Source Editor include the crucial task of instantiating the client so that its methods become available to the client.

To help you understand how the Swing client interacts with the web service, the next few sections discuss the main actions of the client, namely:

- "Displaying Restaurant Data" on page 137
- "Displaying Customer Review Data for a Selected Restaurant" on page 138
- "Creating a New Customer Review Record" on page 141

Displaying Restaurant Data

Displaying restaurant data is accomplished by the RestaurantTable class's methods, which instantiate the client and call its getAllRestaurants method, as follows:

1. RestaurantTable.getAllRestaurants method instantiates the client, calls the client's getAllRestaurants method to fetch the restaurant data, and returns the fetched restaurant data as a vector.

```
private Vector getAllRestaurants() {
        Vector restList = new Vector();
        try {
                WebService.DGWebServiceClientGenClient.DGWebService service1 =
new
                WebService.DGWebServiceClientGenClient.DGWebService_Impl();

WebService.DGWebServiceClientGenClient.DGWebServiceServantInterface port =
                service1.getDGWebServiceServantInterfacePort();

                restList = (java.util.Vector)port.getAllRestaurants();
        }
        catch (Exception ex) {
                System.err.println("Caught an exception." );
                ex.printStackTrace();
        }
        return restList;
}
```

2. The RestaurantTable constructor puts the returned restaurant data into the restaurantList variable and calls RestaurantTable.putDataToTable.

```
public RestaurantTable() {
        initComponents();
        restaurantList=getAllRestaurants();
        putDataToTable();
}
```

3. The RestaurantTable.putDataToTable method iterates through the vector and populates the table.

```
private void putDataToTable() {
        Iterator j=restaurantList.iterator();
        while (j.hasNext()) {
                RestaurantDetail ci = (RestaurantDetail)j.next();
                String strRating = null;
                String[] str ={ci.getRestaurantname(),
ci.getCuisine(), ci.getNeighborhood(), ci.getAddress(),
ci.getPhone(), ci.getDescription(),
strRating.valueOf(ci.getRating()),
                };
                TableModel.addRow(str);
        }
}
```

4. The RestaurantTable.Main method displays the table as a Swing jTable component.

```
public static void main(String args[]) {
        new RestaurantTable().show();
}
```

Displaying Customer Review Data for a Selected Restaurant

Displaying customer review data begins when the RestaurantTable's button component's action instantiates a CustomerReviewTable. The CustomerReviewTable's methods fetch the customer review data by means of the client's methods, and populate the table. The RestaurantTable's button component's action then displays it, as follows:

1. When the RestaurantTable's button is pressed to retrieve customer review data, the RestaurantTable.jButton1ActionPerformed method instantiates a new CustomerReviewTable object, calls its putDataToTable method, and passes it the data of the selected column.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{
        int r =jTable1.getSelectedRow();
        int c = jTable1.getSelectedColumnCount();
        String i =(String)TableModel.getValueAt(r,0);
        CustomerReviewTable crt = new CustomerReviewTable();
        crt.putDataToTable(i);
        crt.show();
        System.out.println(i);
}
```

2. The CustomerReviewTable.putDataToTable method calls the CustomerReviewTable.getCustomerReviewByName method, passing it the selected restaurant name, assigning the returned vector to the customerList variable .

```
public void putDataToTable(java.lang.String restaurantname) {
        RestaurantName = restaurantname;
        java.util.Vector customerList =
getCustomerReviewByName(restaurantname);
        Iterator j=customerList.iterator();
        while (j.hasNext()) {
                CustomerreviewDetail ci =
(CustomerreviewDetail)j.next();
                String[] str = {ci.getCustomername(),ci.getReview()
                };
                TableModel.addRow(str);
        }
}
```

3. The CustomerReviewTable.getCustomerReviewByName method instantiates a client (if required) and calls its getCustomerreviewsByRestaurant method, passing it the name of the selected restaurant.

```
private Vector getCustomerReviewByName(java.lang.String restaurantname) {
       Vector custList = new Vector();
       try {
              WebService.DGWebServiceClientGenClient.DGWebService service2 =
new
              WebService.DGWebServiceClientGenClient.DGWebService_Impl();

WebService.DGWebServiceClientGenClient.DGWebServiceServantInterface port =
service2.getDGWebServiceServantInterfacePort();

              custList =
(java.util.Vector)port.getCustomerreviewsByRestaurant(restaurantname);}
       catch (Exception ex) {
              System.err.println("Caught an exception." );
              ex.printStackTrace();
       }
       return custList;
}
```

4. The review data is passed up to the CustomerReviewTable.putDataToTable method, which iterates through it and populates the table.

```
public void putDataToTable(java.lang.String restaurantname) {
       RestaurantName = restaurantname;
       java.util.Vector customerList =
getCustomerReviewByName(restaurantname);
       Iterator j=customerList.iterator();
       while (j.hasNext()) {
              CustomerreviewDetail ci =
(CustomerreviewDetail)j.next();
              String[] str = {ci.getCustomername(),ci.getReview()
              };
              TableModel.addRow(str);
       }
}
```

5. The RestaurantTable.jButton1ActionPerformed method then displays the data.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent
evt)
{
	int r =jTable1.getSelectedRow();
	int c = jTable1.getSelectedColumnCount();
	String i =(String)TableModel.getValueAt(r,0);
	CustomerReviewTable crt = new CustomerReviewTable();
	crt.putDataToTable(i);
	crt.show();
	System.out.println(i);
}
```

Creating a New Customer Review Record

When the user types a name and review comments on the Customer Review window and clicks the Submit Customer Review button, the CustomerReviewTable's jButton1ActionPerformed method creates the review record in the database by means of the client's methods, then refreshes the Customer Review window, as follows:

1. When the CustomerReviewTable's button is pressed to submit a customer review record, the CustomerReviewTable.jButton1ActionPerformed method instantiates a new client (if required) and calls its createCustomerreview method, passing it the restaurant name, the customer name, and the review data.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        try {
                WebService.DGWebServiceClientGenClient.DGWebService service1 =
new
                WebService.DGWebServiceClientGenClient.DGWebService_Impl();

WebService.DGWebServiceClientGenClient.DGWebServiceServantInterface port =
service1.getDGWebServiceServantInterfacePort();

                port.createCustomerreview(RestaurantName,
                customerNameField.getText(),reviewField.getText());
        }
        catch (Exception ex) {
                System.err.println("Caught an exception." );
                ex.printStackTrace();
        }
        refreshView();
     }
```

2. This same method (jButton1ActionPerformed) calls the CustomerReviewTable.refreshView method.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        try {
                WebService.DGWebServiceClientGenClient.DGWebService service1 =
new
                WebService.DGWebServiceClientGenClient.DGWebService_Impl();

WebService.DGWebServiceClientGenClient.DGWebServiceServantInterface port =
service1.getDGWebServiceServantInterfacePort();

                port.createCustomerreview(RestaurantName,
                customerNameField.getText(),reviewField.getText());
        }
        catch (Exception ex) {
                System.err.println("Caught an exception." );
                ex.printStackTrace();
        }
        refreshView();
     }
```

3. The CustomerReviewTable.refreshView method calls the putDataToTable method, passing it the restaurant name.

```
void refreshView() {
	try{
		while(TableModel.getRowCount()>0) {
		    TableModel.removeRow(0);
		}
		putDataToTable(RestaurantName);
		repaint();
	}
	catch (Exception ex) {
		ex.printStackTrace();
	}
}
```

4. The CustomerReviewTable.putDataToTable method populates the table.

```
public void putDataToTable(java.lang.String restaurantname) {
	RestaurantName = restaurantname;
	java.util.Vector customerList =
getCustomerReviewByName(restaurantname);
	Iterator j=customerList.iterator();
	while (j.hasNext()) {
		CustomerreviewDetail ci =
(CustomerreviewDetail)j.next();
		String[] str = {ci.getCustomername(),ci.getReview()
		};
		TableModel.addRow(str);
	}
}
```

5. Then the CustomerReviewTable.refreshView method repaints the window, showing the new data.

```
void refreshView() {
       try{
              while(TableModel.getRowCount()>0) {
                  TableModel.removeRow(0);
              }
              putDataToTable(RestaurantName);
              repaint();
       }
       catch (Exception ex) {
              ex.printStackTrace();
       }
}
```

APPENDIX **A**

DiningGuide Source Files

This appendix displays the code for the following DiningGuide components:

- EJB tier components:
  - “RestaurantBean.java Source” on page 146
  - “RestaurantDetail.java Source” on page 149
  - “CustomerreviewBean.java Source” on page 154
  - “CustomerreviewDetail.java Source” on page 157
  - “DiningGuideManagerBean.java Source” on page 160
- Client components:
  - “RestaurantTable.java Source” on page 164
  - “CustomerReviewTable.java Source” on page 168

This code is also available as source files within the DiningGuide application zip file, which you can download from the Forte for Java Developer Resources portal at:

`http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

Tip – If you use these files to cut and paste code into the Sun ONE Studio 5 Source Editor, all formatting is lost. To automatically reformat the code in the Source Editor, press Control-Shift F after you paste the code.

Note – Wrapped lines are copied from PDF as separate lines, causing the compiler to interpret certain types of statements as nonesense. These lines are preceeded in this code by the comment, “Join the following two lines in the Source Editor.” Also, the Source Editor will mark such lines as coding errors, so they are easy to find and correct.

Solaris and Linux users are advised not to copy these files, because the Source Editor does not read the carriage returns at the ends of lines. To view source files, unzip the DiningGuide source zip file and then mount the unzipped directory in the IDE.

RestaurantBean.java Source

```
package Data;

import javax.ejb.*;


public abstract class RestaurantBean implements javax.ejb.EntityBean {

    private javax.ejb.EntityContext context;

    /**
     * @see javax.ejb.EntityBean#setEntityContext(javax.ejb.EntityContext)
     */
    public void setEntityContext(javax.ejb.EntityContext aContext) {
        context=aContext;
    }

    /**
     * @see javax.ejb.EntityBean#ejbActivate()
     */
    public void ejbActivate() {

    }

    /**
     * @see javax.ejb.EntityBean#ejbPassivate()
     */
    public void ejbPassivate() {

    }

    /**
     * @see javax.ejb.EntityBean#ejbRemove()
     */
    public void ejbRemove() {

    }
```

```
    /**
     * @see javax.ejb.EntityBean#unsetEntityContext()
     */
    public void unsetEntityContext() {
        context=null;
    }

    /**
     * @see javax.ejb.EntityBean#ejbLoad()
     */
    public void ejbLoad() {

    }

    /**
     * @see javax.ejb.EntityBean#ejbStore()
     */
    public void ejbStore() {

    }

    public abstract java.lang.String getRestaurantname();
    public abstract void setRestaurantname(java.lang.String restaurantname);

    public abstract java.lang.String getCuisine();
    public abstract void setCuisine(java.lang.String cuisine);

    public abstract java.lang.String getNeighborhood();
    public abstract void setNeighborhood(java.lang.String neighborhood);

    public abstract java.lang.String getAddress();
    public abstract void setAddress(java.lang.String address);

    public abstract java.lang.String getPhone();
    public abstract void setPhone(java.lang.String phone);

    public abstract java.lang.String getDescription();
    public abstract void setDescription(java.lang.String description);

    public abstract int getRating();
    public abstract void setRating(int rating);
```

```
    public java.lang.String ejbCreate(java.lang.String restaurantname,
java.lang.String cuisine, java.lang.String neighborhood, java.lang.String
address, java.lang.String phone, java.lang.String description, int rating)
throws javax.ejb.CreateException {
        if (restaurantname == null) {
            // Join the following two lines in the Source Editor
            throw new javax.ejb.CreateException("The restaurant name is
required.");
        }
        setRestaurantname(restaurantname);
        setCuisine(cuisine);
        setNeighborhood(neighborhood);
        setAddress(address);
        setPhone(phone);
        setDescription(description);
        setRating(rating);
        return null;
    }

   public void ejbPostCreate(java.lang.String restaurantname, java.lang.String
cuisine, java.lang.String neighborhood, java.lang.String address,
java.lang.String phone, java.lang.String description, int rating) throws
javax.ejb.CreateException {
    }

    public Data.RestaurantDetail getRestaurantDetail() {
        return (new RestaurantDetail(getRestaurantname(),
        getCuisine(), getNeighborhood(), getAddress(), getPhone(),
        getDescription(), getRating()));
    }
}
```

RestaurantDetail.java Source

```
/*
 * RestaurantDetail.java
 *
 * Created on March 27, 2003, 3:35 PM
 */

package Data;

import java.beans.*;

public class RestaurantDetail extends Object implements java.io.Serializable {

    private static final String PROP_SAMPLE_PROPERTY = "SampleProperty";

    private String sampleProperty;

    private PropertyChangeSupport propertySupport;

    /** Holds value of property restaurantname. */
    private String restaurantname;

    /** Holds value of property cuisine. */
    private String cuisine;

    /** Holds value of property neighborhood. */
    private String neighborhood;

    /** Holds value of property address. */
    private String address;

    /** Holds value of property phone. */
    private String phone;

    /** Holds value of property description. */
    private String description;

    /** Holds value of property rating. */
```

```
    private int rating;

    /** Creates new RestaurantDetail */
    public RestaurantDetail() {
        propertySupport = new PropertyChangeSupport( this );
    }

    public RestaurantDetail(java.lang.String restaurantname, java.lang.String
cuisine, java.lang.String neighborhood, java.lang.String address,
java.lang.String phone, java.lang.String description, int rating) {
        System.out.println("Creating new RestaurantDetail");
        setRestaurantname(restaurantname);
        setCuisine(cuisine);
        setNeighborhood(neighborhood);
        setAddress(address);
        setPhone(phone);
        setDescription(description);
        setRating(rating);
    }

    public String getSampleProperty() {
        return sampleProperty;
    }

    public void setSampleProperty(String value) {
        String oldValue = sampleProperty;
        sampleProperty = value;
        propertySupport.firePropertyChange(PROP_SAMPLE_PROPERTY, oldValue,
sampleProperty);
    }


    public void addPropertyChangeListener(PropertyChangeListener listener) {
        propertySupport.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener) {
        propertySupport.removePropertyChangeListener(listener);
    }

    /** Getter for property restaurantname.
     * @return Value of property restaurantname.
```

```
     *
     */
    public String getRestaurantname() {
        return this.restaurantname;
    }

    /** Setter for property restaurantname.
     * @param restaurantname New value of property restaurantname.
     *
     */
    public void setRestaurantname(String restaurantname) {
        this.restaurantname = restaurantname;
    }

    /** Getter for property cuisine.
     * @return Value of property cuisine.
     *
     */
    public String getCuisine() {
        return this.cuisine;
    }

    /** Setter for property cuisine.
     * @param cuisine New value of property cuisine.
     *
     */
    public void setCuisine(String cuisine) {
        this.cuisine = cuisine;
    }

    /** Getter for property neighborhood.
     * @return Value of property neighborhood.
     *
     */
    public String getNeighborhood() {
        return this.neighborhood;
    }

    /** Setter for property neighborhood.
     * @param neighborhood New value of property neighborhood.
     *
```

```
     */
    public void setNeighborhood(String neighborhood) {
        this.neighborhood = neighborhood;
    }

    /** Getter for property address.
     * @return Value of property address.
     *
     */
    public String getAddress() {
        return this.address;
    }

    /** Setter for property address.
     * @param address New value of property address.
     *
     */
    public void setAddress(String address) {
        this.address = address;
    }

    /** Getter for property phone.
     * @return Value of property phone.
     *
     */
    public String getPhone() {
        return this.phone;
    }

    /** Setter for property phone.
     * @param phone New value of property phone.
     *
     */
    public void setPhone(String phone) {
        this.phone = phone;
    }

    /** Getter for property description.
     * @return Value of property description.
     *
     */
```

```
    public String getDescription() {
        return this.description;
    }

    /** Setter for property description.
     * @param description New value of property description.
     *
     */
    public void setDescription(String description) {
        this.description = description;
    }

    /** Getter for property rating.
     * @return Value of property rating.
     *
     */
    public int getRating() {
        return this.rating;
    }

    /** Setter for property rating.
     * @param rating New value of property rating.
     *
     */
    public void setRating(int rating) {
        this.rating = rating;
    }
}
```

CustomerreviewBean.java Source

```
package Data;

import javax.ejb.*;

public abstract class CustomerreviewBean implements javax.ejb.EntityBean {

    private javax.ejb.EntityContext context;


    /**
     * @see javax.ejb.EntityBean#setEntityContext(javax.ejb.EntityContext)
     */
    public void setEntityContext(javax.ejb.EntityContext aContext) {
        context=aContext;
    }

    /**
     * @see javax.ejb.EntityBean#ejbActivate()
     */
    public void ejbActivate() {

    }

    /**
     * @see javax.ejb.EntityBean#ejbPassivate()
     */
    public void ejbPassivate() {

    }

    /**
     * @see javax.ejb.EntityBean#ejbRemove()
     */
    public void ejbRemove() {

    }
```

```
    /**
     * @see javax.ejb.EntityBean#unsetEntityContext()
     */
    public void unsetEntityContext() {
        context=null;
    }

    /**
     * @see javax.ejb.EntityBean#ejbLoad()
     */
    public void ejbLoad() {

    }

    /**
     * @see javax.ejb.EntityBean#ejbStore()
     */
    public void ejbStore() {

    }

    public abstract java.lang.String getRestaurantname();
    public abstract void setRestaurantname(java.lang.String restaurantname);

    public abstract java.lang.String getCustomername();
    public abstract void setCustomername(java.lang.String customername);

    public abstract java.lang.String getReview();
    public abstract void setReview(java.lang.String review);

    public Data.CustomerreviewKey ejbCreate(java.lang.String restaurantname,
java.lang.String customername, java.lang.String review) throws
javax.ejb.CreateException {
        if ((restaurantname == null) || (customername == null)) {
            // Join the following two lines in the Source Editor
            throw new javax.ejb.CreateException("Both the restaurant name and
customer name are required.");
        }
        setRestaurantname(restaurantname);
        setCustomername(customername);
        setReview(review);
        return null;
```

```
    }

   public void ejbPostCreate(java.lang.String restaurantname, java.lang.String
customername, java.lang.String review) throws javax.ejb.CreateException {
    }

    public Data.CustomerreviewDetail getCustomerreviewDetail() {
        return (new CustomerreviewDetail(getRestaurantname(),
        getCustomername(), getReview()));
    }
}
```

CustomerreviewDetail.java Source

```
 * CustomerreviewDetail.java
 *
 * Created on March 27, 2003, 3:35 PM
 */

package Data;

import java.beans.*;

public class CustomerreviewDetail extends Object implements
java.io.Serializable {

    private static final String PROP_SAMPLE_PROPERTY = "SampleProperty";

    private String sampleProperty;

    private PropertyChangeSupport propertySupport;

    /** Holds value of property restaurantname. */
    private String restaurantname;

    /** Holds value of property customername. */
    private String customername;

    /** Holds value of property review. */
    private String review;

    /** Creates new CustomerreviewDetail */
    public CustomerreviewDetail() {
        propertySupport = new PropertyChangeSupport( this );
    }

    public CustomerreviewDetail(java.lang.String restaurantname,
java.lang.String customername, java.lang.String review) {
        System.out.println("Creating new CustomerreviewDetail");
        setRestaurantname(restaurantname);
        setCustomername(customername);
```

```
        setReview(review);
    }

    public String getSampleProperty() {
        return sampleProperty;
    }

    public void setSampleProperty(String value) {
        String oldValue = sampleProperty;
        sampleProperty = value;
        propertySupport.firePropertyChange(PROP_SAMPLE_PROPERTY, oldValue,
sampleProperty);
    }

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        propertySupport.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener) {
        propertySupport.removePropertyChangeListener(listener);
    }

    /** Getter for property restaurantname.
     * @return Value of property restaurantname.
     *
     */
    public String getRestaurantname() {
        return this.restaurantname;
    }

    /** Setter for property restaurantname.
     * @param restaurantname New value of property restaurantname.
     *
     */
    public void setRestaurantname(String restaurantname) {
        this.restaurantname = restaurantname;
    }

    /** Getter for property customername.
     * @return Value of property customername.
     *
```

```
     */
    public String getCustomername() {
        return this.customername;
    }

    /** Setter for property customername.
     * @param customername New value of property customername.
     *
     */
    public void setCustomername(String customername) {
        this.customername = customername;
    }

    /** Getter for property review.
     * @return Value of property review.
     *
     */
    public String getReview() {
        return this.review;
    }

    /** Setter for property review.
     * @param review New value of property review.
     *
     */
    public void setReview(String review) {
        this.review = review;
    }
}
```

DiningGuideManagerBean.java Source

```
package Data;

import javax.ejb.*;
import javax.naming.*;

public class DiningGuideManagerBean implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext context;
    private RestaurantHome myRestaurantHome;
    private CustomerreviewHome myCustomerreviewHome;

    /**
     * @see javax.ejb.SessionBean#setSessionContext(javax.ejb.SessionContext)
     */
    public void setSessionContext(javax.ejb.SessionContext aContext) {
        context=aContext;
    }

    /**
     * @see javax.ejb.SessionBean#ejbActivate()
     */
    public void ejbActivate() {

    }

    /**
     * @see javax.ejb.SessionBean#ejbPassivate()
     */
    public void ejbPassivate() {

    }

    /**
     * @see javax.ejb.SessionBean#ejbRemove()
     */
    public void ejbRemove() {

    }
```

```
    /**
     * See section 7.10.3 of the EJB 2.0 specification
     */
    public void ejbCreate() {
        System.out.println("Entering DiningGuideManagerEJB.ejbCreate()");
        Context c = null;
        Object result = null;
        if (this.myRestaurantHome == null) {
            try {
                c = new InitialContext();
                result = c.lookup("java:comp/env/ejb/Restaurant");
                myRestaurantHome =
(RestaurantHome)javax.rmi.PortableRemoteObject.narrow(result,
RestaurantHome.class);
            }
            catch (Exception e) {System.out.println("Error: "+
            e); }
        }
        Context crc = null;
        Object crresult = null;
        if (this.myCustomerreviewHome == null) {
            try {
                crc = new InitialContext();
                result = crc.lookup("java:comp/env/ejb/Customerreview");
                myCustomerreviewHome =
(CustomerreviewHome)javax.rmi.PortableRemoteObject.narrow(result,
CustomerreviewHome.class);
            }
            catch (Exception e) {System.out.println("Error: "+
            e); }
        }

    }

    public java.util.Vector getAllRestaurants() {
        System.out.println("Entering
DiningGuideManagerEJB.getAllRestaurants()");
        java.util.Vector restaurantList = new java.util.Vector();
        try {
            java.util.Collection rl = myRestaurantHome.findAll();
            if (rl == null) { restaurantList = null; }
```

```
            else {
                RestaurantDetail rd;
                java.util.Iterator rli = rl.iterator();
                while ( rli.hasNext() ) {
                    rd = ((Restaurant)rli.next()).getRestaurantDetail();
                    System.out.println(rd.getRestaurantname());
                    System.out.println(rd.getRating());
                    restaurantList.addElement(rd);
                }
            }
        }
        catch (Exception e) {
            // Join the following two lines in the Source Editor
            System.out.println("Error in
DiningGuideManagerEJB.getAllRestaurants(): " + e);
        }
        // Join the following two lines in the Source Editor
       System.out.println("Leaving DiningGuideManagerEJB.getAllRestaurants()");
        return restaurantList;
    }

    public java.util.Vector getCustomerreviewsByRestaurant(java.lang.String
restaurantname) {
        System.out.println("Entering
DiningGuideManagerEJB.getCustomerreviewsByRestaurant()");
        java.util.Vector reviewList = new java.util.Vector();
        try {
            java.util.Collection rl =
myCustomerreviewHome.findByRestaurantName(restaurantname);
            if (rl == null) { reviewList = null; }
            else {
                CustomerreviewDetail crd;
                java.util.Iterator rli = rl.iterator();
                while ( rli.hasNext() ) {
                 crd = ((Customerreview)rli.next()).getCustomerreviewDetail();
                    System.out.println(crd.getRestaurantname());
                    System.out.println(crd.getCustomername());
                    System.out.println(crd.getReview());
                    reviewList.addElement(crd);
                }
            }
        }
```

```
        catch (Exception e) {
            // Join the following two lines in the Source Editor
            System.out.println("Error in
DiningGuideManagerEJB.getCustomerreviewsByRestaurant(): " + e);
        }
        // Join the following two lines in the Source Editor
        System.out.println("Leaving
DiningGuideManagerEJB.getCustomerreviewsByRestaurant()");
        return reviewList;
    }

    public void createCustomerreview(java.lang.String restaurantname,
java.lang.String customername, java.lang.String review) {
        System.out.println("Entering
DiningGuideManagerEJB.createCustomerreview()");
        try {
            Customerreview customerrev =
myCustomerreviewHome.create(restaurantname, customername,
            review);
        } catch (Exception e) {
            // Join the following two lines in the Source Editor
            System.out.println("Error in
DiningGuideManagerEJB.createCustomerreview(): " + e);
        }
        // Join the following two lines in the Source Editor
        System.out.println("Leaving
DiningGuideManagerEJB.createCustomerreview()");
    }

    public Data.RestaurantDetail getRestaurantDetail() {
        return null;
    }

    public Data.CustomerreviewDetail getCustomerreviewDetail() {
        return null;
    }
}
```

RestaurantTable.java Source

```
/*
 * RestaurantTable.java
 *
 * Created on March 12, 2003, 4:29 PM
 */

package Client;
import javax.swing.table.*;
import java.util.*;
import WebService.DGWebServiceClientGenClient.*;
/**
 *
 * @author administrator
 */
public class RestaurantTable extends javax.swing.JFrame {
    /** Creates new form RestaurantTable */
    public RestaurantTable() {
        initComponents();
        restaurantList=getAllRestaurants();
        putDataToTable();
    }
    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    private void initComponents() {//GEN-BEGIN:initComponents
        jButton1 = new javax.swing.JButton();
        jScrollPane1 = new javax.swing.JScrollPane();
        jTable1 = new javax.swing.JTable();
        jLabel1 = new javax.swing.JLabel();
        getContentPane().setLayout(new
        org.netbeans.lib.awtextra.AbsoluteLayout());
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
```

```
        });
        jButton1.setText("View Customer Comments");
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });
        getContentPane().add(jButton1, new
        org.netbeans.lib.awtextra.AbsoluteConstraints(200, 240, -1, -1));
        TableModel = (new javax.swing.table.DefaultTableModel(
        new Object [][] {
        },
        new String [] {
            "RESTAURANT NAME", "CUISINE", "NEIGHBORHOOD", "ADDRESS", "PHONE",
            "DESCRIPTION", "RATING"
        }
        ) {
            Class[] types = new Class [] {
                java.lang.String.class, java.lang.String.class,
                java.lang.String.class,

java.lang.String.class,java.lang.String.class,java.lang.String.class,java.lang
                .String.class
            };
            public Class getColumnClass(int columnIndex) {
                return types [columnIndex];
            }
        });
        jTable1.setModel(TableModel);
        jScrollPane1.setViewportView(jTable1);
        getContentPane().add(jScrollPane1, new
        org.netbeans.lib.awtextra.AbsoluteConstraints(0, 60, 600, 100));
        jLabel1.setText("Restaurant Listing");
        getContentPane().add(jLabel1, new
        org.netbeans.lib.awtextra.AbsoluteConstraints(230, 20, 110, 30));
        pack();
    }//GEN-END:initComponents
   private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_jButton1ActionPerformed
        int r =jTable1.getSelectedRow();
        int c = jTable1.getSelectedColumnCount();
```

```
        String i =(String)TableModel.getValueAt(r,0);
        CustomerReviewTable crt = new CustomerReviewTable();
        crt.putDataToTable(i);
        crt.show();
        System.out.println(i);
    }//GEN-LAST:event_jButton1ActionPerformed
    /** Exit the Application */
    private void exitForm(java.awt.event.WindowEvent evt) {//GEN-
FIRST:event_exitForm
        System.exit(0);
    }//GEN-LAST:event_exitForm
    private void putDataToTable() {
        Iterator j=restaurantList.iterator();
        while (j.hasNext()) {
            RestaurantDetail ci = (RestaurantDetail)j.next();
            String strRating = null;
            String[] str =

{ci.getRestaurantname(),ci.getCuisine(),ci.getNeighborhood(),ci.getAddress(),
             ci.getPhone(),ci.getDescription(),
             strRating.valueOf(ci.getRating()),
            };
            TableModel.addRow(str);
        }
    }
    private Vector getAllRestaurants() {
        Vector restList = new Vector();
        try {
            WebService.DGWebServiceClientGenClient.DGWebService service1 = new
            WebService.DGWebServiceClientGenClient.DGWebService_Impl();
          WebService.DGWebServiceClientGenClient.DGWebServiceServantInterface
port
            =
            service1.getDGWebServiceServantInterfacePort();
            restList = (java.util.Vector)port.getAllRestaurants();
        }
        catch (Exception ex) {
            System.err.println("Caught an exception." );
            ex.printStackTrace();
        }
        return restList;
    }
```

```
    private Vector getCustomerreviewByRestaurant(java.lang.String
    restaurantname) {
        Vector reviewList = new Vector();
        try {
            WebService.DGWebServiceClientGenClient.DGWebService service2 = new
            WebService.DGWebServiceClientGenClient.DGWebService_Impl();
          WebService.DGWebServiceClientGenClient.DGWebServiceServantInterface
port
            =
            service2.getDGWebServiceServantInterfacePort();
            reviewList =

(java.util.Vector)port.getCustomerreviewsByRestaurant(restaurantname);
        }
        catch (Exception ex) {
            System.err.println("Caught an exception." );
            ex.printStackTrace();
        }
        return reviewList;
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        new RestaurantTable().show();
    }
    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JButton jButton1;
    private javax.swing.JScrollPane jScrollPane1;
    private javax.swing.JTable jTable1;
    private javax.swing.JLabel jLabel1;
    // End of variables declaration//GEN-END:variables
    private DefaultTableModel TableModel;
    private java.util.Vector restaurantList = null;
}
```

CustomerReviewTable.java Source

```
/*
 * CustomerReviewTable.java
 *
 * Created on March 12, 2003, 4:29 PM
 */

package Client;
import javax.swing.table.*;
import java.util.*;
import WebService.DGWebServiceClientGenClient.*;
/**
 *
 ** @author administrator
 */
public class CustomerReviewTable extends javax.swing.JFrame {
    /** Creates new form CustomerReviewTable */
    public CustomerReviewTable() {
        initComponents();
    }
    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    private void initComponents() {//GEN-BEGIN:initComponents
        jScrollPane1 = new javax.swing.JScrollPane();
        jTable1 = new javax.swing.JTable();
        jButton1 = new javax.swing.JButton();
        customerNameLabel = new javax.swing.JLabel();
        customerNameField = new javax.swing.JTextField();
        reviewLabel = new javax.swing.JLabel();
        reviewField = new javax.swing.JTextField();
        jLabel1 = new javax.swing.JLabel();
        getContentPane().setLayout(new
        org.netbeans.lib.awtextra.AbsoluteLayout());
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
```

```
                exitForm(evt);
            }
        });
        TableModel = (new javax.swing.table.DefaultTableModel(
        new Object [][] {
        },
        new String [] {
            "CUSTOMER NAME", "REVIEW"
        }
        ) {
            Class[] types = new Class [] {
                java.lang.String.class,java.lang.String.class
            };
            public Class getColumnClass(int columnIndex) {
                return types [columnIndex];
            }
        });
        jTable1.setModel(TableModel);
        jScrollPane1.setViewportView(jTable1);
        getContentPane().add(jScrollPane1, new
        org.netbeans.lib.awtextra.AbsoluteConstraints(0, 60, 400, 100));
        jButton1.setText("Submit Customer Review");
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });
        getContentPane().add(jButton1, new
        org.netbeans.lib.awtextra.AbsoluteConstraints(100, 250, 190, -1));
        customerNameLabel.setText("Customer Name");
        getContentPane().add(customerNameLabel, new
        org.netbeans.lib.awtextra.AbsoluteConstraints(40, 170, -1, -1));
        getContentPane().add(customerNameField, new
        org.netbeans.lib.awtextra.AbsoluteConstraints(153, 170, 170, -1));
        reviewLabel.setText("Review");
        getContentPane().add(reviewLabel, new
        org.netbeans.lib.awtextra.AbsoluteConstraints(40, 200, 80, -1));
        getContentPane().add(reviewField, new
        org.netbeans.lib.awtextra.AbsoluteConstraints(153, 200, 170, 20));
        jLabel1.setText("All Customer Review By Restaurant Name");
        getContentPane().add(jLabel1, new
```

```
        org.netbeans.lib.awtextra.AbsoluteConstraints(80, 10, 240, -1));
        pack();
    }//GEN-END:initComponents
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{//GENFIRST:event_jButton1ActionPerformed
        try {
            WebService.DGWebServiceClientGenClient.DGWebService service1 = new
            WebService.DGWebServiceClientGenClient.DGWebService_Impl();
           WebService.DGWebServiceClientGenClient.DGWebServiceServantInterface
            port =
            service1.getDGWebServiceServantInterfacePort();
            port.createCustomerreview(RestaurantName,
            customerNameField.getText(),reviewField.getText());
        }
        catch (Exception ex) {
            System.err.println("Caught an exception." );
            ex.printStackTrace();
        }
        refreshView();
    }//GEN-LAST:event_jButton1ActionPerformed
    void refreshView() {
        try{
            while(TableModel.getRowCount()>0) {
                TableModel.removeRow(0);
            }
            putDataToTable(RestaurantName);
            repaint();
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    /** Exit the Application */
    private void exitForm(java.awt.event.WindowEvent evt)
{//GENFIRST:event_exitForm
        System.exit(0);
    }//GEN-LAST:event_exitForm
    public void putDataToTable(java.lang.String restaurantname) {
        RestaurantName = restaurantname;
       java.util.Vector customerList =getCustomerReviewByName(restaurantname);
        Iterator j=customerList.iterator();
```

```
        while (j.hasNext()) {
            CustomerreviewDetail ci = (CustomerreviewDetail)j.next();
            String[] str = {ci.getCustomername(),ci.getReview()
            };
            TableModel.addRow(str);
        }
    }
    private Vector getCustomerReviewByName(java.lang.String restaurantname) {
        Vector custList = new Vector();
        try {
            WebService.DGWebServiceClientGenClient.DGWebService service2 = new
            WebService.DGWebServiceClientGenClient.DGWebService_Impl();
           WebService.DGWebServiceClientGenClient.DGWebServiceServantInterface
            port =
            service2.getDGWebServiceServantInterfacePort();
            custList =

(java.util.Vector)port.getCustomerreviewsByRestaurant(restaurantname);
        }
        catch (Exception ex) {
            System.err.println("Caught an exception." );
            ex.printStackTrace();
        }
        return custList;
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        new CustomerReviewTable().show();
    }
    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JLabel reviewLabel;
    private javax.swing.JButton jButton1;
    private javax.swing.JScrollPane jScrollPane1;
    private javax.swing.JTextField customerNameField;
    private javax.swing.JTable jTable1;
    private javax.swing.JLabel customerNameLabel;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JTextField reviewField;
    // End of variables declaration//GEN-END:variables
```

```
    private DefaultTableModel TableModel;
    private java.lang.String RestaurantName = null;
    //private java.util.Vector restaurantList = null;
}
```

APPENDIX **B**

DiningGuide Database Script

This appendix displays the following database scripts for the DiningGuide tutorial:

- "Script for a PointBase Database" on page 174
- "Script for an Oracle Database" on page 175

Script for a PointBase Database

```
drop table CustomerReview;
drop table Restaurant;

create table Restaurant(
      restaurantName varchar(80),
      cuisine        varchar(25),
      neighborhood   varchar(25),
      address        varchar(30),
      phone          varchar(12),
      description    varchar(200),
      rating         tinyint,
constraint pk_Restaurant primary key(restaurantName));

create table CustomerReview(
      restaurantName varchar(80) not null references
Restaurant(restaurantName),
      customerName   varchar(25),
      review         varchar(200),
constraint pk_CustomerReview primary key(CustomerName, restaurantName));

insert into Restaurant (restaurantName, cuisine, neighborhood, address, phone,
description, rating) values ('French Lemon','Mediterranean','Rockridge','1200
College Avenue','510 888 8888','Very nice spot.',5);
insert into Restaurant (restaurantName, cuisine, neighborhood, address, phone,
description, rating) values ('Bay Fox','Mediterranean','Piedmont','1200
Piedmont Avenue','510 888 8888','Excellent.',5);

insert into CustomerReview (restaurantName, customerName, review) values
('French Lemon','Fred','Nice flowers.');
insert into CustomerReview (restaurantName, customerName, review) values
('French Lemon','Ralph','Excellent service.');
```

Script for an Oracle Database

```
drop table CustomerReview;
drop table Restaurant;

create table Restaurant(
       restaurantName varchar(80),
       cuisine        varchar(25),
       neighborhood   varchar(25),
       address        varchar(30),
       phone          varchar(12),
       description    varchar(200),
       rating         number(1,0),
constraint pk_Restaurant primary key(restaurantName));
grant all on Restaurant to public;

create table CustomerReview(
       restaurantName varchar(80) not null references
Restaurant(restaurantName),
       customerName   varchar(25),
       review         varchar(200),
constraint pk_CustomerReview primary key(CustomerName, restaurantName));
grant all on CustomerReview to public;

insert into Restaurant (restaurantName, cuisine, neighborhood, address, phone,
description, rating) values ('French Lemon','Mediterranean','Rockridge','1200
College Avenue','510 888 8888','Very nice spot.',5);
insert into Restaurant (restaurantName, cuisine, neighborhood, address, phone,
description, rating) values ('Bay Fox','Mediterranean','Piedmont','1200
Piedmont Avenue','510 888 8888','Excellent.',5);

insert into CustomerReview (restaurantName, customerName, review) values
('French Lemon','Fred','Nice flowers.');
insert into CustomerReview (restaurantName, customerName, review) values
('French Lemon','Ralph','Excellent service.');

commit;
```

APPENDIX **C**

Creating the Tutorial with an Oracle Database

This appendix describes the steps you must perform to create and run the DiningGuide tutorial with an Oracle database. The topics covered are:

- “Setting up Database Connectivity with the Oracle Database,” which follows
- “Creating the Database Tables” on page 183
- “Creating EJB Components with an Oracle Database” on page 185
- “Creating the Web Service with an Oracle Database” on page 186

Note – There are several references in this book to the *DiningGuide application files.* These files include a completed version of the tutorial application, a readme file describing how to run the completed application, and SQL script files for creating the required database tables. These files are compressed into a zip file you can download from the Sun ONE Studio 5 Developer Resources portal at `http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

Setting up Database Connectivity with the Oracle Database

Configure Sun ONE Application Server 7 to connect to the Oracle database by performing the required JDBC-related actions in the application server environment. These include:

- Enabling the database’s JDBC driver
- Creating a connection pool
- Creating a JDBC data source
- Creating a JDBC persistent manager

Enabling the Oracle Type 4 JDBC Driver

Enabling a JDBC driver means putting the driver library in the Sun ONE Studio 5 and Sun ONE Application Server 7 class paths. To do this, you need the Oracle Type 4 JDBC driver library (the `classes12.zip` file). You can download this driver from the Oracle portal. Copy the JDBC Type 4 driver into the program files of Sun ONE Studio 5 before you start the IDE.

To enable the Oracle Type 4 JDBC driver:

1. **Copy the Oracle Type 4 driver library to the *s1studio-install-directory*`/lib/ext` directory.**

   For example, copy the `classes12.zip` file to `c:\Sun\studio5_se\lib\ext`.

Note – You must have root or administrator privileges to write to the Sun ONE Studio 5 home directories.

---

2. **Restart the IDE.**
3. **In the Runtime pane of the Explorer, select your application server instance.**

   It is labeled *app-server-name* (*app-server-host*:*app-server-port*). For example, the default server is server1 (localhost:4848), or a standard user's server could be MyServer (localhost:4855).

4. **Display the properties of the application server instance.**

   The property window is usually below the Explorer window. Selecting the node displays the properties in the window. If the window is not there, right-click the server instance node and choose Properties.

5. **Open the property editor for the Classpath Suffix property.**

   Click on the value field of this property, then on the ellipsis button that appears. The Classpath Suffix editor window is displayed.

6. **Click the Add JAR/ZIP button.**

   Use the Add JAR File file finder to locate your `classes12.zip` file.

7. **Select the `classes12.zip` file and click OK.**
8. **Click OK to close the property editor window.** J

Connecting the IDE to the Oracle Server

To create the JDBC connectivity resources or the EJB tier of the tutorial, you must connect the IDE to the Oracle database. You can either connect before creating these components or during the creation process. Here is how you connect to the database beforehand:

1. **Make sure the Oracle server is running.**
2. **In the Runtime pane of the Explorer, expand the Databases node and its Drivers subnode.**

   A node labeled Oracle thin is displayed.

   If this node has a red strike across it, you have not enabled the Oracle JDBC driver properly. Follow the procedures in “Enabling the Oracle Type 4 JDBC Driver” on page 178.
3. **Right-click this node and choose Connect Using.**

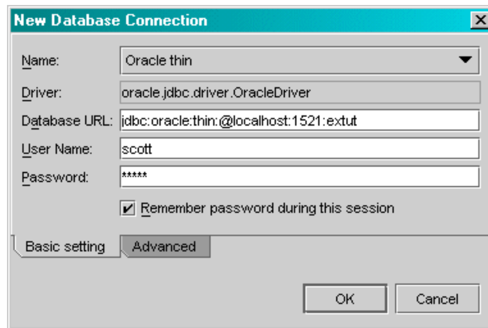
   The New Database Connection dialog box is displayed.
4. **Make sure Oracle thin is selected in the name field.**
5. **Fill in the property values for Database URL, User, and Password.**

   For example, the following values are correct for a locally installed Oracle database with a SID of “extut,” and the default Oracle login of “scott” for User and “tiger” for Password: (1521 is the standard Oracle port number.)

Name	Value
URL	jdbc:oracle:thin:@localhost:1521:extut
User	scott
Password	tiger

6. **Enable the Remember password during this session option.**

The New Database Connection dialog box should look like this:



7. **Click OK.**
8. **Close the** `Drivers` **node.**

   The new Oracle thin driver node is displayed, labeled jdbc:oracle:thin:@*hostname*:1521:*sid* [*Username* on *Password*].

9. **Expand this node and its Tables subnode.**

   The tables in the database, including the RESTAURANT and CUSTOMERREVIEW tables, are displayed.

Creating a JDBC Connection Pool

To create a JDBC connection pool so that the business objects in the system can share database access, you first define a JDBC connection pool with information related to your database, then register it with Sun ONE Application Server 7.

Note – Before starting this procedure, make sure both the admin server and the application server are running (refer to "Starting the Software" on page 23).

To create an Oracle JDBC connection pool for this tutorial:

1. **In the Runtime pane of the Explorer, expand the Server Registry, Installed Servers, and Sun ONE Application Server 7 nodes.**
2. **Right-click the Unregistered JDBC Connection Pools node and choose Add New JDBC Connection Pool.**

   This opens the New JDBC Connection Pool wizard.

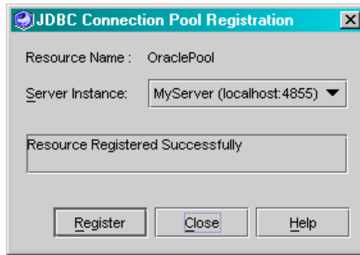
3. **Type** `OraclePool` **for the JDBC Connection Pool Name.**
4. **Enable the Extract From Existing Connection option.**

5. **Select the Oracle thin string from the pull-down menu.**
6. **Click Next, then Finish.**

   A window is displayed, asking whether you want to register this resource.
7. **Click the Register button.**

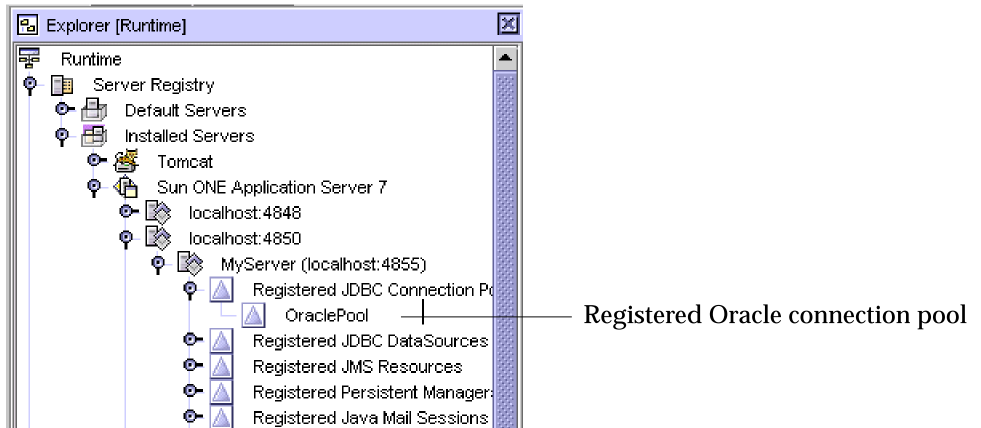
   The JDBC Connection Pool Registration dialog box is displayed.
8. **Select the server instance you wish to register to from the list and click Register.**

   When the connection pool is registered, a message is displayed indicating success.



9. **Click the Close button to close the window.**

   The registered OraclePool connection pool is displayed.



If you do not see the OraclePool connection pool, right-click the Registered JDBC Connection Pools node and choose Refresh List.

Creating a JDBC Data Source

A JDBC data source (also called a JDBC resource) lets you make connections to a database with the getConnection() method. Before creating a data source, make sure both the admin server and the application server are running.

To create a JDBC persistent manager:

1. **If necessary, expand the Server Registry, Installed Servers, and Sun ONE Application Server 7 nodes in the Explorer's Runtime page.**
2. **Right-click the Unregistered JDBC Data Sources node and choose Add New Data Source.**

   This opens the New JDBC Resource wizard.
3. **Enable the Use Existing JDBC Data Source option, and select** `OraclePool` **from the list.**
4. **Type `jdbc/jdbc-oracle` for the JNDI name and select True in the Enabled field.**
5. **Click Finish.**

   A window is displayed, asking whether you want to register this resource.
6. **Click the Register button.**

   The Persistence Manager Registration dialog box is displayed.
7. **Select the server instance you wish to register to from the list and click Register.**

   When the data resource is registered, a message is displayed indicating success.
8. **Click the Close button to close the window.**

   The jdbc/jdbc-oracle node is displayed under the Registered JDBC DataSources node. If you don't see it, right-click the JDBC Data Sources node and choose Refresh List.

Creating a JDBC Persistent Manager

A persistent manager is a component responsible for the persistence of the entity beans installed in the container. Before creating a persistent manager, make sure both the admin server and the application server are running.

To create a JDBC persistent manager:

1. **If necessary, expand the Server Registry, Installed Servers, Sun ONE Application Server 7, and the nodes in the Explorer's Runtime page.**

2. **Right-click the Unregistered persistent managers node and choose Add a Persistent Manager.**

   This opens the New Persistence Manager wizard.

3. **Enable the Use Existing JDBC Resource option, and select** `jdbc/jdbc-oracle` **from the list.**

4. **Type `jdo/OraclePm` for the JNDI name and select True in the Enabled field.**

5. **Click Finish.**

   A window is displayed, asking whether you want to register this resource.

6. **Click the Register button.**

   The Persistence Manager Registration dialog box is displayed.

7. **Select the server instance you wish to register to from the list and click Register.**

   When the persistent manager is registered, a message is displayed indicating success.

8. **Click the Close button to close the window.**

   The jdo/OraclePm node is displayed under the Registered Persistence Managers node. If you don't see it, right-click the node and choose Refresh List.

Creating the Database Tables

The DiningGuide tutorial uses two database tables, which you must create in an Oracle Server database. The instructions that follow describe how to use the provided SQL script to create your tables. Microsoft Windows users can copy and paste the SQL script provided in Appendix B to create these tables. Solaris and Linux users can use a script file, `diningguide_ora.sql`, which is available within DiningGuide application files.

To install the tutorial tables in an Oracle database on Microsoft Windows systems:

1. **Open the Oracle Console by choosing Start → Programs → Oracle (your version) → Application Development → SQL Plus.**

2. **Log in to SQL Plus using your user name and password.**

   For example, use the user name (scott) and password (tiger) for the default Oracle installation.

3. **When the SQL prompt appears, copy the script from Chapter B and paste it next to the prompt.**

Tip – Avoid the first two DROP statements, which refer to tables that have not yet been created and will create harmless errors. These DROP statements are useful in future, however, if you want to rerun the script to initialize the tables.

To install the tutorial database on Solaris or Linux environments:

1. **Unzip the `DiningGuide.zip` file from the Developer Resources portal.**

   For example, unzip it to the `/MyZipFiles` directory.

2. **At a command prompt, type:**

```
$ cd your-unzip-dir/DiningGuide/db
$ sqlplus db-userid/db-password@db-servicename @diningguide_ora.sql
```

For example,

```
$ cd /MyZipFiles/DiningGuide/db
$ sqlplus scott/tiger@MyDB @diningguide_ora.sql
```

The two DROP statements will generate errors, but they are harmless.

Creating EJB Components with an Oracle Database

This section lists the changes to Chapter 3 required to create the EJB tier using an Oracle database. TABLE C-1 lists the sections and the changes required.

TABLE C-1 Oracle-Specific Changes to Chapter 3

Section	Change
"Creating a Database Schema for the Tutorial's Tables" on page 53, Step 2b.	1. Expand the Drivers folder, right-click the Oracle thin node and choose New Database Connection. 2. In the dialog box, specify your Oracle URL, User Name, and Password. 3. Check Remember password during this session and click OK. 4. Close the Drivers folder. An Oracle connection appears.
Same section, Step 5b.	Select the Oracle connection from the list.
"Creating the Restaurant Entity Bean" on page 55	Use the Oracle schema to create the Restaurant entity bean.
"Creating the Customerreview Entity Bean" on page 58	Use the Oracle schema to create the Customerreview entity bean.
"Providing the Sun ONE Application Server 7 Plugin With Database Information" on page 75, Step 9.	Step 9. Type `jdo/OraclePM` in the JNDI Name field. Step 10: Type your Oracle database username and password.
"Checking the Additions to the Database" on page 81	1. Start SQLPlus. 2. Log into your database. 3. Enter the SQL statements.
"Creating a Test Client for the Customerreview Bean" on page 82	Step 8. Type `jdo/OraclePM` in the JNDI Name field. Step 9. Type your Oracle database username and password.
"Providing the Sun ONE Application Server 7 Plugin With Database Information" on page 102	Step 9. Type `jdo/OraclePM` in the JNDI Name field. Step 10: Type your Oracle database username and password.

Creating the Web Service with an Oracle Database

This section lists the changes to Chapter 4 required to create the web service using an Oracle database. TABLE C-2 lists the sections and the changes required.

TABLE C-2 Oracle-Specific Changes to Chapter 4

Section	Change
"Using the Test Application to Test the Web Service" on page 121, Step 2	Start SQLPlus, log into your database, and test the insertion of the record, as described.

Index

A

accessor methods, exposing to the user, 65
Add Admin Server menu item, 25
Add Business Method menu item, 66
Add Constructor menu item, 70
Add Create Method menu item, 60
Add Finder Method menu item, 64
Add Module menu item, 118

B

business methods
  Customerreview
    getCustomerreviewDetail, 72
    getReview, 67
  DiningGuideManager
    createCustomerreview, 95, 107
    getAllRestaurants, 91, 107
    getCustomerreviewDetail, 98, 107
    getCustomerreviewsByRestaurant, 93, 108
    getRestaurantDetail, 97, 108
  Restaurant
    getRating, 66, 81
    getRestaurantDetail, 71
  Restaurant.getRestaurantDetail, 50
business methods, Swing client
  CustomerReviewTable
    getCustomerReviewByName, 140
    jButton1ActionPerformed, 142
    putDataToTable, 139, 140, 143
    refreshView, 142, 144
  RestaurantTable
    getAllRestaurants, 137
    jButton1ActionPerformed, 139, 141
    putDataToTable, 138

C

client file methods
  getAllRestaurants, 127
  getCustomerreviewsByRestaurant, 122
constructors
  CustomerreviewDetail, 71
  RestaurantDetail, 70
Create a Server Instance menu item, 26
create methods
  Customerreview.create, 62, 78, 84
  DiningGuideManager.create, 88 to 90, 106
  JNDI lookup code in, 89
  Restaurant.create, 60, 79, 85
Create New EJB Test Application menu item, 73, 100
Create New Web Service Test Client command, 117
Creating a web service, 44
creating a web service, 113 to 115
creating a web service client, 118
Customerreview entity bean
  create method, creating, 62
  creating, 52 to 67
  getReview method, creating, 67
  testing, 82

CustomerReview table, description, 33
Customerreview_TestApp, 82
  bean methods, testing, 84 to 86
  creating, 82 to 83
  deploying, 84
  undeploying, 104 to 105
CustomerReviewTable
  copying into the DiningGuide application, 133 to 134
  displayed, 37, 135, 136

D

databases
  creating a database schema (Oracle), 185
  creating a database schema (PointBase), 53
  creating an entity bean from a schema, 52 to 59
  creating the tutorial tables (Oracle), 183 to 184
  installing the Oracle JDBC driver, 178
  installing the PointBase JDBC driver, 30
  opening a connection (Oracle), 185
  opening a connection (PointBase), 53
  setting up connectivity (Oracle), 177 to 183
  setting up connectivity (PointBase), 29 to 32
  supported versions, 22
  tutorial SQL script (Oracle), 175
  tutorial SQL script (PointBase), 174
  viewing data in the IDE, 39, 81, 86
Deploy menu item, 120
deploying
  adding Sun ONE Application Server 7 properties for entity beans (Oracle), 185
  adding Sun ONE Application Server 7 properties for entity beans (PointBase), 75 to 77, 83
  adding Sun ONE Application Server 7 properties for session beans (Oracle), 185
  adding Sun ONE Application Server 7 properties for session beans (PointBase), 102 to 103
  test applications for entity beans, 77, 84
  test applications for session beans, 105
  undeploying with the IDE, 104 to 105
detail classes
  creating, 68 to 71
  description, 43, 49
DGApp, 121
DGWebService, 114
DiningGuide application
  application scenarios, 36
  architecture, 40
  database script (Oracle), 175
  database script (PointBase), 174
  database tables, description, 33
  deploying, 45, 119
  EJB tier, 47 to 51
  functional description, 35
  functional specs, 36
  limitations, 46, 53
  requirements, 22
  Swing client, adding to the application, 133 to 134
  Swing client, examining, 137 to 144
  Swing client, executing, 134
  user's view, 37
  zipped source files, 21, 177
DiningGuide Swing client
  executing, 37
  generated from web services, 122
  installing and using, 45
DiningGuideManager session bean
  create method, coding, 88 to 90
  createCustomerreview method, 82, 95 to 96, 107, 124
  creating, 87
  getAllRestaurants method, 91 to 92, 107
  getCustomerreviewDetail method, 97, 107
  getCustomerreviewsByRestaurant method, 93 to 95, 108
  getRestaurantDetail method, 97, 108
  testing, 106 to 108
DiningGuideManager_TestApp
  bean methods, testing, 106 to 108
  creating, 100 to 103
  deploying, 105

E

EJB Builder
  entity beans, creating, 52 to 59
  local or remote interfaces, 52, 87
  session beans, creating, 87
  using, 43
EJB QL, using in finder methods, 63
EJB tier overview, 41, 47 to 51

entity beans
- adding to an EJB module, 101
- business methods, creating, 65
- business methods, testing, 81
- create methods, creating, 60
- create methods, testing, 78
- creating, 52 to 59
- finder methods, creating, 63
- finder methods, testing, 80
- local or remote interfaces, 52
- primary key class, 59
- testing, 78 to 81
- validating, 66

example applications
- StockApp and UDDI registry, 45
- where to download, 18

executing
- test applications for entity beans, 77, 84
- test applications for session beans, 105

F

finder methods
- `Customerreview.findByRestaurantName`, 63, 80
- `Restaurant.findAll`, 50, 63, 80
- testing, 80

G

Generate Client Files menu item, 131

Generate Web Service Files menu item, 115

generated runtime classes, 112

generated web service, 112

I

interfaces, local or remote
- for entity beans, 52
- for session beans, 87

J

J2EE applications
- creating, 118
- deploying, 119
- `DGApp`, 118

Javadoc technology, using in the IDE, 18

JDBC connection pools
- creating (Oracle), 180 to 181
- creating as administrator user (PointBase), 30
- creating as all other users (PointBase), 31
- purpose, 29

JDBC data sources
- creating (Oracle), 182
- creating as administrator user (PointBase), 30
- creating as all other users (PointBase), 31
- purpose, 29

JDBC drivers
- enabling (Oracle), 178
- enabling (PointBase), 30

JNDI lookup code, 89

M

Mount Filesystem menu item, 53

N

Netscape browser, supported version, 22

New CMP Entity EJB menu item, 55

New EJB Test Application menu item, 73

New J2EE Application menu item, 118

New Java Bean menu item, 68

New Web Service menu item, 113

O

Oracle database
- connecting to the IDE, 179 to 180
- installing a type 4 JDBC driver, 178
- *See also* databases

Overview of tasks, 42 to 45

P

parameters
- changing order of, 79
- order in test client, 79

persistent managers
- creating (Oracle), 182 to 183
- creating as administrator user (PointBase), 30
- creating as all other users (PointBase), 31
- purpose, 29

PointBase database
- installing a type 4 JDBC driver, 30
- supported version, 22

R

`Restaurant` entity bean
- `create` method, 60, 79, 85
- creating, 52 to 67
- `findall` method, 50
- `getRating` method, 66, 81
- `getRestaurantDetail` method, 50

Restaurant table, description, 33

`Restaurant_TestApp`
- bean methods, testing, 78 to 81
- creating, 73 to 77
- deploying, 77
- undeploying, 104 to 105

`RestaurantTable`
- copying into the DiningGuide application, 133 to 134
- displayed, 37, 135

`runide.sh` script, 23

S

session beans
- business methods, creating, 91 to 94
- create method, modifying, 88
- create method, testing, 106
- creating, 87 to 100
- EJB references, adding, 98 to 100
- local or remote interfaces, 87
- testing, 100 to 108
- validating, 98

Sun ONE Application Server 7
- confirming as the default server, 29
- database connectivity, 29 to 32
- starting the admin server (standard user), 25 to 27
- starting the admin server (superuser), 24
- starting the application server, 28
- stopping, 136

Sun ONE Application Server 7 server
- plugin properties to set for entity beans (Oracle), 185
- plugin properties to set for entity beans (PointBase), 75 to 77, 83
- plugin properties to set for session beans (Oracle), 185
- plugin properties to set for session beans (PointBase), 102 to 103

Sun ONE Studio 5 IDE
- connecting to the Oracle server, 179 to 180
- creating a database schema (Oracle), 185
- creating a database schema (PointBase), 53
- opening a database connection (Oracle), 185
- opening a database connection (PointBase), 53
- setting up database connectivity (Oracle), 177 to 183
- setting up database connectivity (PointBase), 29 to 32
- starting the IDE, 23

Sun ONE Studio 5, Standard Edition, where to obtain, 22

Swing client
- adding to the DiningGuide application, 133 to 134
- examining the code, 137 to 144
- executing, 135

T

test application facility
- adding entity beans to the EJB module, 101
- entity beans, testing, 78 to 82
- session bean, testing, 100 to 108
- test client, creating, 73 to 77, 82 to 83, 100 to 103
- test client, deploying, 77, 84, 105
- test client, using, 78 to 82, 106 to 108
- using, 43
- web service, testing, 117 to 128

test applications
  Customerreview_TestApp, 82
  DGApp, 117
  DiningGuideManager_TestApp, 100
  Restaurant_TestApp, 73
testing enterprise beans
  business methods, testing, 81
  create method, testing, 79, 85
  finder methods, testing, 80
  results in IDE's output window, 78, 120
  results in J2EE command window, 78, 120
  test client page, 78, 106

U

undeploying an application
  how to, 104 to 105
  reasons for, 104, 119
Using the test application facility, 43

V

Validate EJB menu item, 66, 98
View Data menu item, 39, 81, 86

W

web browsers, supported versions, 22
web service
  client files, 112
  creating, 113 to 115
  description, 111 to 113
  exposing class types underlying collection types, 96 to 98
  generating client files, 131
  generating WSDL, 129
  sharing with other developers, 129 to 131
  testing, 117 to 128
web service client methods
  createCustomerreview, 142
  getAllRestaurants, 137
  getCustomerreviewsByRestaurant, 140
Web Service Descriptive Language (WSDL), generating, 129