



Sun[™] ONE Studio 4, Enterprise Edition for Java[™] Tutorial

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

Part No. 816-7860-10
September 2002, Revision A

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

Copyright © 2002 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.

This product includes code licensed from RSA Data Security.

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.

Netscape and Netscape Navigator are trademarks or registered trademarks of Netscape Communications Corporation in the United States and other countries.

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 © 2002 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 ou document est protégé par un copyright et distribué avec des licences qui 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, par quelque 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.

Ce produit comprend le logiciel licencié par RSA Data Security.

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 portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

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

Netscape et Netscape Navigator sont des marques de Netscape Communications Corporation aux Etats-Unis et dans d'autres pays.

LA DOCUMENTATION EST FOURNIE “EN L'ÉTAT” ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISÉE 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.



Contents

Before You Begin xiii

1. Getting Started 1

Software Requirements for the Tutorial 2

Requirements for IDE-to-Application Server Communication 3

 Software Requirements 3

 Installation Options 3

Installing the Oracle Type 4 Driver in the IDE and the Application Server 4

Starting the Software 4

Installing the Application Server Plugin 6

Connecting the IDE to the Application Server 7

 Configuring the IDE to Use the Application Server 7

 Making the Sun ONE Application Server the Default Server 8

Connecting the IDE and the Application Server to the Database 9

 Defining a JDBC Connection Pool 9

 Registering the Connection Pool 11

 Defining a JDBC Data Source 12

 Registering the JDBC Data Source 12

 Defining a JDBC Persistence Manager 13

Registering the JDBC Persistence Manager	13
Creating the Database Tables	14
Tutorial Database Table Descriptions	15
2. Introduction to the Tutorial	17
Functionality of the Tutorial Application	17
Application Scenarios	18
Application Functional Specification	19
User's View of the Tutorial Application	19
Architecture of the Tutorial Application	22
Application Elements	23
EJB Tier Details	24
Overview of Tasks for Creating the Tutorial Application	25
Creating the EJB Components	25
Creating the Tutorial's Web Service	27
Installing and Using the Provided Client	28
End Comments	29
3. Building the EJB Tier of the DiningGuide Application	31
Overview of the Tutorial's EJB Tier	31
The Entity Beans	32
The Session Bean	33
The Detail Classes	33
Summary of Steps	35
Creating Entity Beans With the EJB Builder	36
Creating the Restaurant and Customerreview Entity Beans	36
Creating the Customerreview Entity Bean	41
Creating Create Methods for CMP Entity Beans	43
Creating Finder Methods on Entity Beans	46

Creating Business Methods for Testing Purposes	48
Creating Detail Classes to View Entity Bean Data	50
Creating the Detail Classes	50
Creating the Detail Class Properties and Their Accessor Methods	51
Creating the Detail Class Constructors	52
Creating Business Methods on the Entity Beans to Fetch the Detail Classes	53
Testing the Entity Beans	54
Creating a Test Client for the Restaurant Bean	55
Providing the Sun ONE Application Server 7 Plugin With Database Information	57
Deploying and Executing the Restaurant Bean's Test Application	59
Using the Test Client to Test the Restaurant Bean	60
Checking the Additions to the Database	63
Creating a Test Client for the Customerreview Bean	64
Deploying and Executing the Customerreview Bean's Test Application	66
Testing the Customerreview Entity Bean	67
Checking the Additions to the Database	68
Creating a Session Bean With the EJB Builder	69
Coding a Session Bean's Create Method	70
Creating Business Methods to Get the Detail Data	72
Creating a Business Method to Create a Customer Review Record	76
Creating Business Methods That Return Detail Class Types	77
Adding EJB References	79
Testing the Session Bean	81
Creating a Test Client for a Session Bean	81
Providing the Sun ONE Application Server 7 Plugin With Database Information	82

Deploying and Executing the Test Application	84
Using the Test Client to Test a Session Bean	86
Checking the Additions to the Database	89
Comments on Creating a Client	89
4. Creating the DiningGuide Application's Web Service	91
Overview of the Tutorial's Web Service	92
The Web Service	92
The Runtime Classes	92
The Client Proxy Pages	92
Creating the Tutorial's Web Service	94
Creating the Web Service Module	94
Specifying the Web Service's SOAP RPC URL	96
Generating the Web Service's Runtime Classes	97
Testing the Web Service	97
Creating a Test Client and Test Application	98
Specifying the Web Context Property	100
Deploying the Test Application	100
Using the Test Application to Test the Web Service	102
Making Your Web Service Available to Other Developers	107
Generating the WSDL File	107
Generating a Client Proxy From the WSDL File	108
5. Creating a Client for the Tutorial Application	111
Creating the Client With the Provided Code	111
Creating the Client on Microsoft Windows Systems	112
Creating the Client in Solaris or Linux Environments	113
Running the Tutorial Application	114
Examining the Client Code	116

Displaying Restaurant Data	117
Displaying Customer Review Data for a Selected Restaurant	118
Creating a New Customer Review Record	121
A. DiningGuide Source Files	125
RestaurantBean.java Source	126
RestaurantDetail.java Source	129
CustomerreviewBean.java Source	134
CustomerreviewDetail.java Source	136
DiningGuideManagerBean.java Source	139
RestaurantTable.java Source	143
CustomerReviewTable.java Source	146
B. DiningGuide Database Script	151
Index	153

Figures

FIGURE 2-1 DiningGuide Application Architecture 23

FIGURE 3-1 Function of a Detail Class 34

Tables

TABLE 1-1	DiningGuide Database Tables	15
TABLE 1-2	Restaurant Table Records	16
TABLE 1-3	CustomerReview Table Records	16

Before You Begin

Welcome to the Sun™ Open Net Environment (Sun ONE) Studio 4, Enterprise Edition for Java tutorial. This tutorial shows you how to use the following features of the Sun ONE Studio 4 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 server 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 7 server—for testing the tutorial application

You can create the examples in this book in the environments listed in the release notes on the following web site:

<http://forte.sun.com/ffj/documentation/index.html>

Screen shots vary slightly from one platform to another. You should have no trouble translating the slight differences to your platform. Although almost all procedures use the interface of the Sun™ ONE Studio 4 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
```

To translate for UNIX® or Linux environments, simply change the prompt and use forward slashes:

```
% 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 4, Enterprise Edition for Java 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* (for J2EE SDK version 1.3)
http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html
- *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 4 IDE and the Sun ONE Application Server 7 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.

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:
<i>AaBbCc123</i>	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> save your changes.
<i>AaBbCc123</i>	Command-line variable; replace with a real name or value	To delete a file, type DEL <i>filename</i> .

Related Documentation

Sun ONE Studio 4 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.comSM 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 4 edition. Describe last-minute release changes and technical notes.

- Getting Started guides (PDF format)

Describe how to install the Sun ONE Studio 4 integrated development environment (IDE) on each supported platform and include other pertinent information, such as system requirements, upgrade instructions, application server configuration instructions, command-line switches, installed subdirectories, database integration, and information on how to use the Update Center.

- *Sun ONE Studio 4, Community Edition Getting Started Guide* - part no. 816-7871-10
- *Sun ONE Studio 4, Enterprise Edition for Java Getting Started Guide* - part no. 816-7859-10
- *Sun ONE Studio 4, Mobile Edition Getting Started Guide* - part no. 816-7872-10

- Sun ONE Studio 4 Programming series (PDF format)

- This series provides in-depth information on how to use various Sun ONE Studio 4 features to develop well-formed J2EE applications. *Building Web Components* - part no. 816-7869-10

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. 816-7863-10

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. 816-7864-10

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 4 EJB Builder wizard and other components of the IDE.

- *Building Web Services* - part no. 816-7862-10

Describes how to use the Sun ONE Studio 4 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. 816-7870-10

Describes how to use the JDBC productivity enhancement tools of the Sun ONE Studio 4 IDE, including how to use them to create a JDBC application.

- Sun ONE Studio 4 tutorials (PDF format)

These tutorials demonstrate how to use the major features of each Sun ONE Studio 4 edition.

- *Sun ONE Studio 4, Community Edition Tutorial* - part no. 816-7868-10

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

- *Sun ONE Studio 4, Enterprise Edition for Java Tutorial* - part no. 816-7860-10

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 4 development environment. You can access help by pressing the help key (Help in a Solaris environment, F1 on Microsoft Windows and Linux), 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 4 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 4 modules. Refer to the release notes for instructions on installing this documentation. When you start the IDE, you can access this Javadoc documentation within the Javadoc pane of the Explorer.

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 (816-7860-10) of your document in the subject line of your email.

Getting Started

This chapter describes what you must do before starting the Sun ONE Studio 4, Enterprise Edition for Java tutorial. The most important task is to get the Sun ONE Studio 4 IDE and the Sun ONE Application Server 7 application server to recognize each other, and to get both to communicate with the Oracle database.

The topics covered in this chapter are:

- “Software Requirements for the Tutorial” on page 2
- “Requirements for IDE-to-Application Server Communication” on page 3
- “Starting the Software” on page 4
- “Installing the Application Server Plugin” on page 6
- “Connecting the IDE to the Application Server” on page 7
- “Connecting the IDE and the Application Server to the Database” on page 9
- “Creating the Database Tables” on page 14

This chapter does not describe how to install the following required software:

- The Sun ONE Studio 4, Enterprise Edition for Java IDE

For installation instructions, see the *Sun ONE Studio 4, Enterprise Edition for Java Getting Started Guide*, available from

<http://docs.sun.com/source/816-7859-10>.

- The Oracle software—See Oracle documents on the Oracle distribution CD
- The Sun ONE Application Server 7 application server

To download the application server, go to

http://sun.com/software/products/appsrvr/appsrvr_download.html

Installation information is available from

<http://docs.sun.com/source/816-7145-10/>.

The Sun ONE Studio 4 and Sun ONE Application Server 7 products can be installed together as the Sun ONE Application Server 7, Sun ONE Studio 4 bundle. The *Sun ONE Studio 4, Enterprise Edition for Java with Application Server 7 Tutorial* describes how to configure them when they are installed together. This chapter assumes that you have installed them separately.

Software Requirements for the Tutorial

Supported versions of the required software are provided in the release notes, available on <http://forte.sun.com/ffj/documentation/index.html>.

The following software is required for the DiningGuide tutorial:

- The Java™ 2 Software Development Kit (the J2SE™ SDK)

Both the IDE and the application server require the J2SE SDK. Supported versions are listed in the *Sun ONE Studio 4, Enterprise Edition for Java Getting Started Guide*, available from <http://docs.sun.com/source/816-7859-10>. The application server requires J2SE SDK v. 1.4.0_02, which is bundled with the application server. You can create the EJB tier of the tutorial with the IDE using a different version of the compiler, but you can't run the web services portion unless the IDE is using v. 1.4.0_02 or later of the J2SE SDK.

The most convenient scenario for installing the J2SE SDK, if you are installing the application server locally, is to install it with the application server. Then when you install the IDE, point to this J2SE SDK when prompted during the install.

Alternatively, you can download the J2SE SDK from the Java Developer's portal at <http://java.sun.com/j2se/>. If your application server is installed on a remote machine, you must install v. 1.4.0_02 or later of the J2SE SDK locally.

- A web browser

A web browser is required to view the pages of the test application client.

- Oracle database software

This tutorial requires an Oracle server or an Oracle client that accesses an Oracle server to be installed on the same machine as the IDE. The JDBC driver is also required. See the release notes for supported versions.

- The Sun ONE Application Server 7 application server

This tutorial requires the Sun ONE Application Server 7 application server to deploy the tutorial application.

This tutorial also uses the web server included in the Sun ONE Application Server 7 software to run the tutorial's test client.

The Sun ONE Application Server 7 server can be installed on the following platforms:

- Solaris 8 operating environment (32-bit or 64-bit SPARC platforms)
- Solaris 9 operating environment (32-bit or 64-bit SPARC platforms)
- Microsoft Windows 2000 Professional system (with the latest service packs)
- Microsoft windows XP Professional system (not Home Edition)

Requirements for IDE-to-Application Server Communication

Setting up the IDE to communicate with the Sun ONE Application Server 7 server requires the right software installed in the right location. There are many options, which this section attempts to clarify.

Software Requirements

For the IDE and the application server to communicate requires:

- The Sun ONE Application Server 7 plugin
Consists of JAR and configuration files that are installed in either the IDE's home directory, or your IDE user directory.
- Administrative client libraries
Consists of JAR files that support the server's administrator. These files must be available on the same machine as the IDE.

Installation Options

How you install the application server plugin and admin client libraries depends on whether the application server and the IDE are on the same machine or not, as follows:

- Local access (both IDE and server on same machine)
 - Option 1:** Obtain the plugin from the IDE's Update Center. The plugin can use the administrative libraries of the application server installed on your machine.
 - Option 2:** Install the plugin with the Sun ONE Application Server 7 installer.
- Remote access (IDE and server on different machines)
 - Option 1:** Install the plugin with the Sun ONE Application Server 7 installer. The admin libraries are included with the plugin.
 - Option 2:** Install the application server on your local machine. Then obtain the plugin from the IDE's Update Center. The plugin can use the admin libraries of the local server to administer the remote server.

If you choose to install the plugin with the Sun ONE Application Server 7 installer, you must use the correct installer, as follows:

- Evaluation installation—*Do not use* this installer, which installs all components of the application server in one operation.
- Development and Operation (or “non-evaluation”) installation—*Use this installer*, which can selectively install components of the application server.

Installing the Oracle Type 4 Driver in the IDE and the Application Server

Now, install Oracle into both the IDE and the application server. To do this, you need the Oracle Type 4 JDBC driver. You can download this driver from the Oracle portal. Copy the JDBC Type 4 driver into the program files of both products before you start either product.

To install the Oracle Type 4 driver in both the application server and the IDE.

- 1. Install Oracle in the Sun ONE Application Server 7 server by copying the Oracle Type 4 driver library to the `/lib` directory of the server’s instance.**

For example, copy the `classes12.zip` file to
`c:\Sun\AppServer7\domains\domain1\server1\lib\.`

- 2. Install the Oracle Type 4 driver library in the IDE, by copying the Oracle Type 4 driver library to the IDE’s `/lib/ext` directory.**

For example, copy the `classes12.zip` file to
`c:\Program Files\s1studio\ee\lib\ext\.`

Starting the Software

This section assumes you have already installed the IDE, the Sun ONE Application Server 7 server and the plugin, and the Oracle server (or the Oracle client, if the server is installed remotely). Although there are several startup methods, only one option is described here. See the installation documentation for the separate products for other options.

To start up the software:

1. Start the application server according to the server's *Getting Started Guide*.

The method depends on your platform.

- On Microsoft Windows choose Start → Programs → Sun Microsystems → Sun ONE Application Server 7 → Start Application Server

A command window appears and displays a message that the server instance has started. A second command window is then displayed showing server event log file content. Event messages appear, and after a few seconds a message appears confirming that the server instance has started successfully. You can close the first window, but do not close the second window.

- On Solaris, UNIX, or Linux environments, type the following in a terminal window:

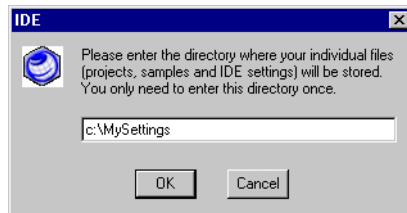
```
$ sh appserver-install-dir/bin/asadmin start-domain --domain domain1
```

The *appserver-install-dir* variable stands for the application server's home directory, which depends on your platform and type of installation. This command starts the initially configured domain, *domain1*, on your system.

2. Start the IDE.

- On Microsoft Windows, choose Start → Programs → Sun Microsystems → Sun ONE Studio 4 EE → Sun ONE Studio.

If this is the first time you have started the IDE, a dialog prompts you for a user settings directory. Use a complete specification, for example, *C:\MySettings*.



This value is stored in the registry for later use. For a given session, you can specify a different user settings directory by using the *-userdir* command-line switch when launching the IDE.

- On Solaris, UNIX, and Linux environments, type the following in a terminal window:

```
$ sh s1studio-home/bin/runide.sh
```

The `s1studio-home` variable stands for the IDE's home directory, for example, `$HOME/s1studio/ee`. The user settings directory is by default located in `user-home/ffjuser40ee`. You can specify a different directory with the `-userdir` command-line switch when you start the IDE.

3. If not already started, start your Oracle database server.

Consult your Oracle documentation for this step.

Installing the Application Server Plugin

If the application server you will access is installed on a remote machine, you must use the Sun ONE Application Server 7 “non-evaluation” installer to install either the plugin or the entire application server. For information on how to do this, see the *Sun ONE Application Server 7 Standard Edition Installation Guide*, obtainable from <http://docs.sun.com/source/816-7145-10/>.

If your application server is installed on the same machine as the IDE, you can install the plugin from the IDE's Update Center, as follows:

1. In the Sun ONE Studio 4 IDE, choose Tools → Update Center.

The Update Center wizard is displayed.

2. Select the Sun ONE Studio Update Center as the Update Center, and deselect NetBeans Update Center.

3. Click the Proxy Configuration button to set your proxy configuration, if needed.

The Proxy Configuration dialog box is displayed.

4. Modify the values as needed and click OK to return to the Update Center wizard.

5. Click Next, and type your Sun ONE Studio Update Center login name and password.

See the *Sun ONE Studio 4, Enterprise Edition for Java Getting Started Guide* for information on registering and creating a login name and password.

The Update Center displays the modules that are available to you.

6. Expand the Sun ONE Studio Update Center node and select the Sun ONE Application Server 7 Plug-in and click the arrow pointing to the right.

The plugin is moved to the right pane.

7. Follow the Update Center installation procedures.

The IDE installs the plugin and then restarts itself.

Once restarted, the IDE has the means to connect with the Sun ONE Application Server 7 server.

Note – When you install the plugin through the IDE’s update center, the IDE may place it in your user directory (depending on user option settings). When you install the plugin using the Sun ONE Application Server 7 installer, it places the plugin in the IDE distribution. If you reinstall the plugin with a different method, you must delete the old plugin files, wherever they are located.

Connecting the IDE to the Application Server

Now configure the IDE to use the Sun ONE Application Server 7 server and make it the default server.

Configuring the IDE to Use the Application Server

1. **In the Sun ONE Studio 4 IDE, click the Explorer’s Runtime tab.**
2. **Expand the Server Registry node and its Installed Servers subnode.**
3. **Select the Sun ONE Application Server 7 node to display its properties in the Properties window.**

The Properties window is usually below the Explorer. If the Properties window is not displayed, choose View > Properties to display it.

4. **Locate the Sun ONE App Server Home property and click in its value field.**

An ellipsis button appears.

5. **Click the ellipsis button.**

A file finder window is displayed.

6. **Use the file finder to select the Sun ONE Application Server 7 home directory.**

For example: c:\Sun\appServer7

7. **Click OK to close the file finder window.**

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

The Add Admin Server dialog box is displayed.

9. **Type the server's host name in the Admin Server Host field.**

If the application server that you want to use is installed on the same machine as the IDE, type `localhost`. If the server is remote, type the host name of the remote machine.

10. **Type your admin port number in the Admin Server Port field, your server user name in the User Name field, and your server password in the User Password field.**

During the installation of the Sun ONE Application Server 7 server, the following values were suggested for these fields:

- Admin Server Port: 4848
- User Name: admin
- User Password: (your choice)

If you did not use these defaults, specify the values that you did use. If you don't know the values to use, consult *Getting Started with Sun ONE Application Server 7* for instructions on where to look for this information.

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

A node labeled *hostname:admin-server-port* (for example, `localhost:4848`) is created under the Sun ONE Application Server 7 node. This node represents the admin server. You can issue commands to this node and the admin server will execute them.

12. **Expand the *hostname:admin-server-port* node.**

A node representing an instance of the application server is displayed.

Making the Sun ONE Application Server the Default Server

By default, Sun ONE Studio 4 is configured to use the J2EE reference implementation for its application server. To make the Sun ONE Application Server 7 server the default:

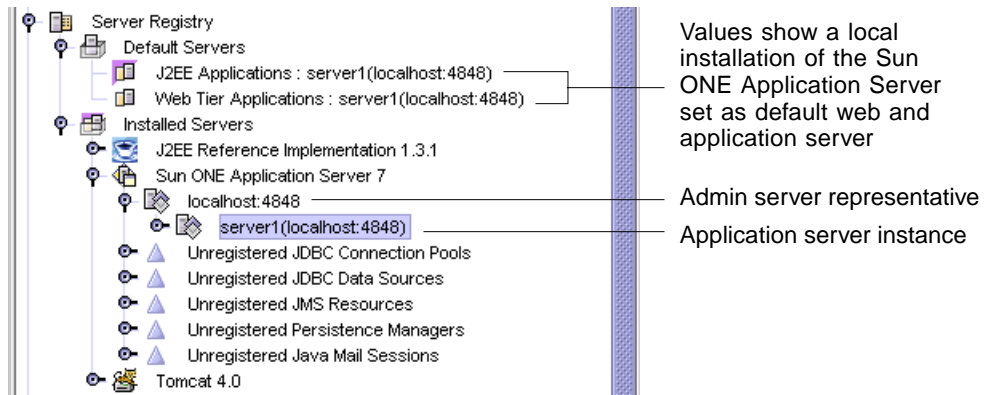
1. **Expand the Sun ONE Application Server 7 node and the admin representative subnode.**

For example, this node might be labeled `localhost:4848`. An application server instance node (labeled something like `server1(localhost:4848)`) is a subnode of the admin node.

2. Right-click the instance node and choose Set As Default.

3. Expand the Default Servers node under the Server Registry node.

If the Server Registry looks like this, then the IDE is using the correct servers.



Connecting the IDE and the Application Server to the Database

Now, configure the Sun ONE Application Server 7 application server to connect to the Oracle database. Do this by creating and registering a new connection pool, a JDBC data source, and a JDBC persistence manager. These items are required when you deploy an application that includes entity beans that use container-managed persistence (the CMP EJB components used in the tutorial's EJB tier).

Before you continue, make sure you have added the Oracle JDBC Type 4 driver to both the IDE and the application server, as described in "Installing the Oracle Type 4 Driver in the IDE and the Application Server" on page 4.

Defining a JDBC Connection Pool

1. In the Explorer, click the Runtime tab.

2. Expand the Server Registry, Installed Servers, and Sun ONE Application Server 7 nodes.

3. **Right-click the Unregistered JDBC Connection Pools node and choose Add New JDBC Connection Pool.**

This opens the JdbcConnectionPool property window and a new node is created under the Unregistered JDBC Connection Pools node.

4. **For the Datasource Classname property, type**
`oracle.jdbc.pool.OracleDataSource.`
5. **For the Name property, type** `OraclePool.`
6. **Select the ellipsis button for the Properties property.**

The Properties property editor is displayed.

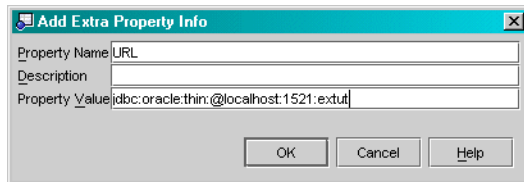
7. **Click the Add button.**

The Add Extra Property Info dialog box is displayed.

8. **Type URL in the Property Name field.**

9. **Type** `jdbc:oracle:thin:@oraclehost:1521:oracle_SID` **in the Property Value field.**

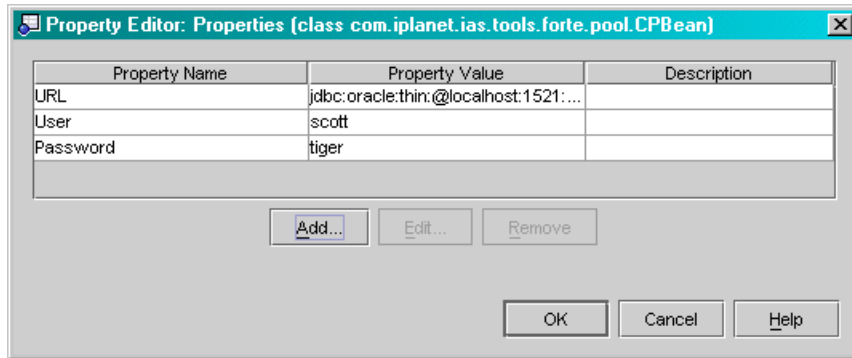
The Add Extra Property Info dialog box looks similar to this:



10. **Click OK to create the property and close the dialog box.**
11. **Repeat Step 7 through Step 10 to add two more new properties with the following values:**

Property Name	Property Value
User	<i>your_username</i>
Password	<i>your_password</i>

The values for these properties are the user name and password you use to access your database. When you are done, the property editor looks similar to this:

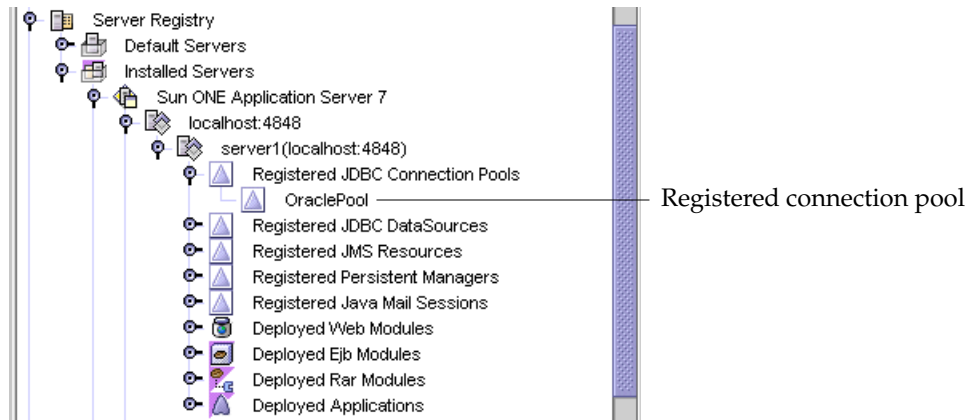


12. Click OK to close the property editor, and close the connection pool's property window.

Registering the Connection Pool

To register the connection pool you just created:

1. **Expand the Unregistered JDBC Connection Pools node.**
Note the new node for OraclePool.
2. **Right-click the OraclePool node and choose Register.**
The Select Server to Register to dialog box is displayed.
3. **Select the entry (for example, server1(localhost:4848) and click the Register button.**
When the connection pool is registered, an information window is displayed, indicating success.
4. **Close the information window and the Select Server to Register to dialog box.**
5. **Expand the server instance under the Sun ONE Application Server 7 node.**
Nodes for registered resources are displayed.
6. **Expand the Registered JDBC Connection Pools node in this list.**
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.

Defining a JDBC Data Source

To define a JDBC data source for the Oracle database:

1. **If necessary, expand the Server Registry, Installed Servers, and Sun ONE Application Server 7 nodes in the Explorer's Runtime pane.**
2. **Right-click the Unregistered JDBC Data Sources node and choose Add New Data Source.**

This opens the DataSource property window and a new node is created under the Unregistered JDBC Data Sources node.

3. **Type `jdbc/jdbc-oracle` for the JNDI Name property.**
4. **Select `OraclePool(server1:hostname:admin-server-port)` from the list for the Pool Name.**
5. **Close the property window.**

Registering the JDBC Data Source

Register the JDBC data source you just created the same way you registered the connection pool in "Registering the Connection Pool" on page 11, namely:

1. **Expand the Unregistered JDBC Data Sources node.**

2. **Right-click the `jdbc/jdbc-oracle` node and choose Register.**

The Select Server to Register to dialog box is displayed.

3. **Select the entry (for example, `server1(localhost:4848)` and click the Register button.**

When the data source is registered, an information window is displayed, indicating success.

4. **Close the information window and the Select Server to Register to dialog box.**

If you do not see the `jdbc/jdbc-oracle` node, right-click the `JDBC Data Sources` node and choose Refresh List.

Defining a JDBC Persistence Manager

To define a JDBC persistence manager to support container-managed persistence (CMP) for the Oracle database:

1. **If necessary, expand the Server Registry, Installed Servers, Sun ONE Application Server 7, and the nodes in the Explorer's Runtime pane.**

2. **Right-click the Unregistered Persistence Managers node and choose Add a Persistence Manager.**

This opens the PersistenceManagers property window and a new node is created under the Unregistered Persistence Managers node.

3. **Type `jdbc/jdbc-oracle` for the JDBC Resource JNDI Name property.**

4. **Type `jdo/OraclePm` for the JNDI Name property.**

5. **Close the property window.**

Registering the JDBC Persistence Manager

Register the JDBC persistence manager you just created in the usual way, namely:

1. **Expand the Unregistered Persistence Managers node.**

2. **Right-click the `jdo/OraclePm` node and choose Register.**

The Select Server to Register to dialog box is displayed.

3. **Select the entry (for example, `server1(localhost:4848)` and click the Register button.**

When the data source is registered, an information window is displayed, indicating success.

4. Close the information window and the Select Server to Register to dialog box.

Right-click the Registered Persistence Managers node and choose Refresh List if you don't see the `jdo/OraclePm` node.

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, `rest_pb.sql`, which is available within DiningGuide application files.

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 a SQL script file for creating the required database tables. These files are compressed into a ZIP file you can download from the Sun ONE Studio 4 Developer Resources portal at <http://forte.sun.com/ffj/documentation/tutorialsandexamples.html>

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 in an Oracle database on Solaris or Linux environments:

- 1. Unzip the DiningGuide2.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/DiningGuide2/db
$ sqlplus db-userid/db-password@db-servicename @rest_ora.sql
```

For example,

```
$ cd /MyZipFiles/DiningGuide2/db
$ sqlplus scott/tiger@MyDB @rest_ora.sql
```

The two DROP statements will generate errors, but they are harmless.

Tutorial Database Table Descriptions

The script in Appendix B creates the database schemas shown in TABLE 1-1.

TABLE 1-1 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 1-2.

TABLE 1-2 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 1-3.

TABLE 1-3 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.

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 4, Enterprise Edition for Java 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 4, Enterprise Edition for Java 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 19
- “Architecture of the Tutorial Application” on page 22
- “Overview of Tasks for Creating the Tutorial Application” on page 25

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 redisplay 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 17 are realized.

1. **In the Sun ONE Studio 4 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 created in "Creating the Database Tables" on page 14.

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.

CUSTOMER NAME	REVIEW
---------------	--------

Customer Name

Review

In this case, no records are shown, because none are in the database. Refer to TABLE 1-3.

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 redisplay the customer review window, as shown:

CUSTOMER NAME	REVIEW
New User	I'm speechless

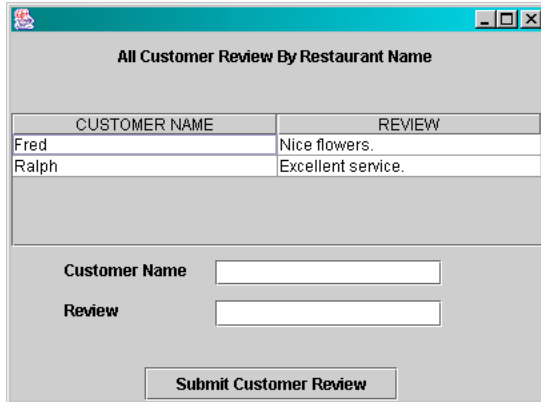
Customer Name

Review

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.



CUSTOMER NAME	REVIEW
Fred	Nice flowers.
Ralph	Excellent service.

Customer Name

Review

Two customer review records are displayed. Refer to TABLE 1-3 for confirmation.

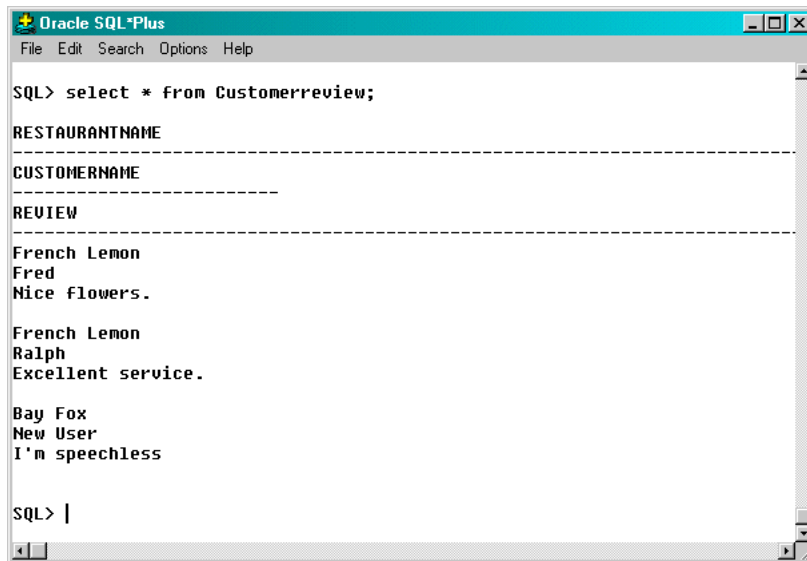
5. Continue to add and view customer review records.
6. When you are done, quit the application by closing any of the application's windows.
7. To verify that the new customer review records were written to the database, start the Oracle SQL Plus console.

The Oracle server must first be running. Refer to the procedures in "Creating the Database Tables" on page 14 for information.

8. In the SQL Plus Console, type the following statement:

```
select * from CustomerReview;
```

Your console should display the CustomerReview table with whatever reviews you entered, for example:



```
Oracle SQL*Plus
File Edit Search Options Help

SQL> select * from Customerreview;

RESTAURANTNAME
-----
CUSTOMERNAME
-----
REVIEW
-----
French Lemon
Fred
Nice flowers.

French Lemon
Ralph
Excellent service.

Bay Fox
New User
I'm speechless

SQL> |
```

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.

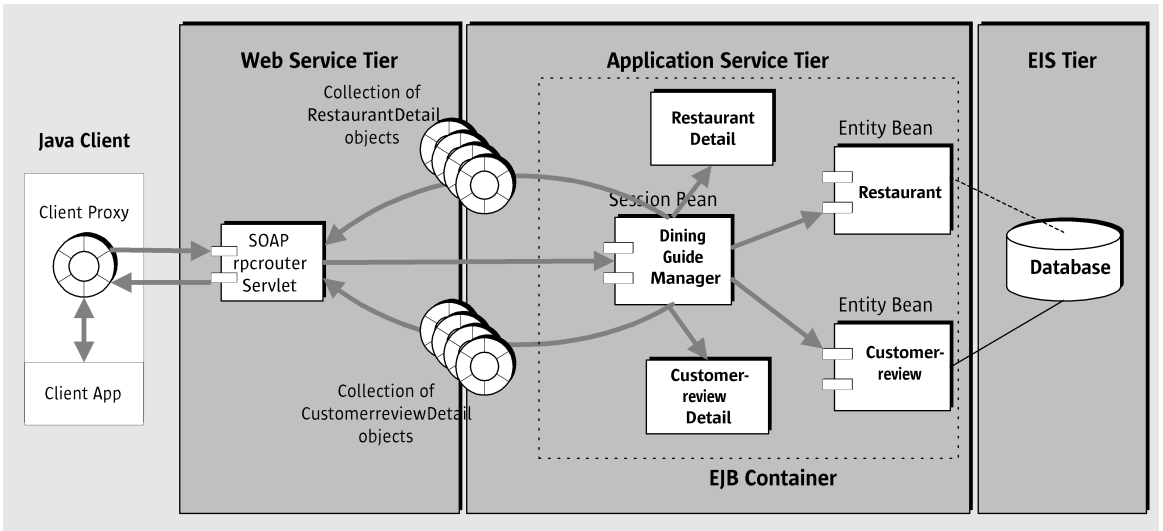


FIGURE 2-1 DiningGuide Application Architecture

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 4 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 4

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 4 IDE includes a facility for testing enterprise JavaBean components without your having to create a client for this purpose. This facility uses the J2EE Reference Implementation 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 4 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 4 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 4 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 4 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 4, Enterprise Edition for Java, 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 4 Developer's portal at:

<http://forte.sun.com/ffj/documentation/tutorialsandexamples.html>

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 36
- "Creating Detail Classes to View Entity Bean Data" on page 50
- "Testing the Entity Beans" on page 54
- "Creating a Session Bean With the EJB Builder" on page 69
- "Testing the Session Bean" on page 81
- "Comments on Creating a Client" on page 89

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 4 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 4 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 4 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 has the same fields as the corresponding entity bean, 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.

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

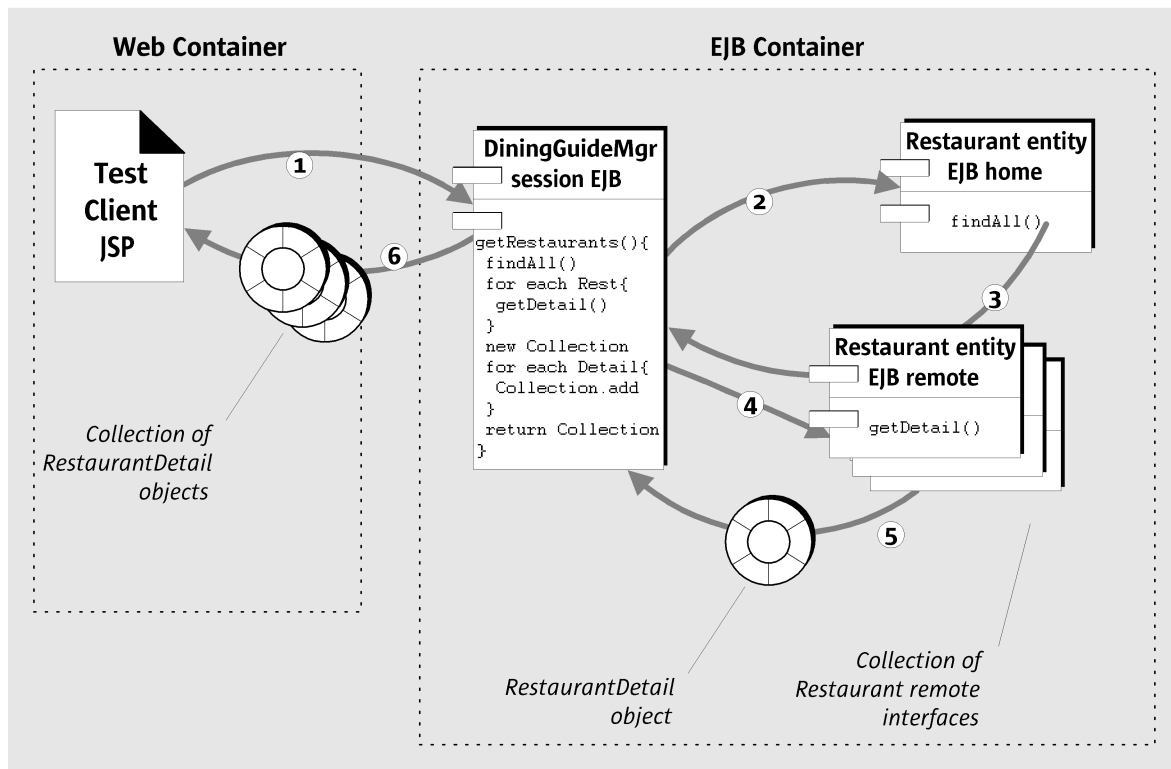


FIGURE 3-1 Function of a Detail Class

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 CMP entity bean skeletons with the EJB Builder wizard. Then you add their create and finder methods and simple business methods for testing purposes.

2. Creating detail classes that have the same fields as the entity beans

You create regular JavaBeans `Restaurant` and `Customerreview` classes and the getter and setter methods for each field.

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

Viewing the automatically generated test client in a web browser, you exercise the create method to create an instance of the bean and make its business methods available. Then you exercise the bean's business methods.

5. Creating the session bean

You create a stateless session bean skeleton with the EJB Builder, and modify the bean's create method to perform a lookup on the entity beans. Then you create getter methods for constructing collections of detail objects (from the detail classes) for each entity bean and a method to create a customer review record in the database. You also create two dummy business methods that are required by the SOAP runtime.

6. Using the test application facility again to test the session bean

In the EJB module's property sheet, you add references to the CMP entity beans. Then you create a test application and add the EJB modules to the test application's EJB module. To conclude, 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 4 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 mount as a filesystem to contain the application and create a package for the EJB tier. Next, create database schemas to model the two database tables. Finally, create two entity beans within the package.

Note – The following instructions assume that the Sun ONE Studio 4 IDE and the Oracle server in which you created the tables (see “Creating the Database Tables” on page 14) are both running.

Creating the Tutorial's Directory

Create a directory to contain the tutorial's files and mount it in the IDE's file system. Then create a Java package under this directory, as follows:

1. **Somewhere on your file system, create a directory and name it `DiningGuide2`.**

This tutorial uses `DiningGuide2` for the directory name to distinguish it from the directory name (`DiningGuide`) given to the tutorial released with Version 4.0 of Sun ONE Studio.

Note – If you create this as a subdirectory of another directory, the specifications of some methods you create for this tutorial may be 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 `d:\DiningGuide2`).

2. **In the Sun ONE Studio 4 IDE, choose the `File` → `Mount Filesystem`.**

The New wizard is displayed.

3. **Select `Local Directory`, and click `Next`.**

The Select Directory pane of the New wizard is displayed.

4. **Use the `Look In` file finder to find the `DiningGuide2` directory, select it, and click `Finish`.**

The new directory (for example, `c:\DiningGuide2`) is mounted in the Explorer.

5. **Right-click the new filesystem you just mounted and choose `New` → `Java Package`.**

You will use this package to hold the EJB tier of the application. This tier holds the data of your application.

6. **Name the new package `Data` and click `Finish`.**

The new `Data` package appears under the `DiningGuide2` directory.

Creating Database Schemas for the Tutorial's Tables

Now create database schemas for the `Restaurant` and `Customerreview` tables in the `Data` package, as follows:

1. **Right-click the `DiningGuide2` node in the Explorer and choose `New` → `Databases` → `Database Schema`.**

The Name Object pane of the New wizard is displayed.

2. **Type `restSchema` in the `Name` field and click `Next`.**

The Database Connection pane of the wizard is displayed.

3. **Select the `New Connection` option.**

4. Enter values in the fields appropriate to your database.

For example, the following values are correct for a locally installed database with a SID of "extut," and the default Oracle login of "scott" for User Name and "tiger" for Password:

Field Name	Value
Name	Oracle thin (select from the list)
Driver	oracle.jdbc.driver.OracleDriver
Database URL	jdbc:oracle:thin:@localhost:1521:extut
User Name	scott
Password	tiger

5. Click Next.

The Tables and Views pane is displayed.

6. Select the RESTAURANT table in the list of available tables and click the Add button.

The RESTAURANT table moves to the list of selected tables and views.

7. Click Finish.

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



8. Repeat Step 1 through Step 7 to create a second database schema, named custSchema, for the CustomerReview table.

Name the schema custSchema in Step 2 and select the CUSTOMERREVIEW table in Step 6.

Creating the Entity Beans

Now create the two entity beans, as follows:

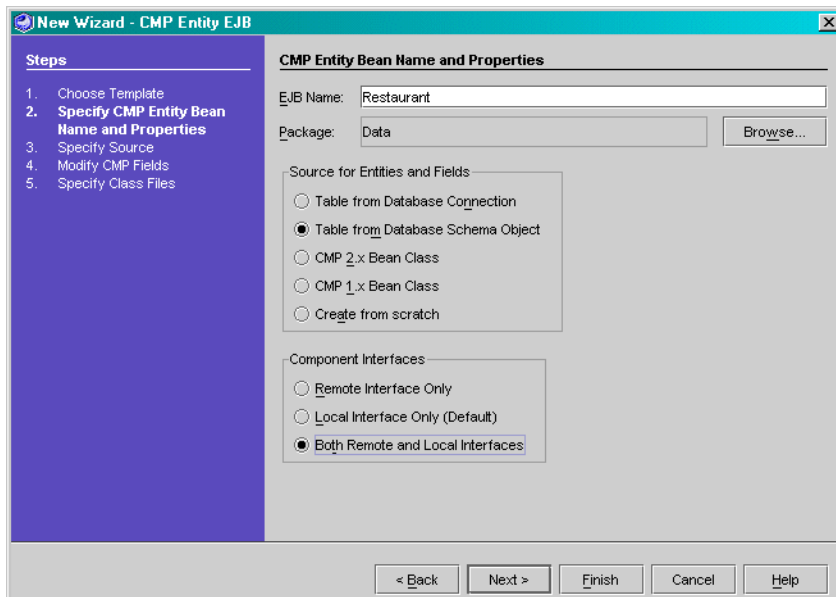
1. **Right-click the new Data package and choose New → J2EE → CMP Entity EJB.**

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

2. **Name the new CMP bean Restaurant and select the following options:**

Source for Entities and Fields: **Table from Database Schema Object**
Component Interfaces: **Both Remote and Local Interfaces**

The New wizard should look like this.

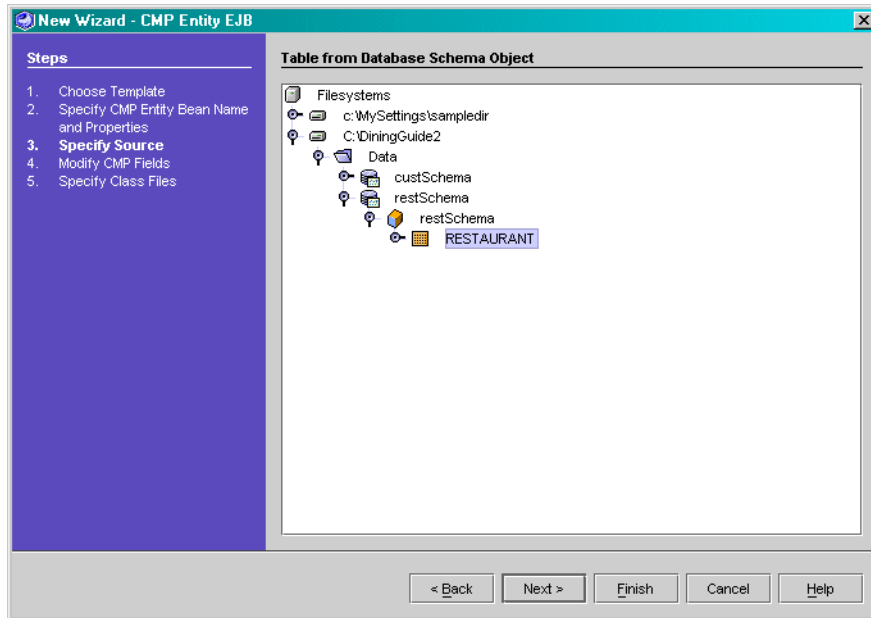


3. **Click Next.**

This displays the Table from Database Schema Object pane.

4. **Expand the DiningGuide2 node and all the nodes under the restSchema node, and select the RESTAURANT table.**

The pane looks like this:



5. Click Next.

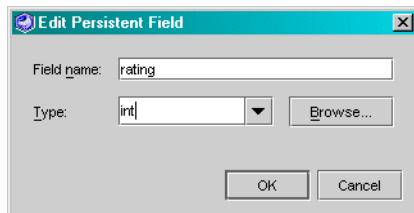
The CMP Fields pane is displayed. You see a side-by-side display of the columns of the Restaurant database table and the corresponding Java fields that the columns will be mapped to when the wizard creates the Restaurant entity bean.

6. Select the rating field and click the Edit button.

The Edit Persistent Field dialog box is displayed.

7. Delete the text for the Type field and type in int.

The dialog box looks like this:

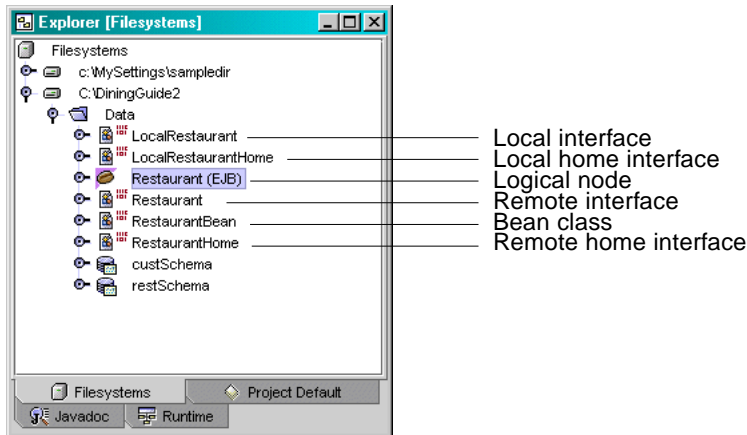


8. Click OK to close the dialog box, then click Next on the wizard's window.

The CMP Entity Bean Class Files pane 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.

9. Accept all the default labels and click 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 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.

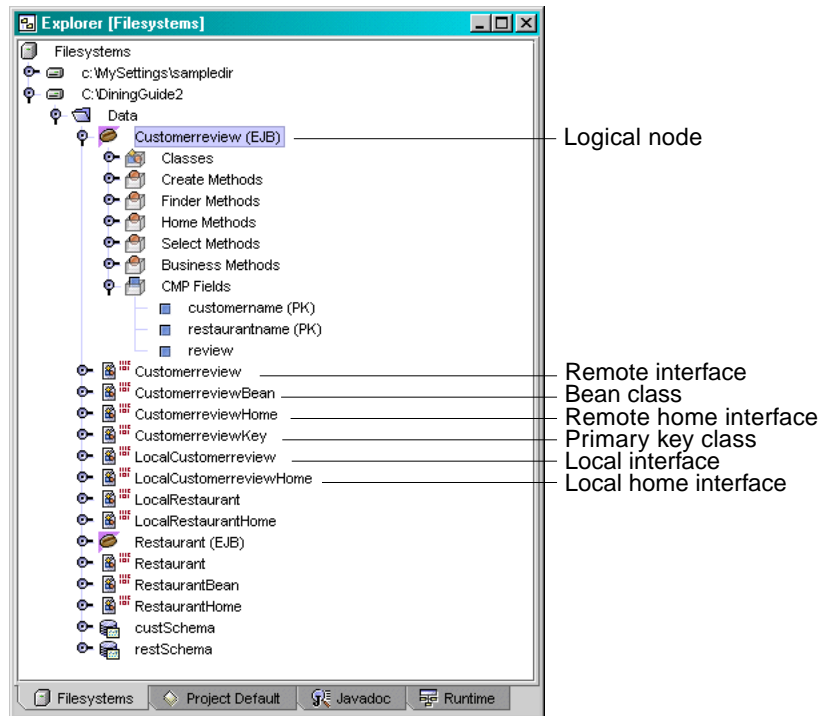
Creating the Customerreview Entity Bean

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

- 1. Right-click the new `Data` package and choose `New` → `J2EE` → `CMP Entity EJB`.**
- 2. Name the new `CMP` bean `Customerreview` and select the following options:**
Source for Entities and Fields: **Table from Database Schema Object**
Component Interfaces: **Both Remote and Local Interfaces**
- 3. Click Next.**
The Table from Database Schema Object pane is displayed.
- 4. Expand the `DiningGuide2` node and all the nodes under the `custSchema` node, select the `CUSTOMERREVIEW` table, and click Next.**

5. Click **Next** on the **CMP Fields** pane, and click **Finish** on the last pane (**CMP Entity Bean Class Files** pane).

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 1-1 in Chapter 1 to confirm the composite primary key in this table.)




6. Choose **File** → **Save All** to save your work.

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. **Using the Add button, create seven new parameters, one for each column of the Restaurant table:**

```
restaurantname (java.lang.String)
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 placed on 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, java.lang.Integer rating) throws
    javax.ejb.CreateException {
    if (restaurantname == null) {
    // Make the following two lines a single line 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, select the block of code and press Control-Shift F to reformat it properly.


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:

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

Note – As in Step 3, 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)) {
        // Make the following two lines a single line 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;
}
```

Note – If you are copying and pasting this code from a PDF version of this document, be sure to heed the comments in the code about making broken lines into a single line. The Source Editor will not correct them automatically.

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 `findAll` 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	<code>Object(o)</code>
From	<code>Restaurant o</code>

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 `findByRestaurantName` 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	<code>Object(o)</code>
From	<code>Customerreview o</code>
Where	<code>o.restaurantname = ?1</code>

(Which numeral you use 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. Expand the `Restaurant(EJB)` logical node, and then expand its `Business Methods` node.

There are no business methods yet for this entity bean.

2. 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.

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

The `Add New Business Method` dialog box is displayed.

4. Type `getRating` in the `Name` field.

5. Type `int` 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.

6. Click OK.

- 7. Expand the Restaurant (EJB) logical node, and expand the 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.

- 8. Right-click the Restaurant (EJB) logical node and choose 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.

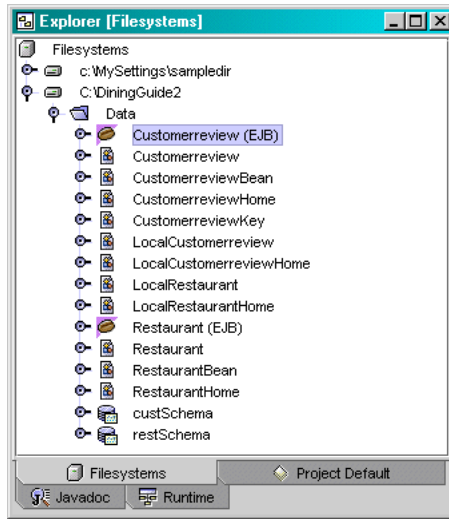
- 4. Click OK.**

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 33, 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** → **Beans** → **Java Bean**.

2. Name the new bean `RestaurantDetail` and click **Finish**.

The new bean is displayed in the Explorer.

3. Repeat Step 1 and Step 2 to create the `CustomerreviewDetail` bean.

Creating the Detail Class Properties and Their Accessor Methods

Now, add the same bean properties to the classes as those in the corresponding entity beans' CMP fields. (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.**
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 selecting the code block and 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 RestaurantDetail(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 the getter 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.
Be sure to select the class (🔍), not the bean's node. `Data.RestaurantDetail` is displayed in the Return Type field.
4. Accept all other default values and click **OK** to create the method.

5. Double-click the method to access it in the Source Editor and add the following bold code:

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

6. Select the Restaurant (EJB) logical node and press F9 to compile the code.
7. In the Explorer, right-click the Customerreview (EJB) logical node and choose Add Business Method.
8. Type getCustomerreviewDetail in the Name field.
9. For the return type, use the Browse button to select the CustomerreviewDetail class icon.
10. Accept all other default values and click OK to create the method.
11. Open the method in the Source Editor and add the following bold code:

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

12. 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

The Sun ONE Studio 4 IDE includes a mechanism for testing enterprise beans that includes the enterprise bean within a test client application that uses JavaServer Pages technology. You can display the test client pages in a web browser. These pages enable you to create instances of the bean and exercise the bean's create, finder, and business methods.

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.

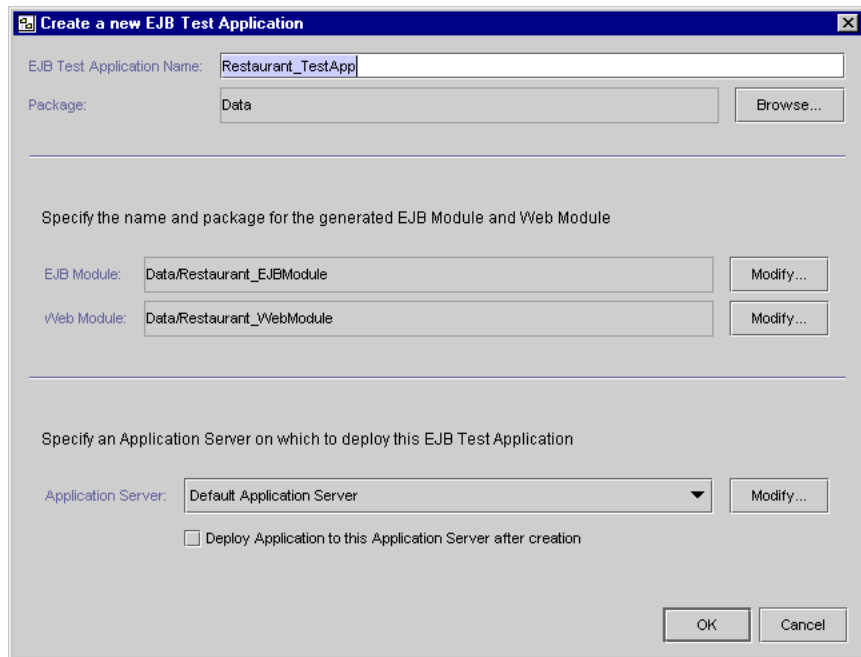
To create a test client for the Restaurant entity bean:

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:

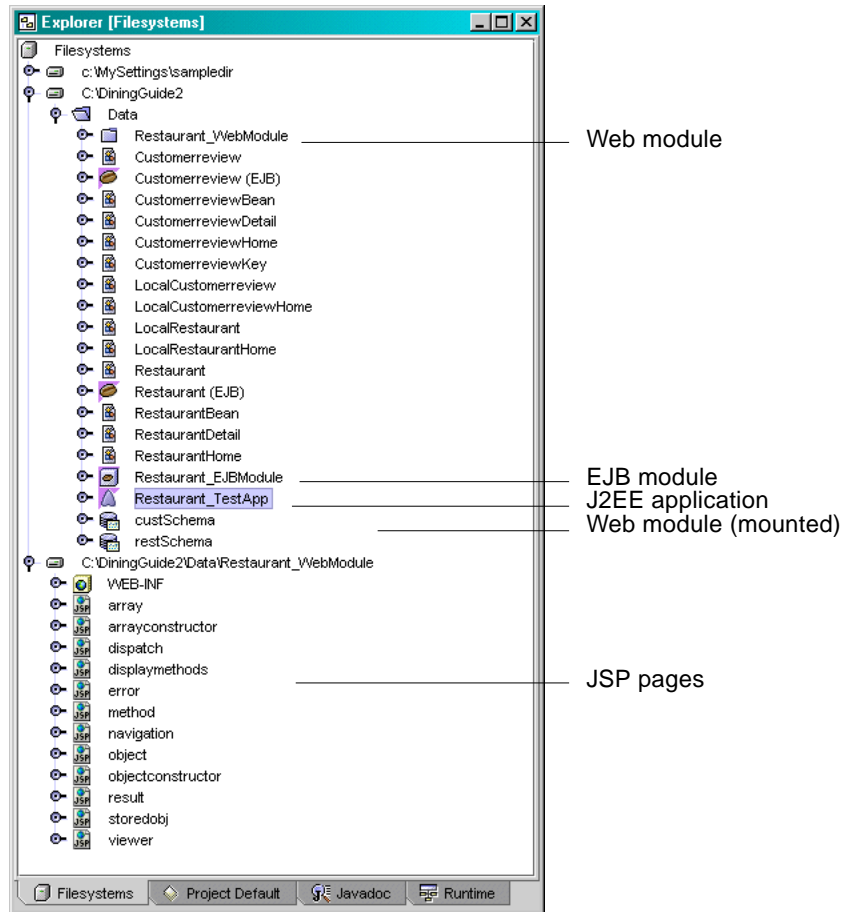


3. **Click OK.**

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. It 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.**

Note – If there is no Sun ONE AS tab on the Properties window, there is no instance of the Sun ONE Application Server 7 server in the Server Registry. See “Configuring the IDE to Use the Application Server” on page 7 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	Data/restSchema

If these values are displayed, continue with Step 5.

4. **If the values are not displayed, remap the Restaurant bean 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 pane.**

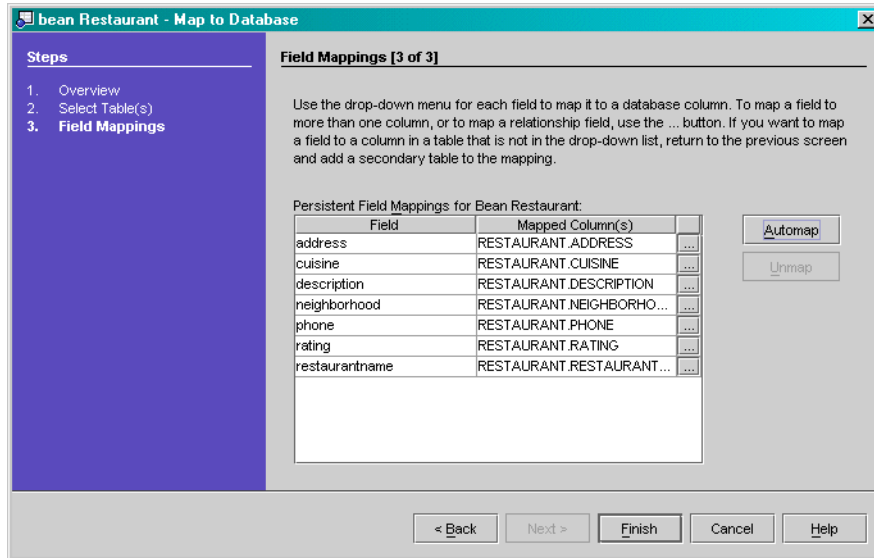
- c. **Select RESTAURANT from the drop list of the Primary Table field.**

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

- d. **Click Next to view the Field Mappings pane.**

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.

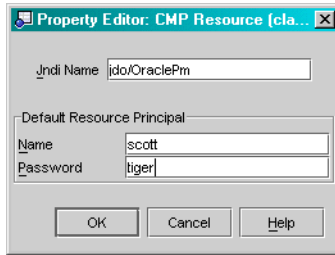
8. Click the ellipsis button to display the **CMP Resource** property editor.

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

This is the JNDI name of the JDBC Persistence Manager you defined in “Defining a JDBC Persistence Manager” on page 13.

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

You specified this name and password when you defined the connection pool in “Defining a JDBC Connection Pool” on page 9. 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 Oracle server is running before you deploy the test application, or any other J2EE application that accesses the database. In addition, make sure the Sun ONE Application Server 7 server is running and is the default application server of the IDE. See “Making the Sun ONE Application Server the Default Server” on page 8 for information.

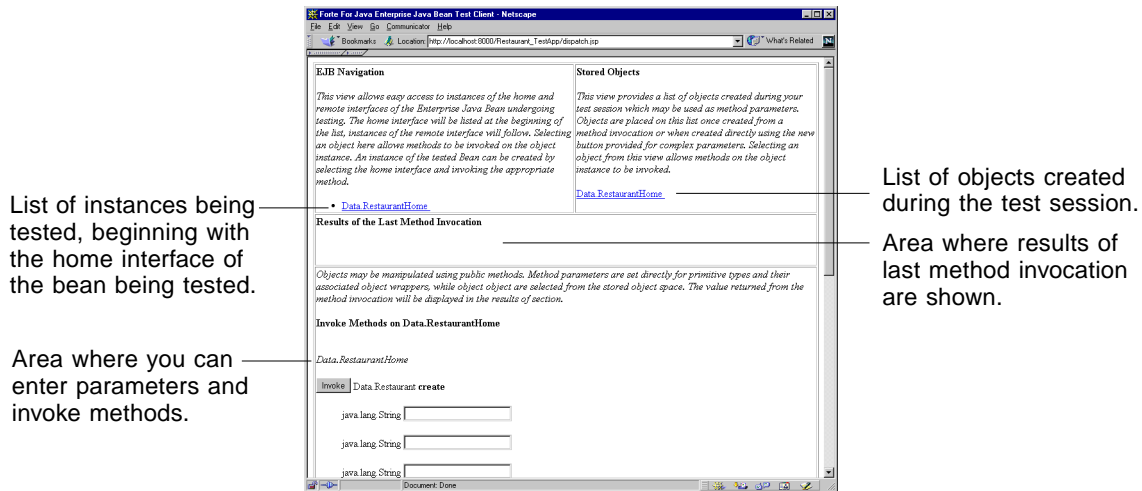
To deploy the Restaurant test application:

- **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, similar to `http://localhost/Restaurant_TestApp/` if your application server is installed locally; `localhost` will be replaced by your server's host name if it is installed remotely.

Your browser displays the test client like this:



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 3 under “Creating Create Methods for CMP Entity Beans” on page 43).

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):

Invoke Methods on Data.RestaurantHome

Data.RestaurantHome

Invoke

Data.Restaurant **create**

java.lang.String

java.lang.String

java.lang.String

java.lang.String

java.lang.String

java.lang.String

java.lang.Integer

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.

<p>EJB Navigation</p> <p><i>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.</i></p> <ul style="list-style-type: none"> • Data RestaurantHome • Joe's House of Fish 	<p>Stored Objects</p> <p><i>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.</i></p> <div> <input type="button" value="Remove Selected"/> <input type="button" value="Remove All"/> </div> <ul style="list-style-type: none"> <input type="checkbox"/> Data Restaurant Joe's House of Fish <input type="checkbox"/> java.lang.Integer 4 <input type="checkbox"/> java.lang.String Interesting variety <input type="checkbox"/> java.lang.String 510-222-3333 <input type="checkbox"/> java.lang.String 1234 Mariner Sq Loop <input type="checkbox"/> java.lang.String Alameda Island <input type="checkbox"/> java.lang.String American <input type="checkbox"/> java.lang.String Joe's House of Fish <p>Data RestaurantHome</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

The results are shown in the Results area.

<p>Results of the Last Method Invocation</p> <p>Joe's House of Fish</p> <p>Method Invoked: <i>create</i> (<i>java.lang.String,java.lang.String,java.lang.String,java.lang.String,java.lang.String,java.lang.String,int</i>)</p> <p>Parameters:</p> <p><i>Joe's House of Fish</i> <i>American</i> <i>Alameda Island</i> <i>1234 Mariner Sq Loop</i> <i>510-222-3333</i> <i>Interesting variety</i> <i>4</i></p>

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

The results area should look like this:

<p>Results of the Last Method Invocation</p> <p>size = 3</p> <p>Method Invoked: <i>findAll ()</i></p> <p>Parameters:</p> <p><i>none</i></p>

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:

```
Results of the Last Method Invocation
4
Method Invoked: getRating ()
Parameters:
none
```

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. **Start the Oracle console.**

Refer to under "Creating the Database Tables" on page 14.

2. **Copy the following SQL into the Oracle console:**

```
select * from Restaurant;
```

3. **Press Return.**

If you entered the values in Step 1 under "Using the Test Client to Test the Restaurant Bean" on page 60, the results should look like this:

```

Oracle SQL*Plus
File Edit Search Options Help

SQL*Plus: Release 8.1.5.0.0 - Production on Fri Sep 13 13:11:56 2002

(c) Copyright 1999 Oracle Corporation. All rights reserved.

Connected to:
Oracle8i Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning and Java options
PL/SQL Release 8.1.5.0.0 - Production

SQL> select * from Restaurant;

RESTAURANTNAME
-----
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
Joe's House of Fish
American         Alameda Island    1234 Mariner Sq Loop    510-222-3333
Interesting variety
4

SQL> |

```

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 55, 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 57, 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	Data/custSchema

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/custSchema`.**
 - 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 pane.**
 - e. **Select `CUSTOMERREVIEW` from the drop list of the Primary Table field.**
 - f. **Click Next to view the Field Mappings pane.**
 - 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/OraclePm` in the Jndi Name field.
9. For the Name and Password fields, type the User Name and Password for your database.
Use the name and password you used when you defined the connection pool in “Defining a JDBC Connection Pool” on page 9.
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 Oracle server is running before you deploy the test application, or any other J2EE application that accesses the database. In addition, make sure the Sun ONE Application Server 7 server is running and is the default application server of the IDE. See “Making the Sun ONE Application Server the Default Server” on page 8 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://localhost/Customerreview_TestApp/` if your application server is installed locally; `localhost` will be replaced by your server’s host name if it is installed remotely if it is installed remotely.

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. You can then start the browser manually and type in the test application’s home URL.

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

`Data.Customerreview` **create**

`java.lang.String`

`java.lang.String`

`java.lang.String`

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 `Customerreview` 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

To verify that the `Customerreview_TestApp` application inserted data in the database:

1. **Start the Oracle console.**
Refer to under “Creating the Database Tables” on page 14.
2. **Copy the following SQL into the Oracle console:**

```
select * from Customerreview;
```
3. **Press Return.**

You will see a new record of the values you entered in Step 1 under “Testing the `Customerreview` Entity Bean” on page 67.

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 4 Explorer, right-click the Data package and choose New → J2EE → Session EJB.**

The New wizard is displayed, displaying the Session Bean Name and Properties pane.

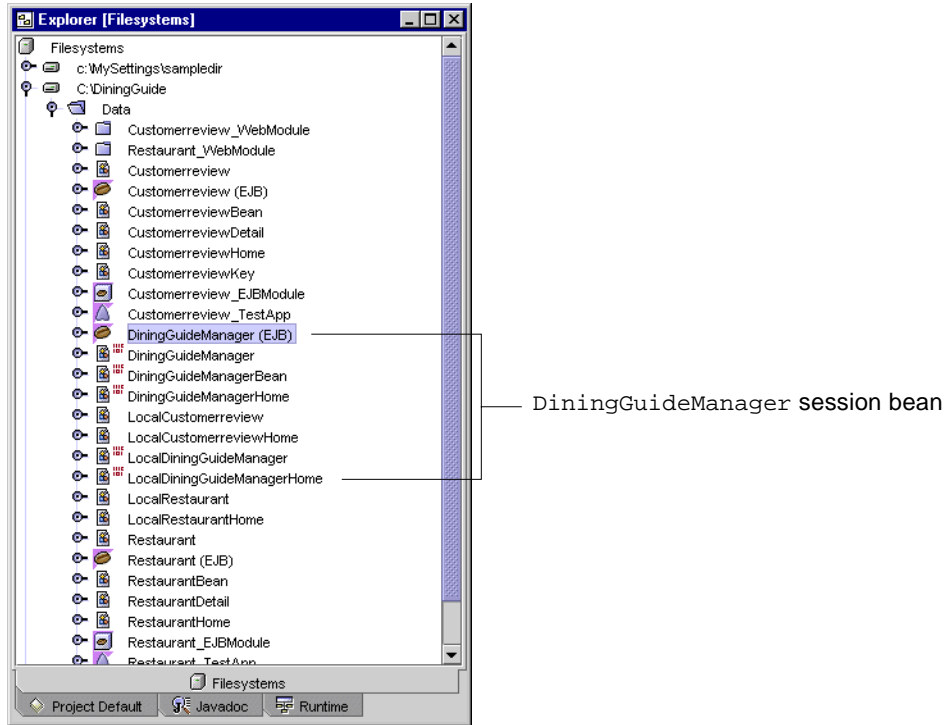
2. **Type `DiningGuideManager` in the Name field.**
3. **Select `Stateless` for the State option.**
4. **Select `Container Managed` for the Transaction Type option.**
5. **Select `Both Remote and Local Interfaces` for the Component Interfaces option.**
6. **Click Next.**

The Session Bean Class Files pane 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`.

7. 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. Double-click the `DiningGuideManager`'s create method to display it in the Source Editor.

Use the logical node (`DiningGuideManager (EJB)`) to locate the method.

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

```
public void ejbCreate(){
// Make the following two lines one line 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); }
    }
}
```

Note – Remember, you can reformat the code you enter in the Source Editor by selecting it and 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. Accept all other default values and click OK.

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

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

```
public java.util.Vector getAllRestaurants() {
// Make the following two lines a single line 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) {
// Make the following two lines a single line in the Source Editor
        System.out.println("Error in
DiningGuideManagerEJB.getAllRestaurants(): " + e);
    }
// Make the following two lines a single line in the Source Editor
    System.out.println("Leaving
DiningGuideManagerEJB.getAllRestaurants()");
    return restaurantList;
}
```

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.

6. 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.**

The Add New Parameter dialog box is displayed.

5. **Type `restaurantname` in the Field Name field.**

6. **Select `java.lang.String` in the Type field.**

7. **Click OK to close the dialog box and create the method parameter.**

8. **Accept all other default values and click OK again create the business method.**

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

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

```
public java.util.Vector
getCustomerreviewsByRestaurant( java.lang.String
    restaurantname) {
    // Make the following two lines a single line 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) {
        // Make the following two lines a single line in the Source Editor
        System.out.println("Error in
DiningGuideManagerEJB.getCustomerreviewsByRestaurant(): " + e);
    }
    // Make the following two lines a single line in the Source Editor
    System.out.println("Leaving
DiningGuideManagerEJB.getCustomerreviewsByRestaurant()");
    return reviewList;
}
```

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.

10. 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. **Click the `Add` button to add a parameter.**
The `Add New Parameter` dialog box is displayed.
5. **Type `restaurantname` in the `Field Name` field.**
6. **Type `java.lang.String` in the `Type` field.**
7. **Click `OK` to close the dialog box and create the method parameter.**
8. **Repeat Step 4 through Step 7 twice to create the following two parameters:**

```
java.lang.String customername  
java.lang.String review
```
9. **Click `OK` again create the business method.**

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

10. 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) {
// Make the following two lines a single line in the Source Editor
    System.out.println("Entering
DiningGuideManagerEJB.createCustomerreview()");
    try {
        Customerreview customerrev =
myCustomerreviewHome.create(restaurantname, customername,
review);
    } catch (Exception e) {
// Make the following two lines a single line in the Source Editor
        System.out.println("Error in
DiningGuideManagerEJB.createCustomerreview(): " + e);
    }
// Make the following two lines a single line in the Source Editor
    System.out.println("Leaving
DiningGuideManagerEJB.createCustomerreview()");
}
```

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

11. 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 53), 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 4 web services and the SOAP runtime, see *Building Web Services* in the Sun ONE Studio 4 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. **Accept all default values and click OK to create the business method and close the dialog box.**

The method is created in the DiningGuideManager session bean.

5. **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. **Click OK to create the business method and close the dialog box.**

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

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

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

6. **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.**

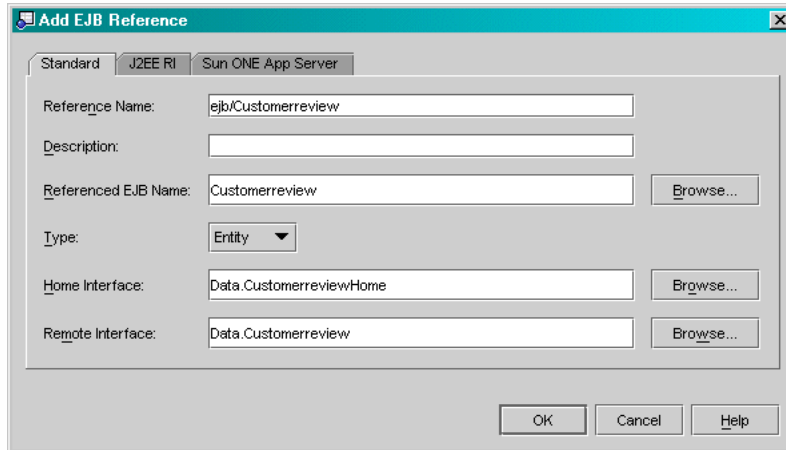
The Select an EJB browser is displayed.

8. **Select the `Restaurant (EJB)` bean under the `DiningGuide2/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:



The 'Add EJB Reference' dialog box has a title bar with a close button. It contains three tabs: 'Standard', 'J2EE RI', and 'Sun ONE App Server'. The 'Sun ONE App Server' tab is selected. The dialog has several input fields and buttons:

- Reference Name:** A text field containing 'ejb/Customreview'.
- Description:** An empty text field.
- Referenced EJB Name:** A text field containing 'Customerreview' and a 'Browse...' button to its right.
- Type:** A dropdown menu with 'Entity' selected.
- Home Interface:** A text field containing 'Data.CustomerreviewHome' and a 'Browse...' button to its right.
- Remote Interface:** A text field containing 'Data.Customerreview' and a 'Browse...' button to its right.
- At the bottom are 'OK', 'Cancel', and 'Help' buttons.

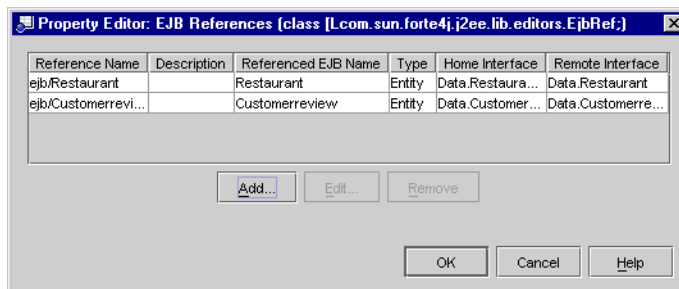
10. Select the Sun ONE App Server tab.

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. Repeat Step 5 through Step 11 to add a reference to the Customerreview entity bean.

The EJB References dialog box looks like this:



The 'Property Editor: EJB References' dialog box has a title bar with a close button. It contains a table with the following data:

Reference Name	Description	Referenced EJB Name	Type	Home Interface	Remote Interface
ejb/Restaurant		Restaurant	Entity	Data.Restaura...	Data.Restaurant
ejb/Customreviewi...		Customerreview	Entity	Data.Customer...	Data.Customerre...

Below the table are 'Add...', 'Edit...', and 'Remove' buttons. At the bottom are 'OK', 'Cancel', and 'Help' buttons.

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 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.

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

4. **Right-click the `DiningGuideManager_EJBModule` and choose Add EJB.**

The Add EJB to EJB Module browser is displayed.

5. **Expand the `DiningGuide2` filesystem and the Data package.**

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

7. Click OK.

The DiningGuideManager_EJBModule should look like this:



8. 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 57.

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 server in the Server Registry. See “Configuring the IDE to Use the Application Server” on page 7 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	Data/restSchema

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/restSchema.**
- 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 pane.**
- 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 restSchema schema.
- f. Click Next to view the Field Mappings pane.**
- 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.

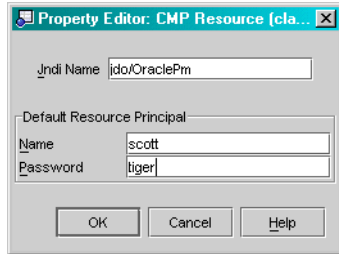
The CMP Resource property editor is displayed.

9. Type jdo/OraclePm in the Jndi Name field.

This is the JNDI name of the JDBC Persistence Manager you defined in “Defining a JDBC Persistence Manager” on page 13.

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

You specified this name and password when you defined the connection pool in “Defining a JDBC Connection Pool” on page 9. 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

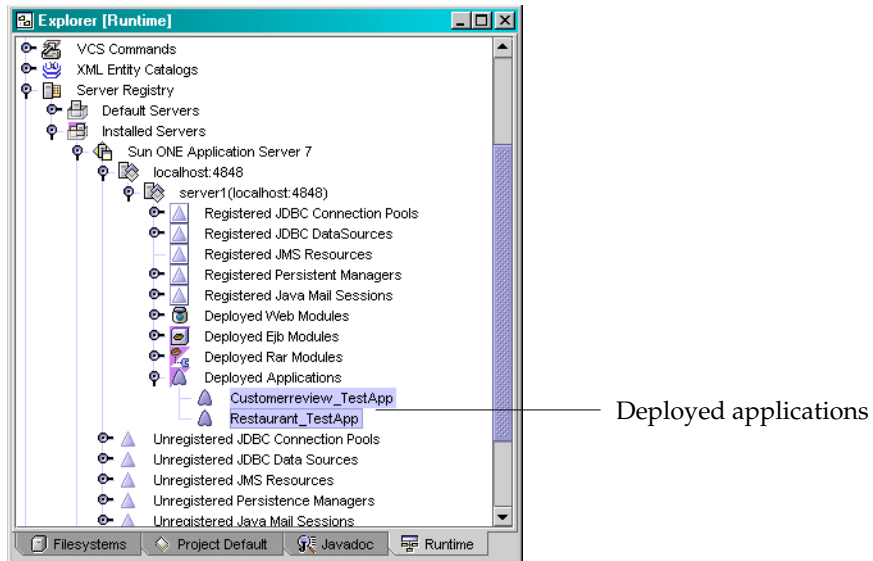
You must first undeploy the two test applications of the entity beans (if they are still deployed) before you deploy the session bean’s test application. This is because they use the same JNDI lookups to the Restaurant and Customerreview beans that are used by the DiningGuideManager_TestApp application.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

Undeploying the Entity Bean Test Applications

To undeploy any previously deployed applications.

1. Click the Explorer’s Runtime tab to display the Runtime pane.
2. Expand the `server1(hostname:host-number)` instance node under the Sun ONE Application Server 7 node under Installed Servers.
3. Select the Deployed Applications node.
4. Choose Refresh List.

The two entity bean test applications are displayed.



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

The application is undeployed.

6. Repeat Step 5 to undeploy the other application.

7. Restart the server instance.

a. Right-click the `server1(hostname:host-number)` node and choose Status.

The Server Status window is displayed.

b. Click the Stop Server Instance button.

c. When the instance is stopped, click the Start Server Instance button.

The server's command window appears.

d. When the messages in the command window indicate that the server instance is restarted, click OK to close the Server Status dialog box.

Deploying the DiningGuideManager Test Application

Note – Make sure the Oracle server is running before you deploy the test application, or any other J2EE application that accesses the database. In addition, make sure the Sun ONE Application Server 7 server is running and is the default application server of the IDE. See “Making the Sun ONE Application Server the Default Server” on page 8 for information.

To deploy the `DiningGuideManager` test application:

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

Invoke Methods on Data.DiningGuideManager [7]

Data.DiningGuideManager

Invoke	void createCustomerreview
java.lang.String	<input type="text" value="Bay Fox"/>
java.lang.String	<input type="text" value="Marcia Green"/>
java.lang.String	<input type="text" value="This is the best!"/>

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:

Results of the Last Method Invocation

void

Method Invoked: *createCustomerreview (java.lang.String,java.lang.String,java.lang.String)*

Parameters:

Bay Fox

Marcia Green

This is the best!

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 60), 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.

Results of the Last Method Invocation

[Data.RestaurantDetail@4da86b, Data.RestaurantDetail@6c14c0, Data.RestaurantDetail@468059]

Method Invoked: *getAllRestaurants ()*

Parameters:

none

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 `getCustomerreviewDetail` 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 63 and “Checking the Additions to the Database” on page 68.

You are now ready to create the web service.

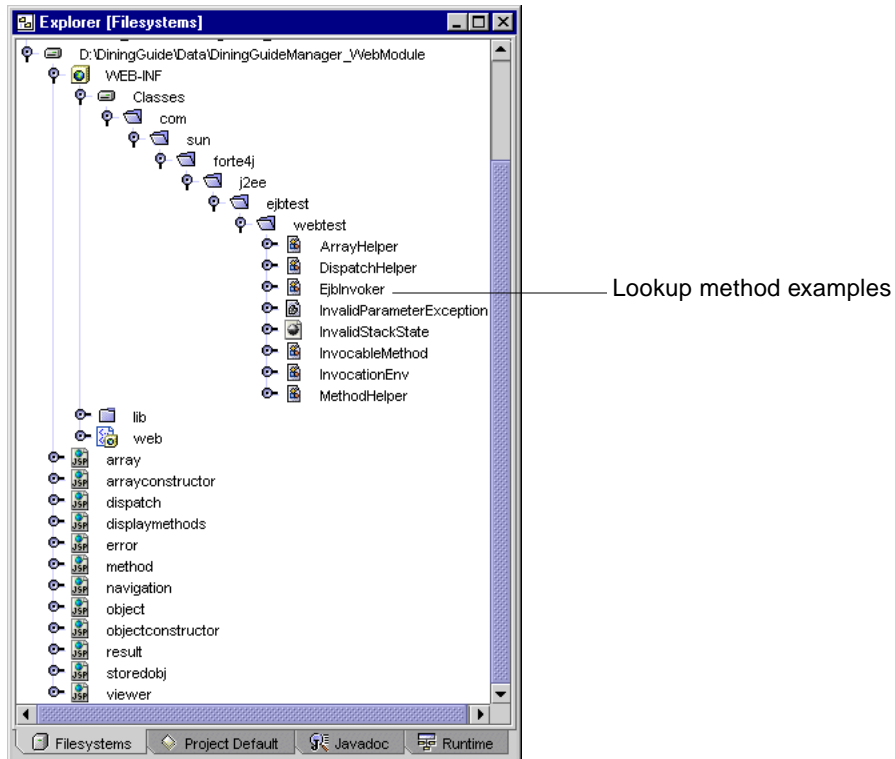
Comments on Creating a Client

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 4 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 4 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`

Creating the DiningGuide Application's Web Service

This chapter describes how to use the Sun ONE Studio 4 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 94
- "Testing the Web Service" on page 97
- "Making Your Web Service Available to Other Developers" on page 107

For a complete discussion of Sun ONE Studio 4 web service features, see *Building Web Services* from the Sun ONE Studio 4 Programming series. This book is available from the Sun ONE Studio 4 portal's Documentation page at <http://forte.sun.com/ffj/documentation/index.html>. For information on specific features, see the Sun ONE Studio 4 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
- A test client proxy

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 4 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 Proxy Pages

When you create a client and generate the client proxy, 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 proxy pages 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 proxy.
2. The SOAP client proxy 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 proxy.
6. The SOAP client proxy passes the response to the JSP page, which displays the response on a web page.

Creating the Tutorial's Web Service

Create the tutorial's web service with the following tasks:

- "Creating the Web Service Module," which follows
 - Use the IDE's Web Service wizard to create the logical web service and specify the methods you want to reference.
- "Specifying the Web Service's SOAP RPC URL" on page 96
 - This URL is used at runtime to access your web service.
- "Generating the Web Service's Runtime Classes" on page 97
 - This task generates the supporting EJB components that are used for testing and implementing the web service.

Creating the Web Service Module

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 web service module:

1. **In the Explorer, right-click the mounted `DiningGuide2` 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 `DiningGuide2` directory.

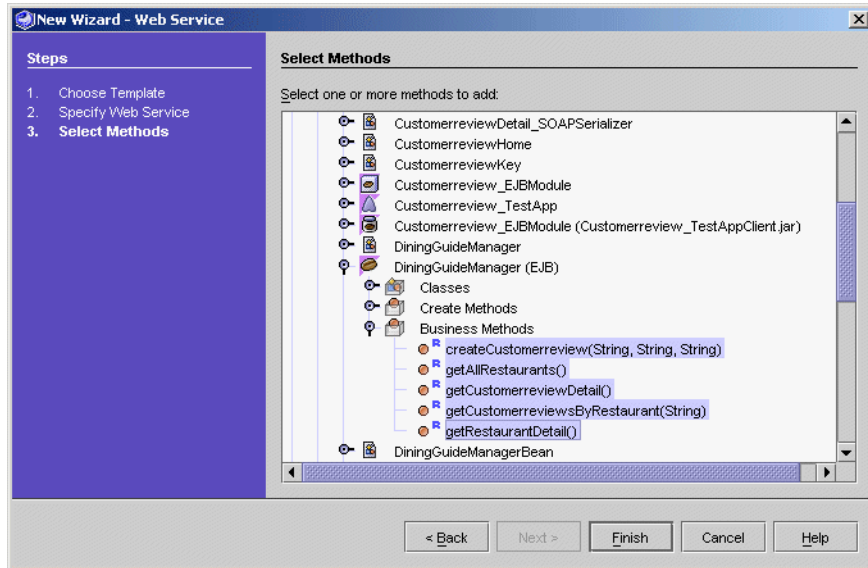
3. **Right-click the `WebService` package and choose `New` → `Web Services` → `Web Service`.**

The New wizard displays the Web Service pane.


4. **Type `DGWebService` in the `Name` field, make sure the `Multitier` option is selected for the `Architecture type`, and click `Next`.**

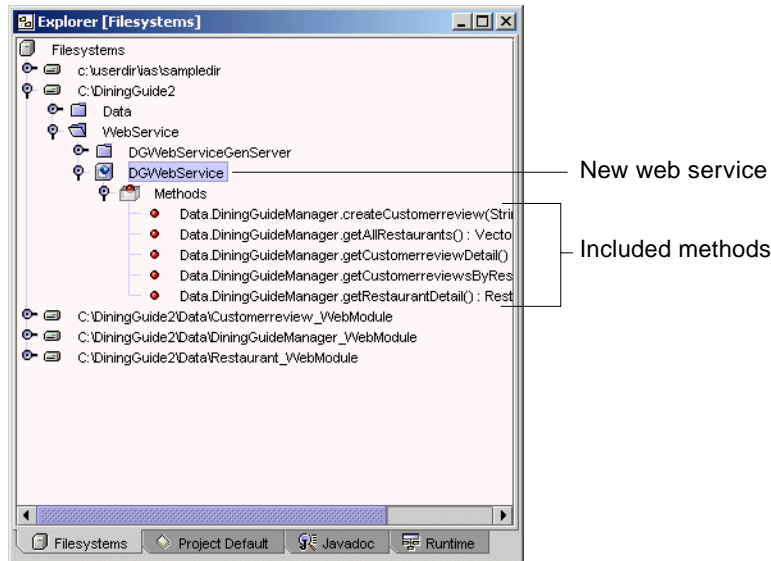
The Methods pane of the New wizard is displayed.

5. **Expand the Data, DiningGuideManager(EJB), and Business Methods nodes.**
6. **Use Control-Click to select all the DiningGuideManager's business methods:**
The Methods pane looks like this:



7. **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:



Specifying the Web Service's SOAP RPC URL

The SOAP RPC URL property locates the SOAP `rpcrouter` Servlet of the JAX-RPC runtime on the application server. This property includes a string called the *context root* or *web context*. This string must match the web context property of the J2EE application WAR node that you will create later in “Specifying the Web Context Property” on page 100.

To set the SOAP RPC URL property:

- 1. Display the properties of the `DGWebService` node.**

Select the `DGWebService` node and view the properties in the Properties window. If the Properties window is not displayed, choose View → Properties.

- 2. Display the property editor for the SOAP RPC URL property.**

Click once in the value field, then click the ellipsis button that appears to display the editor.

- 3. Change the first instance of the string `DGWebService` in the URL to `DGWSContext`, so that the entire URL is:**

```
http://localhost:80/DGWSContext/DGWebService
```

Note – 80 is the Sun ONE Application Server’s default port for HTTP communication. If you supplied a different number for the HTTP port when you installed the application server, substitute that number in the URL.

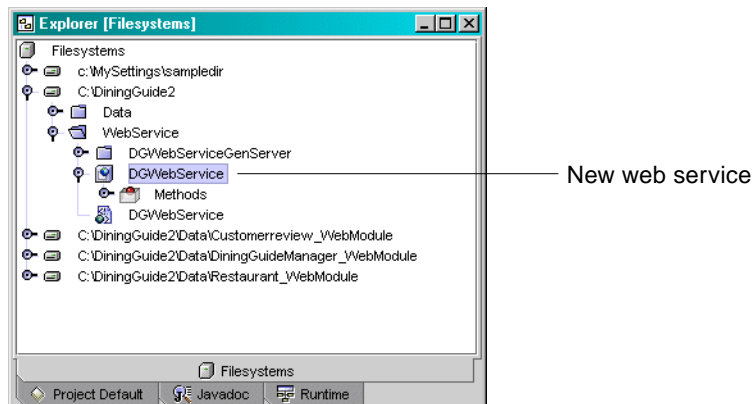
Generating the Web Service’s Runtime Classes

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 four classes to implement the web service, three of which are for a generated EJB component.

To generate a web service’s runtime classes:

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

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:



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. Specify the web context property of the web service WAR file
3. Deploying the test application
4. 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.

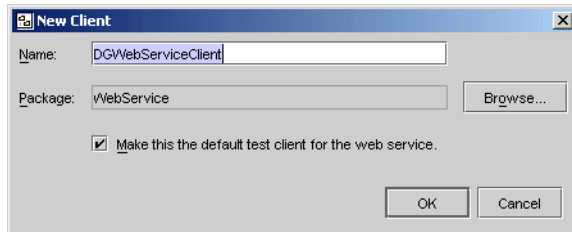
Creating a Test Client and Test Application

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 **New Client**.**
The New 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** → **J2EE** → **Application**.**
The New wizard is displayed.

4. Type DiningGuideApp in the Name field and click Finish.

The new J2EE application node () appears under the WebService package. Now add the web service to the application.

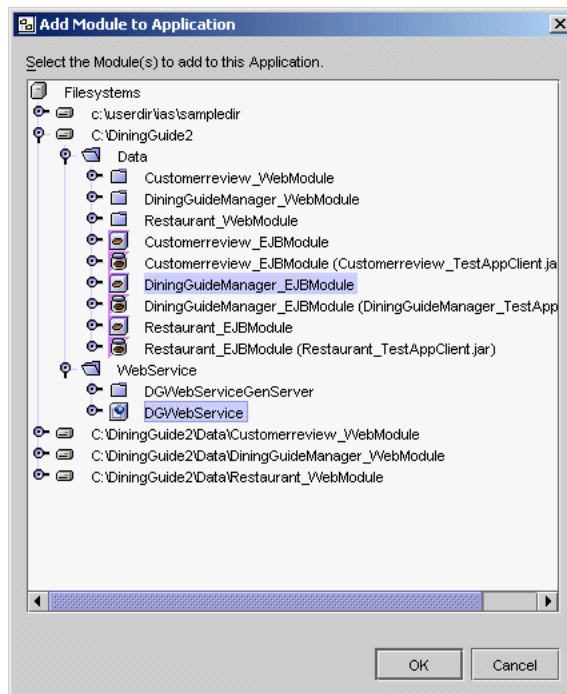
5. Right-click the DiningGuideApp node and choose Add Module.

The Add Module to Application dialog box appears.

6. In the dialog box, expand the DiningGuide filesystem and both the Data and WebService packages

7. Using Control-Click, select both the DiningGuideManager_EJBModule and the DGWebService nodes.

The dialog box looks like this:



8. Click OK to accept the selection and close the dialog box.

9. In the Explorer, expand the DiningGuideApp J2EE application.

Both the DGWebService's WAR and EJB JAR files have been added to the application, as well as the DiningGuideManager_EJBModule:



Specifying the Web Context Property

Now specify a web context for the new J2EE application in the web service's WAR file. This must be the same context that you specified in "Specifying the Web Service's SOAP RPC URL" on page 96.

1. **Display the Properties window of the `DGWebService_War` file inside the `DiningGuideApp` application.**

Select the node and view the properties in the Properties window. If the Properties window is not already displayed, choose View → Properties.

2. **In the Web Context field, type `DGWSContext` for the property value.**

3. **Choose File → Save All.**

You are now ready to deploy the `DiningGuideApp` test application.

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 84. (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 84.

Note – Make sure the Oracle server is running before you deploy the test application or any other J2EE application that accesses the database. In addition, make sure the Sun ONE Application Server 7 server is running and is the default application server of the IDE. See "Making the Sun ONE Application Server the Default Server" on page 8 for information.

To deploy the `DiningGuideApp` application:

1. **In the Explorer, right-click the `DiningGuideApp` node () and choose Deploy.**

A progress monitor window shows the deployment process running. You might see a message and a prompt like the following:

```
....implements the java.rmi.Remote interface. Do you wish to  
mark it as an RMI object?
```

If this appears, respond with No To All.

2. 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 window of the Explorer displays a `server1 (hostname: hostport)` node.

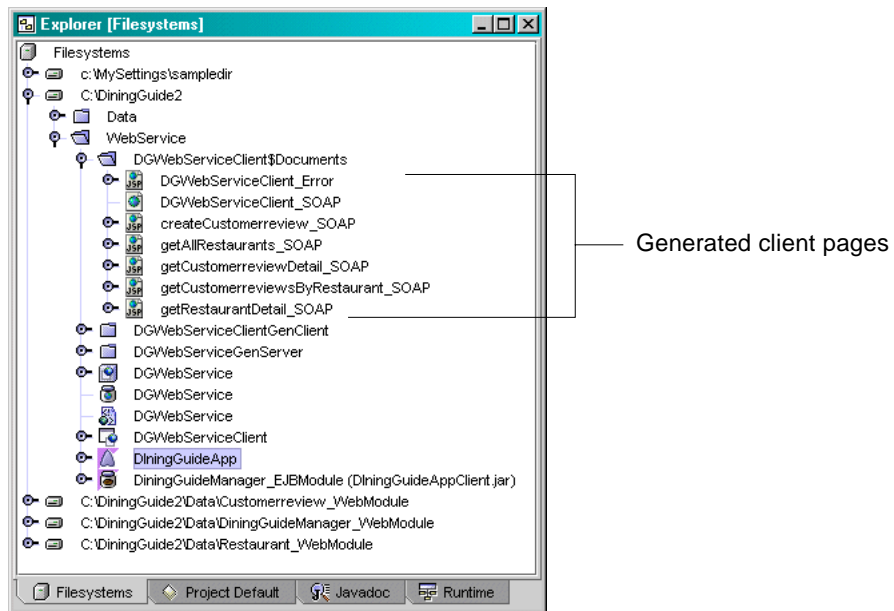
3. Click the Editing tab of the IDE to return to the Explorer.

4. Expand the `DGWebServiceClient$Documents` node under the `WebService` package.

The following supporting items have been created:

- A JSP page for each method
- An HTML welcome page
- A JSP error page

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 4 portal's Documentation page at <http://forte.sun.com/ffj/documentation/index.html>.

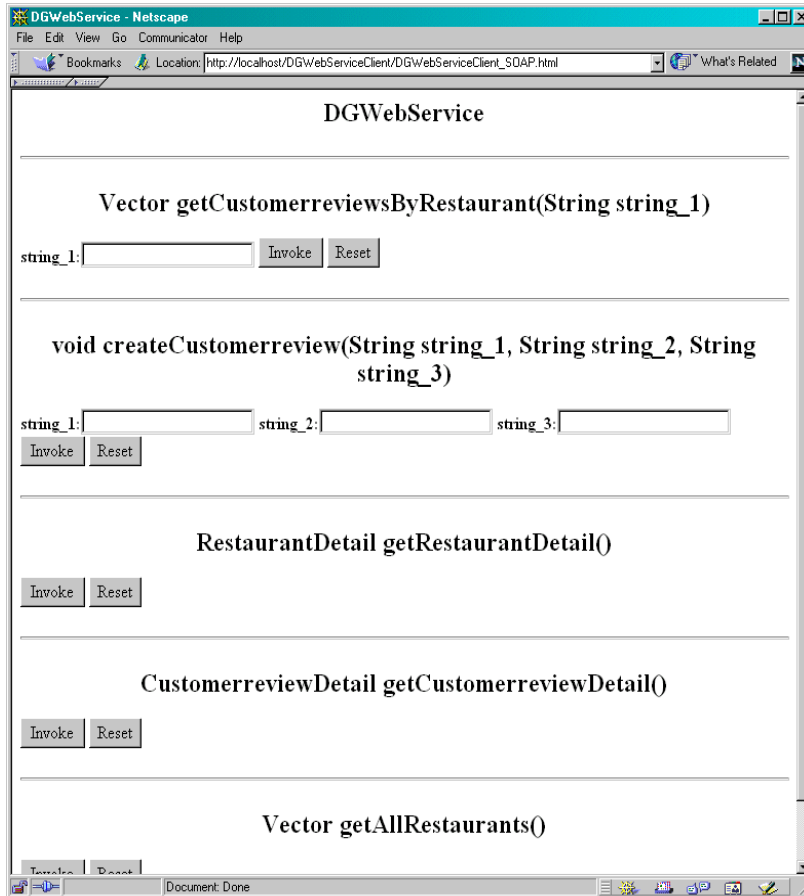
To test the web service:

1. **In the Explorer, right-click the `DGWebServiceClient` node  and choose **Deploy**.**

Check the messages in the Output window for completion of the deployment process.

2. **Right-click the `DGWebServiceClient` node again, and choose **Execute**.**

The IDE automatically starts the Sun ONE Application Server 7 web server, launches the default web browser, and displays the client's generated welcome page (`DGWebServiceClient_SOAP.html`):



This page lets you to test whether the operations work as expected.

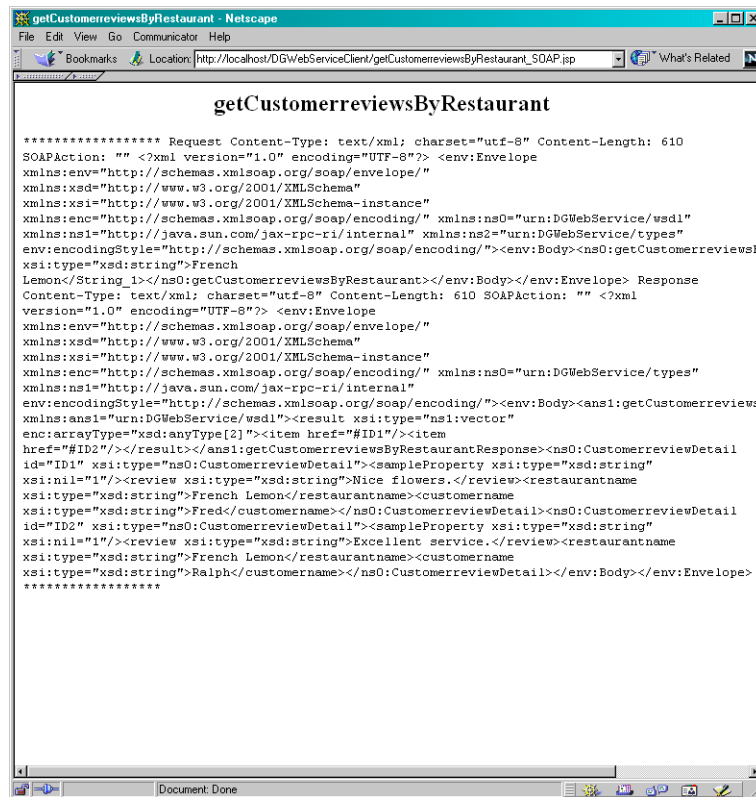
3. **Test the `getCustomerreviewsByRestaurant` by typing French Lemon in the text field and clicking the Invoke button.**

Vector `getCustomerreviewsByRestaurant(String string_1)`

string_1:

A SOAP message is created and sent to the application server. The DiningGuideApp web service turns the SOAP message into a method invocation of the `DiningGuideManager.getCustomerreviewsByRestaurant` method. This method returns a collection that the generated JSP page,

getCustomerreviewsByRestaurant_SOAP.jsp, displays as a collection of customer review data. The XML wrapper containing the return value is displayed as shown.



The data includes all the records entered for the French Lemon restaurant. Refer to TABLE 1-3 to verify the data. Or you can verify the data by starting the Oracle console and running the following SQL statement:

```
select * from CustomerReview;
```

The results show what CustomerReview records you have entered.

4. Use the Back button on your browser to return to the welcome page.
5. 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:

string_2:

string_3:

6. 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 to 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:



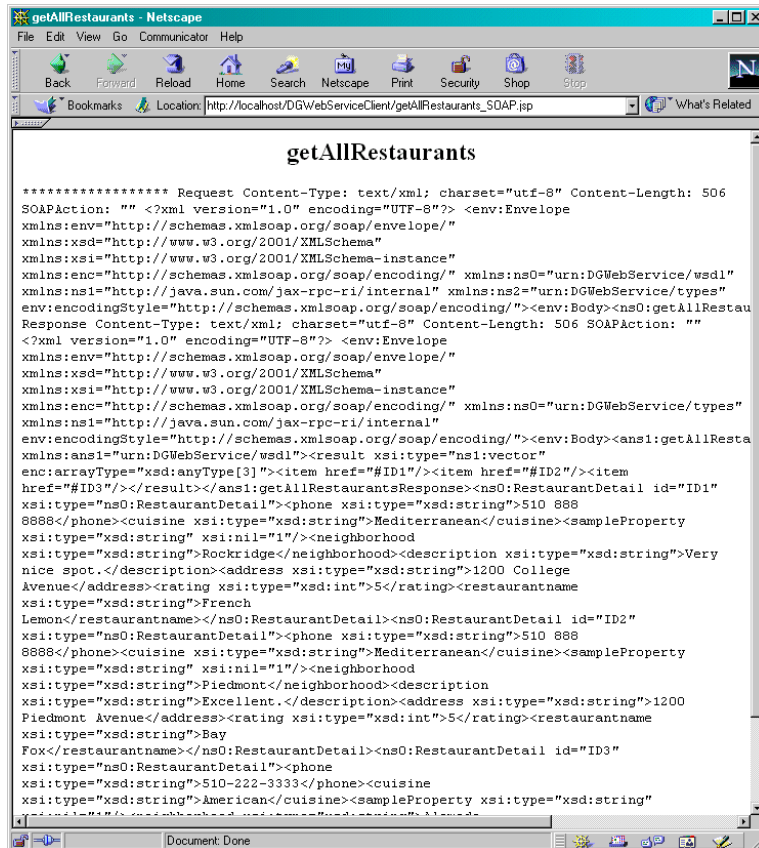
7. Use the Back button on your browser to return to the welcome page.

8. Test the `getAllRestaurants` operation by clicking its `Invoke` button on the welcome page.

Vector `getAllRestaurants()`

[Invoke](#) [Reset](#)

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 60) 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 proxy 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 a client proxy
 - Builds the client around the client proxy
 - Specifies the web service URL as the SOAP RPC URL property of the client proxy

Generating the client proxy 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 on your operating system's file system, named DGWebService.wsdl.

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 a Client Proxy 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 proxy from the WSDL file:

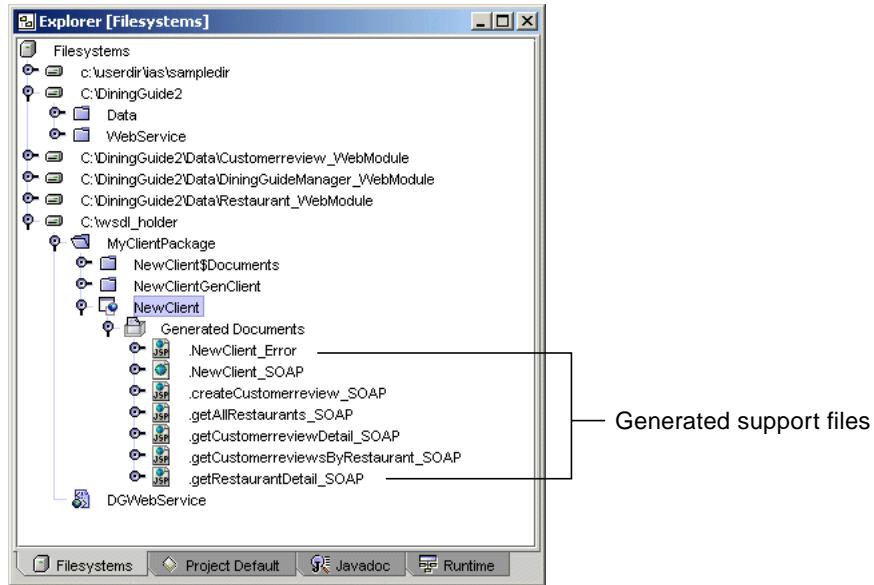
1. **On your operating system's file system, create a directory and place the DGWebService.wsdl file in it.**
2. **In the Sun ONE Studio 4 Explorer, choose New → Mount Filesystem.**
The New wizard is displayed.
3. **Select Local Directory and click Next.**
The Select Directory pane of the New wizard is displayed.
4. **Find the directory you created in Step 1 and click Finish.**
The directory is mounted in the Explorer.
5. **Right-click the new directory and choose New → Java Package.**
6. **Type MyClientPackage in the Name field and click Finish.**
MyClientPackage is displayed in the mounted directory.
7. **In the Explorer, right-click MyClientPackage and choose New → Web Services → Web Service Client.**
The New wizard is displayed.
8. **Type NewClient in the Name field.**
9. **Make sure the package is the MyClientPackage package.**
10. **For the Source, select the Local WSDL File option and click Next.**
The Select Local WSDL File pane of the New wizard is displayed.

11. Expand the directory you created in Step 1 and the `MyClientPackage` package. Select the `DGWebService WSDL` file and click **Finish**.

A new client node () appears in the Explorer.

12. Right-click the `NewClient` client node and choose **Generate Client Proxy**.

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 102.

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 4 provides a single-user internal UDDI registry to test this process, and the `StockApp` example, available from the Sun ONE Studio 4 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*.

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 proxy 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 114
- “Examining the Client Code” on page 116

Creating the Client With the Provided Code

The client classes are provided in two forms: source code and Java class files.

If you are using a Microsoft Windows system, you can create two classes in the IDE and replace their default code with the code you copy and paste from Appendix A. Code from these classes instantiates the client proxy, which is assumed to be in the same package. Therefore, create the client classes within the `WebService` package.

If you are using Solaris or Linux environments, copying code from a PDF file into the source editor results in an unformatted single line of code, which is rather inconvenient to read. Instead, use the class files provided in the `DiningGuide2.zip` file, which you can download from <http://forte.sun.com/ffj/documentation/tutorialsandexamples.html>

Creating the Client on Microsoft Windows Systems

To create the two classes using by copying the source in Appendix A:

1. **In the Explorer, right-click the `WebService` node and choose `New → Classes → Class`.**

The New wizard is displayed.

2. **Name the class `RestaurantTable` and click `Finish`.**

The new `RestaurantTable` classes is created under the `WebService` package.

3. **Repeat Step 1 and Step 2 to create the `CustomerReviewTable` class.**

4. **Open the classes in the Source Editor and delete all the default code from each class.**

5. **Copy all the code from “`RestaurantTable.java Source`” on page 143 and the following three pages and paste it into the body of the `RestaurantTable` class.**

Tip – Copy this long code very carefully. Set your Acrobat Reader to display a whole page at a time. Select all the code from the first page of `RestaurantTable` code and paste it into the target file in the Source Editor. At the end of the pasted code, press Enter to start a new line. Then copy all the code from the next page of `RestaurantTable` code and paste it into the Source Editor, starting at the new line you created previously. Repeat until all the code is pasted.

6. **Select all the pasted code in the Source Editor and press `Control-Shift F`.**

This action reformats all the code correctly.

7. **Repeat Step 5, copying all the code from “`CustomerReviewTable.java Source`” on page 146 and the following three pages into the `CustomerReviewTable` class body.**

8. **Repeat Step 6 to format the pasted code properly.**

9. **Right-click the `CustomerReviewTable` node and choose `Compile`.**

The `CustomerReviewTable` class should compile without errors.

10. **Right-click the `RestaurantTable` class node and choose `Compile`.**

The `RestaurantTable` class should compile without errors.

If you examine the code in the `RestaurantTable` and `CustomerReviewTable` classes, there are several comments warning against modifying some sections. These sections are Swing component code created in the Form Editor and should not be

modified in the Source Editor. Normally, such restricted code has a blue background. If you restart the IDE, the source for this file will have a blue background for the restricted areas, and you will not be able to edit the code in those sections.

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.

Creating the Client in Solaris or Linux Environments

To copy the two provided Java client classes into the `WebService` package:

1. **Unzip the `DiningGuide2.zip` file from the Developer Resources portal.**

The portal is at

<http://forte.sun.com/ffj/documentation/tutorialsandexamples.html>

For example, unzip the file to the `/MyZipFiles` directory.

2. **Using a file system command, copy the two client files from the `DiningGuide2` source files to the `WebServices` package.**

For example, type:

```
$ cp /MyZipFiles/DiningGuide2/WebService/*.java /DiningGuide2/WebService
```

3. **In the IDE’s Explorer, expand the `DiningGuide2/WebService` package and observe the two new classes.**

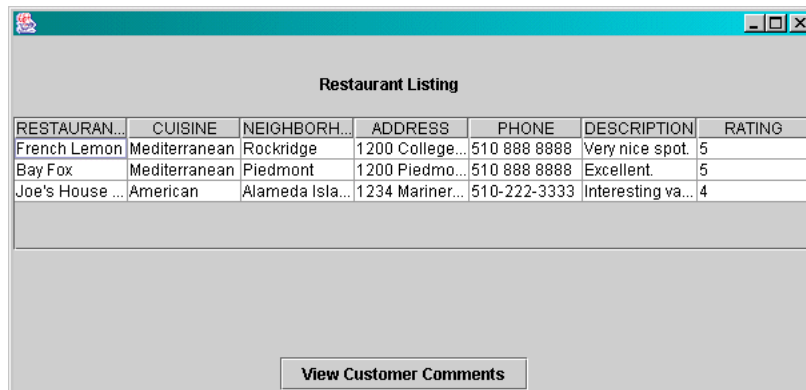
Note – When you create a Swing client in the IDE’s Form Editor, the IDE generates a `.form` file and a `.java` file. The `.form` file enables you to edit the GUI components in the Form Editor. However, the `.form` files have not been provided for the `RestaurantTable` and `CustomerReviewTable` classes, which prevents you from modifying the GUI components in the Form Editor.

Running the Tutorial Application

Run the DiningGuide application by executing the RestaurantTable class, as follows:

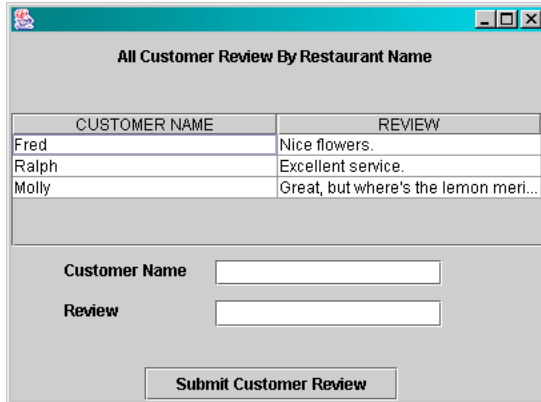
1. In the IDE, click the **Runtime** tab of the Explorer.
2. **Expand the** Server Registry, **the** Installed Servers, **the** Sun ONE Application Server 7, **and its subnodes.**
3. **Right-click the** Deployed Applications **subnode of the server instance.**
4. **Make sure the** DiningGuideApp **application 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 100.**
6. **Right-click the server instance and choose Status to verify that the instance is running.**
If the Stop Server Instance button is activated, the server is running. If it is not activated, click the Start Server Instance button to start it.
7. **Click the Filesystems tab of the Explorer.**
8. **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:



9. 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.



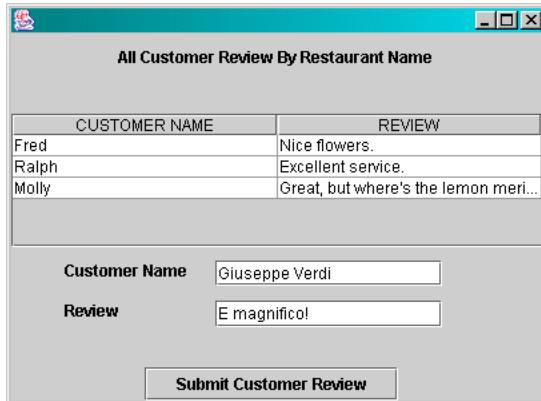
The screenshot shows a window titled "All Customer Review By Restaurant Name". It contains a table with two columns: "CUSTOMER NAME" and "REVIEW". The table has three rows of data:

CUSTOMER NAME	REVIEW
Fred	Nice flowers.
Ralph	Excellent service.
Molly	Great, but where's the lemon meri...

Below the table, there are two input fields labeled "Customer Name" and "Review", and a "Submit Customer Review" button.

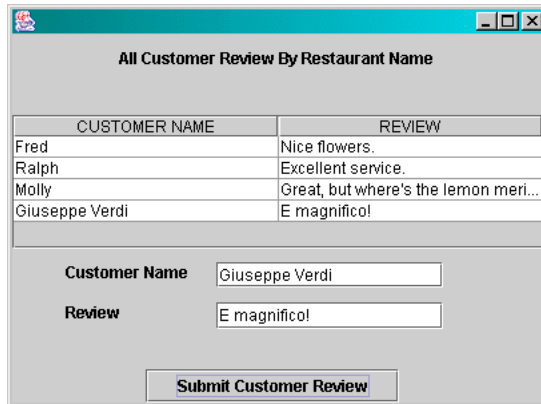
10. Type a something in the Customer Name field and in the Review field and click the Submit Customer Review button.

For example:



The screenshot shows the same window as before, but now the "Customer Name" field contains "Giuseppe Verdi" and the "Review" field contains "E magnifico!". The "Submit Customer Review" button is still present.

The record is entered in the database and is displayed on the same `CustomerReviewTable` window, as shown:



11. Play around with the features, as described in “User’s View of the Tutorial Application” on page 19.

12. Quit the application by closing any window.

After you quit the application, the execution window shows that the Sun ONE Application Server 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 Tomcat 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 proxy 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 117
- “Displaying Customer Review Data for a Selected Restaurant” on page 118
- “Creating a New Customer Review Record” on page 121

Displaying Restaurant Data

Displaying restaurant data is accomplished by the `RestaurantTable` class's methods, which instantiate the client proxy and call its `getAllRestaurants` method, as follows:

1. `RestaurantTable.getAllRestaurants` method instantiates the client proxy, calls the client proxy'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 proxy'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 proxy (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.jButtonActionPerformed` method then displays the data.

```
private void jButtonActionPerformed(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 `jButtonActionPerformed` method creates the review record in the database by means of the proxy'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.jButtonActionPerformed method instantiates a new client proxy (if required) and calls its createCustomerreview method, passing it the restaurant name, the customer name, and the review data.

```
private void jButtonActionPerformed(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 (jButtonActionPerformed) calls the CustomerReviewTable.refreshView method.

```
private void jButtonActionPerformed(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();
    }
}
```


DiningGuide Source Files

This appendix displays the code for the following DiningGuide components:

- EJB tier components:
 - “RestaurantBean.java Source” on page 126
 - “RestaurantDetail.java Source” on page 129
 - “CustomerreviewBean.java Source” on page 134
 - “CustomerreviewDetail.java Source” on page 136
 - “DiningGuideManagerBean.java Source” on page 139
- Client components:
 - “RestaurantTable.java Source” on page 143
 - “CustomerReviewTable.java Source” on page 146

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 4 Source Editor, all formatting is lost. To automatically reformat the code in the Source Editor, select the code you want to reformat, then press Control-Shift F.

Solaris and Linux users are advised not to copy from these files, because the Source Editor does not read the CRs 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) {
            // Make the following two lines a single line 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

```
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)) {
            // Make the following two lines a single line 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

```

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 crcresult = 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 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) {
        // Make the following two lines a single line in the Source Editor
        System.out.println("Error in
DiningGuideManagerEJB.getCustomerreviewsByRestaurant(): " + e);
    }
    // Make the following two lines a single line 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) {
            // Make the following two lines a single line in the Source Editor
            System.out.println("Error in
DiningGuideManagerEJB.createCustomerreview(): " + e);
        }
        // Make the following two lines a single line in the Source Editor
        System.out.println("Leaving
DiningGuideManagerEJB.createCustomerreview()");
    }
}

```

```

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

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

public java.util.Vector getAllRestaurants() {
    // Make the following two lines a single line 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) {
        // Make the following two lines a single line in the Source Editor
        System.out.println("Error in
DiningGuideManagerEJB.getAllRestaurants(): " + e);
    }
    // Make the following two lines a single line in the Source Editor
    System.out.println("Leaving DiningGuideManagerEJB.getAllRestaurants()");
    return restaurantList;
}
}

```

RestaurantTable.java Source

```
package WebService;
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
};
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

```

package WebService;
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.setViewportViewView(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) { //GEN-
FIRST: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) { //GEN-FIRST: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;
}

```

DiningGuide Database Script

This Oracle database script for the DiningGuide tutorial is as follows:

```
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;
```

Index

A

- accessor methods, exposing to the user, 48
- Add Business Method menu item, 48
- Add Constructor menu item, 52
- Add Create Method menu item, 43
- Add Finder Method menu item, 46
- Add Module menu item, 99

B

- business methods
 - Customerreview
 - getCustomerreviewDetail, 54
 - getReview, 49
 - DiningGuideManager
 - createCustomerreview, 76, 87
 - getAllRestaurants, 72, 87
 - getCustomerreviewDetail, 78, 87
 - getCustomerreviewsByRestaurant, 74, 88
 - getRestaurantDetail, 78, 88
 - Restaurant
 - getRating, 48, 63
 - getRestaurantDetail, 53
 - Restaurant.getRestaurantDetail, 34
- business methods, Swing client
 - CustomerReviewTable
 - getCustomerReviewByName, 120
 - jButton1ActionPerformed, 122
 - putDataToTable, 119, 120, 123
 - refreshView, 122, 124

- RestaurantTable

- getAllRestaurants, 117
 - jButton1ActionPerformed, 119, 121
 - putDataToTable, 118

C

- client proxy, 92
- client proxy methods
 - createCustomerreview, 122
 - getAllRestaurants, 106, 117
 - getCustomerreviewsByRestaurant, 103, 120
- constructors
 - CustomerreviewDetail, 53
 - RestaurantDetail, 52
- create methods
 - Customerreview.create, 44, 60, 67
 - DiningGuideManager.create, 70 to 72, 86
 - JNDI lookup code in, 71
 - Restaurant.create, 43, 61, 67
- Create New EJB Test Application menu item, 55, 81
- Creating a web service, 27
- creating a web service, 94 to 97
- Customerreview entity bean
 - create method, creating, 44
 - creating, 37 to 50
 - getReview method, creating, 49
 - testing, 64
- CustomerReview table, description, 15

- Customerreview_TestApp, 65
 - bean methods, testing, 67 to 68
 - creating, 64 to 66
 - deploying, 66
 - undeploying, 84 to 85
- CustomerReviewTable
 - creating, 112
 - displayed, 20, 115

D

- Deploy menu item, 100
- deploying
 - adding Sun ONE Application Server 7 properties for entity beans, 57 to 59, 65 to 66
 - adding Sun ONE Application Server 7 properties for session beans, 82 to 84
 - test applications for entity beans, 59, 66
 - test applications for session beans, 86
 - undeploying with the IDE, 84 to 85
- detail classes
 - creating, 50 to 53
 - description, 26, 33
- DGWebService, 94
- DiningGuide application
 - application scenarios, 18
 - architecture, 22
 - deploying, 28, 100
 - EJB tier, 31 to 35
 - functional description, 17
 - functional specs, 19
 - limitations, 29
 - requirements, 2
 - Swing client, adding to the application, 111 to 112
 - Swing client, examining, 116 to 124
 - Swing client, executing, 114
 - user's view, 19
 - zipped source files, 14
- DiningGuide Swing client
 - executing, 19
 - generated from web services, 102
 - installing and using, 28
- DiningGuide2
 - Microsoft Windows limitation on application location, 37
- DiningGuideApp, 102

- DiningGuideManager session bean
 - create method, coding, 70 to 72
 - createCustomerreview method, 64, 76 to 77, 87, 104
 - creating, 69
 - getAllRestaurants method, 72 to 73, 87
 - getCustomerreviewDetail method, 78, 87
 - getCustomerreviewsByRestaurant method, 74 to 75, 88
 - getRestaurantDetail method, 78, 88
 - testing, 86 to 88
- DiningGuideManager_TestApp
 - bean methods, testing, 86 to 88
 - creating, 81 to 84
 - deploying, 86

E

- EJB Builder
 - entity beans, creating, 36 to 42
 - local or remote interfaces, 36, 69
 - session beans, creating, 69 to 70
 - using, 25
- EJB QL
 - using in finder methods, 46
- EJB tier overview, 24, 31 to 35
- entity beans
 - adding to an EJB module, 81
 - business methods, creating, 48
 - business methods, testing, 63
 - create methods, creating, 43
 - create methods, testing, 60
 - creating, 36 to 42
 - finder methods, creating, 46
 - finder methods, testing, 62
 - local or remote interfaces, 36
 - primary key class, 42
 - testing, 60 to 63
 - validating, 49
- example applications
 - StockApp and UDDI registry, 28
 - where to download, xviii
- executing
 - test applications for entity beans, 59, 66
 - test applications for session beans, 86

F

`ffjuser40ee`, UNIX default user settings file, 6
finder methods
 `Customerreview.findByRestaurantName`,
 46, 62
 `Restaurant.findAll`, 34, 46, 62
 testing, 62

G

Generate Client Proxy menu item, 109
Generate/Compile Java File menu item, 97
generated runtime classes, 92
generated web service, 92

I

interfaces, local or remote
 for entity beans, 36
 for session beans, 69

J

J2EE applications
 creating, 98
 deploying, 100
 `DiningGuideApp`, 98
J2EE Reference Implementation (RI)
 stopping, 116
Javadoc technology
 using in the IDE, xviii
JDBC connection pool
 defining in the IDE, 9 to 11
 registering, 11 to 12
JDBC data source
 defining in the IDE, 12
 registering, 12
JDBC persistence manager
 defining in the IDE, 13
 registering, 13
JNDI lookup code, 71

M

Mount Filesystem menu item, 37

N

New CMP Entity EJB menu item, 39
New EJB Test Application menu item, 55
New J2EE Application menu item, 98
New Java Bean menu item, 50
New Web Service menu item, 94

O

Oracle database
 configuring to use with the IDE and the
 application server, 9 to 14
 installing a table, 14
 installing a type 4 JDBC driver, 4
Overview of tasks, 25 to 28

P

parameters
 changing order of, 61
 order in test client, 61

R

reating a web service client, 98
Restaurant entity bean
 `create` method, 43, 61, 67
 creating, 37 to 50
 `findAll` method, 34
 `getRating` method, 48, 63
 `getRestaurantDetail` method, 34
Restaurant table, description, 15
`Restaurant_TestApp`
 bean methods, testing, 60 to 63
 creating, 55 to 59
 deploying, 59
 undeploying, 84 to 85
`RestaurantTable`
 creating, 112

displayed, 19, 114

S

session beans

- business methods, creating, 72 to 75
- create method, modifying, 70
- create method, testing, 86
- creating, 69 to 80
- EJB references, adding, 79 to 80
- local or remote interfaces, 69
- testing, 81 to 88
- validating, 79

Sun ONE Application Server 7 server

- configuring for Oracle, 9 to 14
- evaluation installation, 4
- installing the Oracle JDBC driver, 4
- J2SE SDK version requirements, 2
- non-eval installation, 4
- plugin properties to set for entity beans, 57 to 59, 65 to 66
- plugin properties to set for session beans, 82 to 84
- restarting with the IDE, 85
- starting on Microsoft Windows, 5
- starting on Solaris, UNIX, or Linux environments, 5
- supported platforms, 2
- undeploying an application with the IDE, 85

Sun ONE Application Server plugin

- and administrative client libraries, 3
- description, 3
- installation options, 3
- installing with Update Center, 6

Sun ONE Studio 4 IDE

- configuring for Oracle, 9 to 14
- configuring to use the Sun ONE Application Server 7 server, 7 to 9
- installing the Oracle JDBC driver, 4
- setting the default application server, 8
- starting on Microsoft Windows, 5
- starting on Solaris, UNIX, or Linux environments, 5

Sun ONE Studio 4 requirements

- application servers, 2
- J2SE SDK, 2
- web browsers, 2

web servers, 2

Swing client

- adding to the DiningGuide application, 111 to 112
- editing limitations, 112
- examining the code, 116 to 124
- executing, 114

T

test application facility

- adding entity beans to the EJB module, 81
- entity beans, testing, 60 to 64
- session bean, testing, 81 to 88
- test client, creating, 55 to 59, 64 to 66, 81 to 84
- test client, deploying, 59, 66, 86
- test client, using, 60 to 64, 86 to 88
- using, 26
- web service, testing, 97 to 106

test applications

- Customerreview_TestApp, 65
- DiningGuideApp, 98
- DiningGuideManager_TestApp, 81
- Restaurant_TestApp, 55

testing enterprise beans

- business methods, testing, 63
- create method, testing, 61, 67
- finder methods, testing, 62
- results in IDE's output window, 60, 101
- results in J2EE command window, 60, 101
- test client page, 60, 86

U

undeploying an application

- how to, 84 to 85
- reasons for, 84, 100

Update Center, using, 6

user settings directory

- specifying at initial launch, 5
- UNIX default, 6

Using the test application facility, 26

V

Validate EJB menu item, 49, 79

W

web service

- creating, 94 to 97

- creating a client, 108

- description, 92 to 93

- exposing class types underlying collection types, 77 to 79

- generating a client proxy, 108, 109

- generating WSDL, 107

- sharing with other developers, 107 to 109

- testing, 97 to 106

Web Service Descriptive Language (WSDL),
generating, 107

