Forte™ for Java™ 4, Enterprise Edition Tutorial

Forte for Java 4

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

Part No. 816-4057-10
June 2002, Revision A

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

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.

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

Please
Recycle

Adobe PostScript

Contents

Figures

Tables

Before You Begin

Welcome to the Forte™ for Java™, Enterprise Edition tutorial. This tutorial shows you how to use the following Enterprise Edition features:

- 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 J2EE™ Reference Implementation, version 1.3.1 as the application server.
- Web Services features—for building a SOAP RPC-based web service from the existing EJB components and generating JSP™ pages viewable from a web browser
- J2EE Reference Implementation—a customized version of the J2EE RI, version 1.3.1 for deploying and 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 Forte™ for Java™ 4 user interface, 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 Forte for Java 4, Enterprise Edition to create, develop, and deploy a J2EE compliant application, you will benefit from working through this tutorial.

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

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

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

- Java 2 Platform, Enterprise Edition Blueprints
  `http://java.sun.com/j2ee/blueprints`
- *Java 2 Platform, Enterprise Edition Specification*
  `http://java.sun.com/j2ee/download.html#platformspec`
- *The J2EE Tutorial* (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 Apache implementation of the SOAP (Simple Object Access Protocol) version 1.1 specification is helpful. Apache SOAP is a subproject of the Apache XML Project. For more information, see:

`http://xml.apache.org/soap/index.html`

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 describes the software requirements for the DiningGuide tutorial, explains how to start the Forte for Java 4 integrated development environment (IDE), how to verify that the IDE is using the correct web server and application server, and how to create the tutorial database tables. This chapter includes a descriptive list of the installed Forte for Java 4 directories.

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 Forte for Java 4 test application facility to test each bean.

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

Chapter 5 explains how the 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 `.login` file. Use `ls -a` to list all files. `% You have mail.`

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`**
		`Password:`

AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Related Documentation

Forte for Java 4 documentation includes books delivered in Acrobat Reader (PDF) format, online help, readme files of example applications, and Javadoc™ documentation.

Documentation Available Online

The documents in this section are available from the Forte for Java 4 portal and the `docs.sun.com`℠ web site.

The documentation link of the Forte for Java Developer Resources portal is at `http://forte.sun.com/ffj/documentation/`. The `docs.sun.com` web site is at `http://docs.sun.com`.

- Release notes (HTML format)

  Available for each Forte for Java 4 edition. Describe last-minute release changes and technical notes.

- *Forte for Java 4 Getting Started Guide* (PDF format) - Community Edition part no. 816-4062-10, Enterprise Edition part no. 816-4063-10

  Available for each Forte for Java 4 edition. Describes how to install the Forte for Java 4 product on each supported platform and includes other pertinent information, such as system requirements, upgrade instructions, web server and application server installation instructions, command-line switches, installed subdirectories, Javadoc setup, database integration, and information on how to use the Update Center.

- The Forte for Java 4 Programming series (PDF format)

  This series provides in-depth information on how to use various Forte for Java 4 features to develop well-formed J2EE applications.

  - *Building Web Components* - part no. 816-4337-10

    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 With Forte for Java* - part no. 816-4061-10

    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. 816-4060-10

    Describes how to build EJB components (session beans, message-driven beans, and entity beans with container-managed or bean-managed persistence) using the Forte for Java 4 EJB Builder wizard and other components of the IDE.

  - *Building Web Services* - part no. 816-4059-10

    Describes how to use the Forte for Java 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. 816-4685-10

    Describes how to use the JDBC productivity enhancement tools of the Forte for Java 4 IDE, including how to use them to create a JDBC application.

- Forte for Java 4 tutorials (PDF format)

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

  - *Forte for Java 4, Community Edition Tutorial* - part no. 816-4058-10

    Provides step-by-step instructions for building a simple J2EE web application using Forte for Java 4, Community Edition tools.

  - *Forte for Java 4, Enterprise Edition Tutorial* - part no. 816-4057-10

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

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.

Online Help

Online help is available inside the Forte for Java 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 several examples that illustrate a particular Forte for Java 4 feature, as well as the source files for the tutorial applications from the Developer Resources portal, at:

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

Javadoc Documentation

Javadoc documentation is available within the IDE for many Forte for Java 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 (816-4057-10) of your document in the subject line of your email.

CHAPTER 1

Getting Started

This chapter explains what you must do before starting the Forte for Java 4, Enterprise Edition tutorial. For your convenience, it duplicates some installation information from the *Getting Started Guide*.

The topics covered in this chapter are:

- "Software Requirements for the Tutorial," which follows
- "Starting the Forte for Java 4 IDE" on page 3
- "Verifying the Correct Default Application Server and Web Server" on page 7
- "Understanding the Forte for Java 4 Directory Structure" on page 6
- "Creating the Tutorial Database Tables" on page 8

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 the SQL script for creating the required database tables. You can obtain these files in a compressed zip file from `http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

Software Requirements for the Tutorial

This section describes how to prepare your system before starting the Forte for Java 4, Enterprise Edition tutorial. This means making sure you have everything required to run the Forte for Java 4 integrated development environment (IDE), as well as having the additional requirements for creating and running the tutorial.

You can access general system requirements from the release notes or from the Forte for Java 4 portal's Documentation page at `http://forte.sun.com/ffj/documentation/index.html`.

What You Need to Run the Forte for Java 4 IDE

The Forte for Java 4 IDE requires the Java™ 2 Platform, Standard Edition (the Java 2 SDK). When you install the IDE, the installer searches your system for the Java 2 SDK software and will notify you and stop the installation if the correct version is not installed on your system. You can download the correct version of the Java 2 SDK from the Java Developer's portal at `http://java.sun.com/j2se/`.

What You Need to Create and Run the Tutorial

You need the following software to create and run the tutorial. Some of these items are included in the default installation of Forte for Java 4, Enterprise Edition. Check the release notes on `http://forte.sun.com/ffj/documentation/index.html` for supported versions of these items:

- A web browser

  You need a web browser to view the pages of the test application client.

- A web server

  The Forte for Java 4 test client is a web application, which requires a web server. This tutorial uses an embedded version of Tomcat, version 4.0, within the IDE that provides the functionality of a web server for testing purposes.

- Database software—PointBase Server 4.2 Restricted Edition

  The tutorial application accesses a database. This tutorial describes how to use the PointBase Network Server software that can be installed with the Forte for Java 4 IDE. If you did not install Forte for Java 4 yourself, you can verify whether PointBase was installed by looking for a `pointbase` directory under the Forte for Java 4 home directory. If PointBase was not installed, you can run the installer again to install it.

- An application server

  You need an application server to deploy the tutorial's Java 2 Platform, Enterprise Edition (J2EE™) application. This tutorial describes how to use the J2EE Reference Implementation (J2EE RI) server. You *must* use the version of the J2EE RI, version 1.3.1, that is included in Forte for Java 4, Enterprise Edition installation. This version of the J2EE RI server also requires the PointBase Network Server database.

  The IDE is by default configured to use J2EE RI. To verify this, see "Verifying the Correct Default Application Server and Web Server" on page 7.

Starting the Forte for Java 4 IDE

Start the Forte for Java 4 IDE by running the program executable, as described in the following sections, and more fully in the *Forte for Java 4, Enterprise Edition Getting Started Guide*.

Starting the IDE on Solaris, UNIX, and Linux Environments

After installation, a `runide.sh` script is located in the *forte4j-home*`/bin` directory. Launch this script by typing the following in a terminal window:

```
$ sh runide.sh
```

For ways to customize this script, see "Modifying the Session With Command-Line Switches" on page 4.

Starting the IDE on Microsoft Windows

After installation, there are three ways to start the IDE:

- Double-click the Forte for Java 4.0 EE icon on your desktop.

  This runs the `runidew.exe` executable, which launches the IDE without a console window. This executable exists in the *forte4j-home*`\bin` directory, along with an alternative executable—`runide.exe`. The `runide.exe` icon launches the IDE with a console window that includes standard error and standard output from the IDE. On the console, you can press Ctrl-Break to get a list of running threads or Ctrl-C to immediately terminate the program.
- Choose Start → Programs → Forte for Java 4.0 EE → Forte for Java.
- Run any of the executables from the command line.

  For example:

```
C:\> runide.exe [switch]
```

See the next section for information on switches.

Modifying the Session With Command-Line Switches

TABLE 1-1 describes the switches that you can use to modify how you launch the IDE. This information is also available from the *Forte for Java 4, Enterprise Edition Getting Started Guide*, but is provided here for your convenience.

- On Microsoft Windows systems

  You can set options when running the IDE on the command line.

- In Solaris, Linux, and other UNIX environments

  You can modify the `ide.sh` file in the `bin` subdirectory of the installation directory, or you can create your own shell script that calls `ide.sh` with options.

TABLE 1-1 `runide` Command-Line Switches

Switch	**Meaning**
`-classic`	Uses the classic JVM.
`-cp:p` *addl-classpath*	Adds a class path to the beginning of the Forte for Java 4 class path.
`-cp:a` *addl-classpath*	Adds a class path to the end of the Forte for Java 4 class path
`-fontsize` *size*	Sets the font size used in the GUI to the specified size.
`-locale` *language* [:*country*[:*variant*]]	Uses the specified locale for the session instead of the default locale.
`-J`*jvm-flags*	Passes the specified flag directly to the JVM. (There is no space between `-J` and the argument.)
`-jdkhome` *jdk-home-dir*	Uses the specified Java 2 SDK instead of the default SDK.
`-h` or `-help`	Opens a GUI dialog box that lists the command-line options.
`-hotspot` or `-client` or `-server` or `-classic` or `-native` or `-green`	Uses the specified variant of JVM.

TABLE 1-1 `runide` Command-Line Switches *(Continued)*

Switch	Meaning
`-single`	Runs the IDE in single-user mode. Enables you to launch the IDE from *forte4j-home* instead of from your user settings directory.
`-ui` *UI-class-name*	Runs the IDE with the given class as the IDE's look and feel.
`-userdir` *user-directory*	Uses the specified directory for your user settings for the current session. See the next section for more information.

Specifying Your User Settings Directory

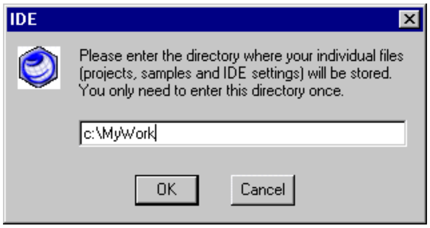
The Forte for Java 4 IDE stores your individual projects, samples, and IDE settings in your own special directory. This enables individual developers to synchronize their development activities, while keeping their own personal work and preferences separate.

- In Solaris, UNIX, or Linux environments

  If you don't explicitly specify a user settings directory with the `-userdir` command-line switch, user settings are located by default in *user-home*`/ffjuser40ee`.

- On Microsoft Windows systems

  At first launch of the Forte for Java 4 IDE, you are prompted to specify a user settings directory. Use a complete specification, for example, `C:\MyWork`.

  IDE

  Please enter the directory where your individual files (projects, samples and IDE settings) will be stored. You only need to enter this directory once.

  c:\MyWork

  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.

Understanding the Forte for Java 4 Directory Structure

When you install the Forte for Java 4 software, the subdirectories described in TABLE 1-2 are included in your installation directory.

TABLE 1-2 Forte for Java 4 Directory Structure

Directory	Purpose
`beans`	Contains JavaBeans™ components installed in the IDE.
`bin`	Includes Forte for Java 4 launchers (as well as the `ide.cfg` file on Microsoft Windows installations).
`docs`	Contains the Forte for Java 4 help files and other miscellaneous documentation. (Release notes are found under *forte4j-home*.)
`examples`	Contains source files and readme files for enterprise edition examples.
`iPlanet`	Contains files used by the iPlanet plug-ins.
`j2sdkee1.3.1`	Contains the J2EE Reference Implementation, if installed.
`javadoc`	The directory mounted by default in the IDE's Javadoc repository. Both Javadoc provided with the IDE and user-created Javadoc are stored here.
`jwsdp`	Contains Java Web Services Developer's Pack software (UDDI internal registry).
`lib`	Contains JAR files that make up the IDE's core implementation and the open APIs.
`modules`	Contains JAR files of Forte for Java 4 modules.
`pointbase`	Contains the executables, classes, databases, and documentation for the PointBase Server 4.2 Restricted Edition database (if installed).
`sampledir`	Contains source files and readme files for several examples.
`sources`	Contains sources for libraries that might be redistributed with user applications.
`system`	Includes files and directories used by the IDE for special purposes. Among these are `ide.log`, which provides information useful when seeking technical support.
`tomcat401`	Contains sources for the Tomcat, 4.01, web server.

When you launch the Forte for Java 4 software, the subdirectories in TABLE 1-3 are installed in your user settings directory. Most of them correspond to subdirectories in the Forte for Java 4 home directory, and are used to hold your settings.

TABLE 1-3 Directory Structure for the User Settings Directory

Directory	Purpose
`beans`	Contains user settings for JavaBeans components installed in the IDE.
`javadoc`	Contains user settings for Javadoc files installed in the IDE.
`lib`	Contains user settings for the system `lib` files.
`modules`	Contains modules downloaded from the Update Center.
`sampledir`	The directory mounted by default in the Filesystems pane of the Explorer. Objects you create in the IDE are saved here unless you mount other directories and use them instead.
`sampledir/examples`	Contains several NetBeans example applications.
`SunONE`	Contains user settings for the iAS application server.
`system`	Contains user settings for system files and directories.
`tomcat401_base`	Contains user settings for your work with JSP pages.

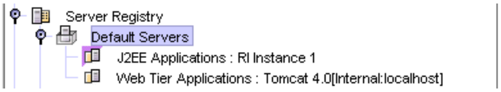
Verifying the Correct Default Application Server and Web Server

The DiningGuide tutorial uses the J2EE RI application server. The test application generated for testing DiningGuide's web services uses the Tomcat web server. Both of these servers are set as the default servers by the installer. However, if you use other servers, you should make these servers the default servers before you test or run the DiningGuide application.

To verify that the J2EE RI application server and Tomcat web server are the default servers:

1. **In the Forte for Java 4 IDE, click the Explorer's Runtime tab.**

2. **Expand the `Server Registry` node and its `Default Servers` subnode.**
   - If the `Default Servers` node looks like this, then the IDE is using the correct servers.



Server Registry
Default Servers
J2EE Applications : RI Instance 1
Web Tier Applications : Tomcat 4.0[Internal:localhost]

   - If anything other than RI Instance 1 and Tomcat 4.0 are listed, then:

     i. **Right-click the default server that is wrong and choose Set Default Server.**

     The Select Default Web (or Application) Server dialog box is displayed.

     ii. **Select the correct server and click OK.**

Creating the Tutorial Database Tables

Before you can start the Forte for Java, Enterprise Edition tutorial, you must create and install two database tables in the PointBase Network Server database. Use the SQL script in Appendix B to create these tables. A script file, `rest_pb.sql`, is also available within the `diningguide.zip` file for the DiningGuide tutorial, available from:

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

The script in Appendix B creates the database schemas shown in TABLE 1-4.

TABLE 1-4 DiningGuide Database Tables

Table Name	Columns	Primary Key	Other
Restaurant	restaurantName	yes	
	cuisine		
	neighborhood		
	address		
	phone		
	description		
	rating		

TABLE 1-4 DiningGuide Database Tables *(Continued)*

Table Name	Columns	Primary Key	Other
CustomerReview	restaurantName	yes	Compound primary key with CustomerName; references Restaurant(restaurantName)
	customerName	yes	
	review		

The Restaurant table contains the records shown in TABLE 1-5.

TABLE 1-5 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 1-6.

TABLE 1-6 CustomerReview Table Records

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

Install the tutorial tables in the default PointBase database according to the following instructions.

Note – If you have already started the Forte for Java IDE, you can either leave it running while you create your database tables, or quit and restart after you finish.

1. **Start the PointBase Server.**

   - In Solaris or Linux environments: Run the `Server` file in the *forte4j-home*`/pointbase/server` directory.
   - On Microsoft Windows: Choose Start → Forte for Java 4.0 EE → PointBase → Network Server → Server or double-click the `server.bat` file in the *forte4j-home*`/pointbase/server` directory.

2. **Start the PointBase Console.**
   - In Solaris or Linux environments: Run the `Console` file in the *forte4j-home*`/pointbase/client` directory.
   - On Microsoft Windows: Choose Start → Forte for Java 4.0 EE → PointBase → Client Tools → Console or double-click the `console.bat` file in the *forte4j-home*`/pointbase/client` directory.

     The Connect To Database dialog box appears, showing values for the PointBase driver to the default `sample` database.

3. **Click OK.**

   The PointBase Console is displayed.

4. **Copy the PointBase script from Appendix B and paste it into the SQL entry window of the Console.**

   Alternatively, if you have the SQL script file from the `diningguide.zip` file, you can choose File → Open and open the `rest_pb.sql` script.

5. **Choose SQL → Execute All.**

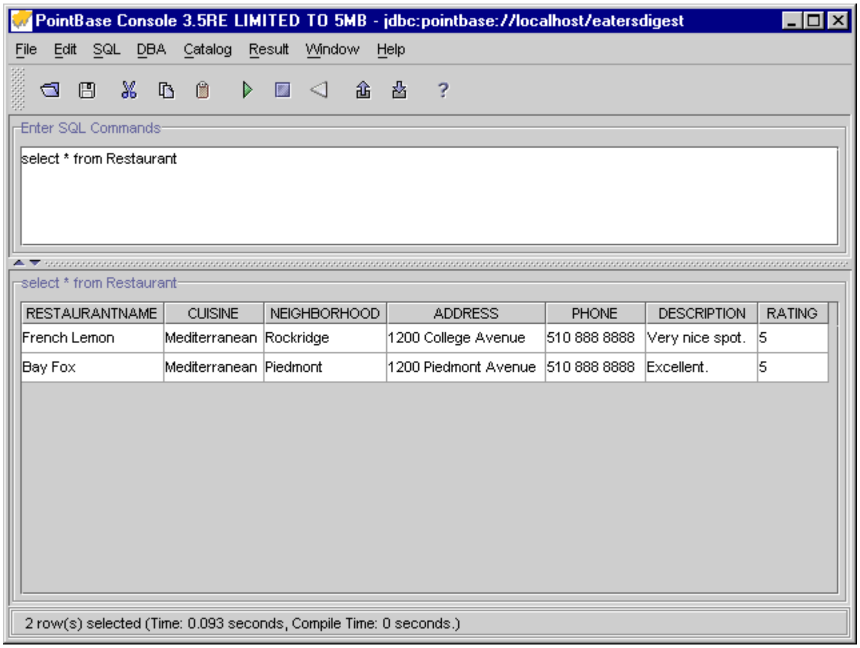
   The message window confirms that the script was executed. (Ignore the initial messages beginning "Cannot find the table…" These appear because there are DROP statements for tables that have not been created yet. These DROP statements will be useful in the future if you want to rerun the script to initialize the tables.)

6. **Test that you have created the table by clearing the SQL entry window (Window → Clear Input) and typing:**

   ```
   select * from Restaurant;
   ```

7. **Choose SQL → Execute.**

   Your console should display the Restaurant table.



PointBase Console 3.5RE LIMITED TO 5MB - jdbc:pointbase://localhost/eatersdigest
File
Edit
SQL
DBA
Catalog
Result
Window
Help
Enter SQL Commands

```
select * from Restaurant
```

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

```
2 row(s) selected (Time: 0.093 seconds, Compile Time: 0 seconds.)
```

Note – If your display does not look like this table, choose Window → Windows to change the display type.

---

8. **Close the PointBase Console window.**

   Now, you're ready to start the tutorial.

CHAPTER 2

Introduction to the Tutorial

In the process of creating the tutorial example application, you will learn how to build a simple J2EE application using Forte for Java 4, Enterprise Edition features.

This chapter describes the application you will build, first describing its requirements, and then presenting an architecture that fulfills the requirements. The final section describes how you use Forte for Java 4, Enterprise Edition features—the EJB Builder, the test application facility, and the New Web Service wizard—to create the application.

This chapter is organized into the following sections:

- "Functionality of the Tutorial Application," which follows
- "User's View of the Tutorial Application" on page 15
- "Architecture of the Tutorial Application" on page 18
- "Overview of Tasks for Creating the Tutorial Application" on page 21

Functionality of the Tutorial Application

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

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

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

Application Scenarios

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

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

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

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

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

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

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

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

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

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

Application Functional Specification

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

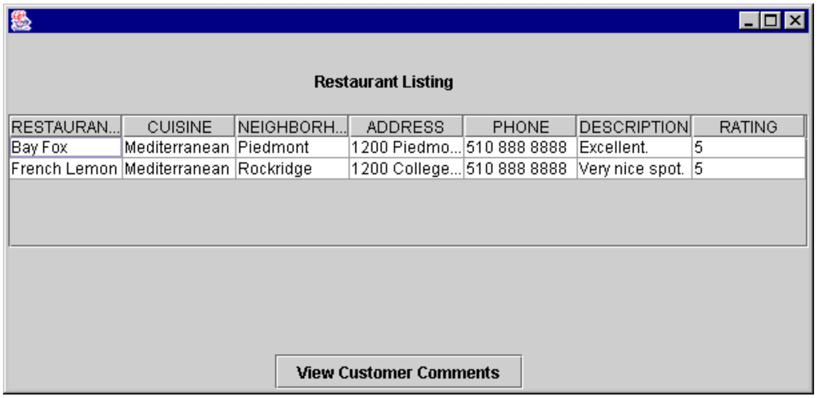
- A master view of all restaurant data through a displayed list
- A button on the master restaurant list window for retrieving all customer review data for a given restaurant
- A master view of all customer review data for a given restaurant
- A button on the customer review list window for adding a new review
- Text entry fields on the customer review list window for typing in a new customer name and new customer review for the current restaurant
- A button on the customer review list window for submitting the finished review data to the database

User’s View of the Tutorial Application

The user’s view of the application illustrates how the scenarios and the functional specification, described in “Functionality of the Tutorial Application” on page 13 are realized.

1. **In the Forte for Java 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:



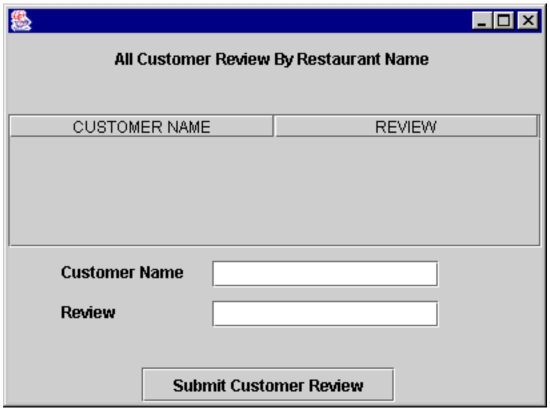
Restaurant Listing

RESTAURAN...	CUISINE	NEIGHBORH...	ADDRESS	PHONE	DESCRIPTION	RATING
Bay Fox	Mediterranean	Piedmont	1200 Piedmo...	510 888 8888	Excellent.	5
French Lemon	Mediterranean	Rockridge	1200 College...	510 888 8888	Very nice spot.	5

This window displays the data from the Restaurant table created in “Creating the Tutorial Database Tables” on page 8.

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

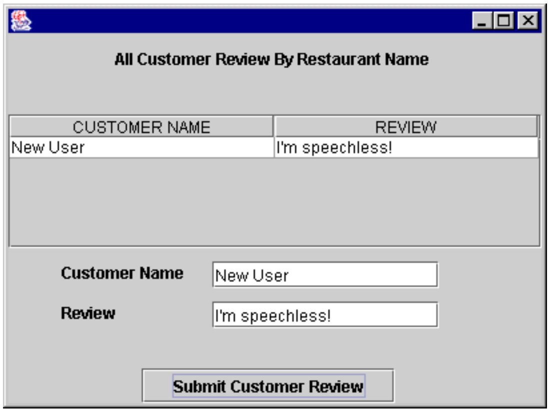
   For example, select the Bay Fox restaurant. The `CustomerReviewTable` window is displayed.



   In this case, no records are shown, because none are in the database. Refer to TABLE 1-6.

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

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



CUSTOMER NAME	REVIEW
New User	I'm speechless!

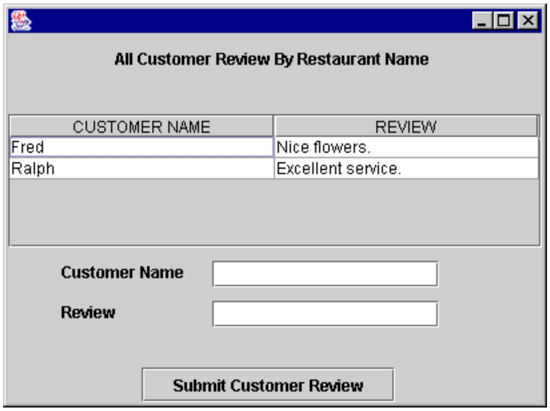
Customer Name: New User

Review: I'm speechless!

   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.

   Two customer review records are displayed. Refer to TABLE 1-6 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 PointBase database console.**

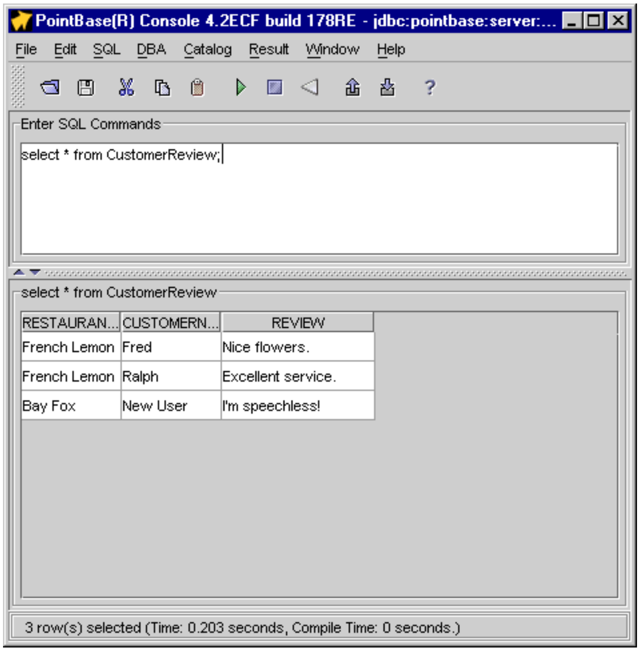
   The PointBase server must first be running. Refer to the procedures in "Creating the Tutorial Database Tables" on page 8 for information.

8. **In the PointBase Console, type the following statement:**

```
select * from CustomerReview;
```

9. **Choose SQL → Execute.**

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


PointBase(R) Console 4.2ECF build 178RE - jdbc:pointbase:server:...
File Edit SQL DBA Catalog Result Window Help
Enter SQL Commands

```
select * from CustomerReview;
```

select * from CustomerReview

RESTAURAN...	CUSTOMERN...	REVIEW
French Lemon	Fred	Nice flowers.
French Lemon	Ralph	Excellent service.
Bay Fox	New User	I'm speechless!

3 row(s) selected (Time: 0.203 seconds, Compile Time: 0 seconds.)

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.

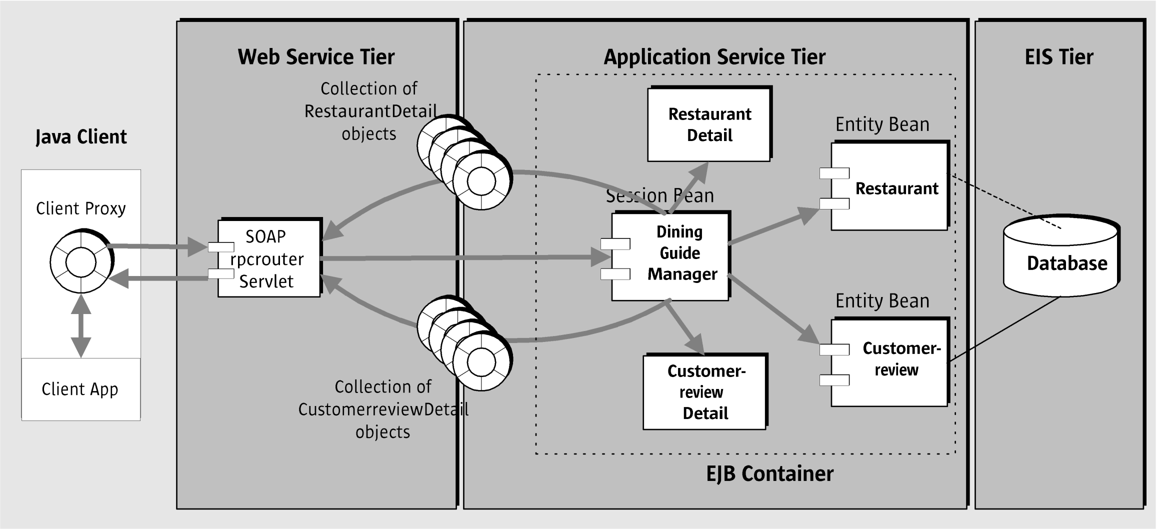


FIGURE 2-1 DiningGuide Application Architecture

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 Forte for Java 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 Forte for Java 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 Forte for Java 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 Forte for Java 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 Apache SOAP 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 Forte for Java 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 Forte for Java 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 Forte for Java 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 Forte for Java 4, Enterprise Edition, while still brief enough for you to create in a short time (perhaps a day). This places certain restrictions on its scope, for example:

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

Although the tutorial application described in this book is designed to be a simple application that you can complete quickly, you might want to import the entire application, view the source files, or copy and paste method code into methods you create. The DiningGuide application is accessible from the Examples and Tutorials page of the Forte for Java 4 Developer's portal at:

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

CHAPTER 3

Building the EJB Tier of the DiningGuide Application

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

- "Overview of the Tutorial's EJB Tier," which follows
- "Creating Entity Beans With the EJB Builder" on page 32
- "Creating Detail Classes to View Entity Bean Data" on page 44
- "Testing the Entity Beans" on page 48
- "Creating a Session Bean With the EJB Builder" on page 59
- "Testing the Session Bean" on page 71
- "Comments on Creating a Client" on page 77

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 Forte for Java 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 Forte for Java 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 Forte for Java 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 J2EE RI 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.

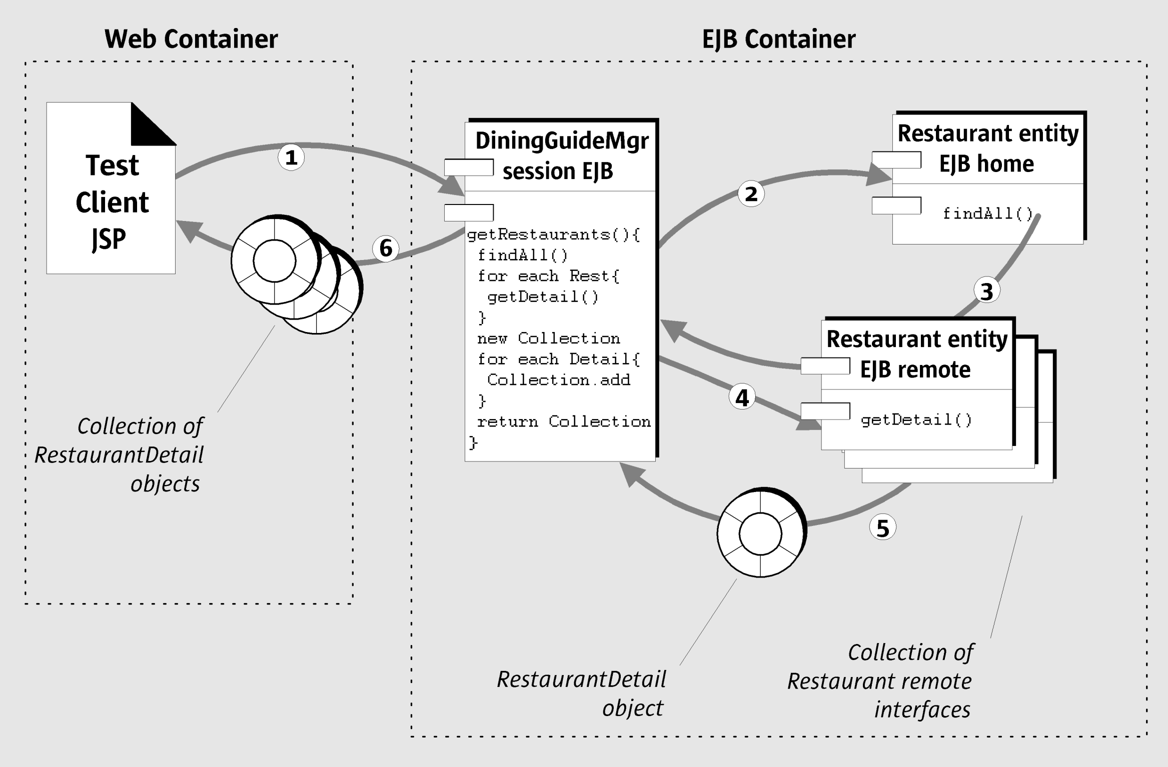
In the DiningGuide application, client requests might include obtaining data on all the restaurants in the database or finding all the customer reviews for a given restaurant. Submitting a review for a given restaurant is another client request. These requests are not interrelated, and don't require 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.


Web Container
EJB Container
Test Client JSP
1
DiningGuideMgr session EJB
2
Restaurant entity EJB home
findAll()
3
Restaurant entity EJB remote
getDetail()
4
5
6
Collection of RestaurantDetail objects
RestaurantDetail object
Collection of Restaurant remote interfaces

```
getRestaurants(){
 findAll()
 for each Rest{
  getDetail()
 }
 new Collection
 for each Detail{
  Collection.add
 }
 return Collection
}
```

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 tasks:

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 Forte for Java 4 help topics on EJB components.

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

Creating the `Restaurant` Entity Bean

First, create a directory to mount as a filesystem to contain the application. Next, create a package for the EJB tier. Finally, create the entity beans within the package.

Note – The following instructions assume that the Forte for Java 4 IDE and the PointBase server (see Step 1 under "Creating the Tutorial Database Tables" on page 8) are both running.

To create the `Restaurant` entity bean:

1. **Somewhere on your file system, create a directory and name it `DiningGuide`.**
2. **In the Forte for Java 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 `DiningGuide` directory, select it, and click Finish.**

   The new directory (for example, `c:\DiningGuide`) 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.

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

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

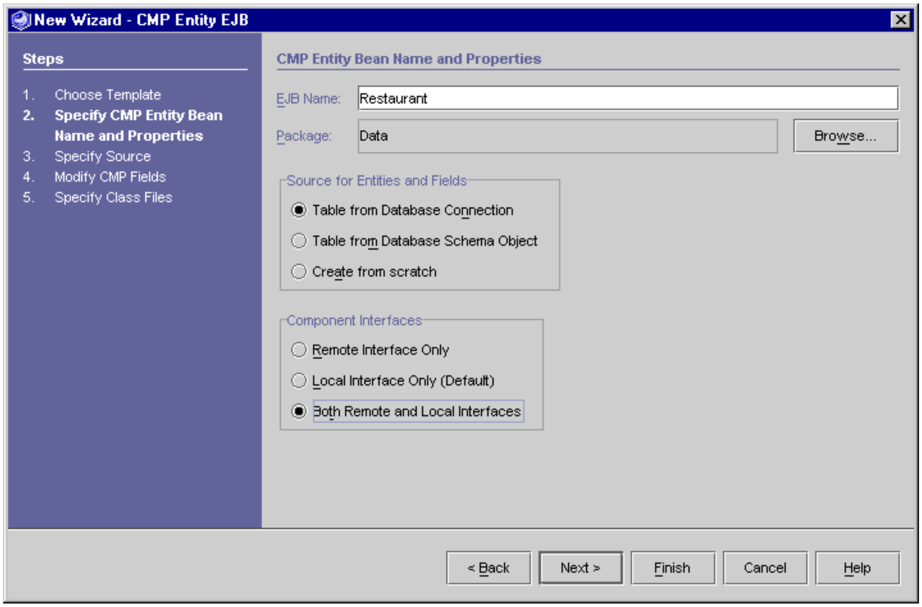
7. **Right-click the new `Data` package and choose New → J2EE → CMP Entity EJB.**

   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.

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

   - Source for Entities and Fields: **`Table from Database Connection`**
   - Component Interfaces: **`Both Remote and Local Interfaces`**

   The New wizard should look like this.



New Wizard - CMP Entity EJB

Steps

1. Choose Template
2. **Specify CMP Entity Bean Name and Properties**
3. Specify Source
4. Modify CMP Fields
5. Specify Class Files

CMP Entity Bean Name and Properties

EJB Name: Restaurant

Package: Data

Source for Entities and Fields

● Table from Database Connection

○ Table from Database Schema Object

○ Create from scratch

Component Interfaces

○ Remote Interface Only

○ Local Interface Only (Default)

● Both Remote and Local Interfaces

9. **Click Next.**

   This displays the Table from Database pane. The connection to the database you created in “Creating the Tutorial Database Tables” on page 8 is displayed as a broken square (▣) that is labeled `jdbc:pointbase:server://localhost:9092/sample [pbpublic on PBPUBLIC]`.

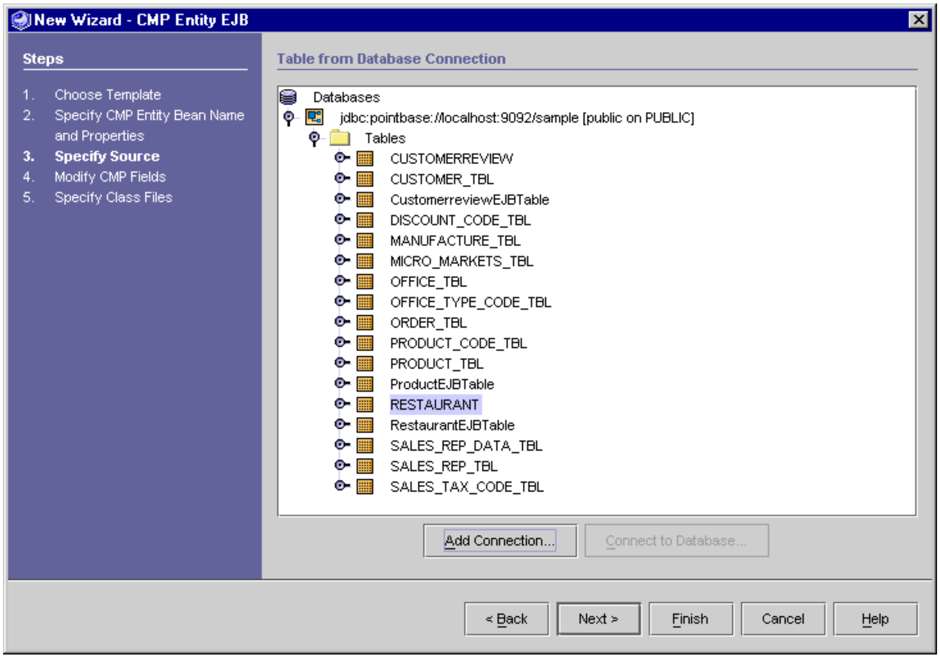
10. **Select the broken square and click the Connect to Database button.**

    The square is displayed as whole (▣), indicating the connection exists.

Note – If you did not get this result, you probably forgot to start the PointBase Network Server. Choose Tools → PointBase Network Server → Start Server. Then click the Add Connection button to add the connection.

---

11. **Expand the database connection node and the `Tables` node under it, and select the `RESTAURANT` table.**



New Wizard - CMP Entity EJB

Steps

1. Choose Template
2. Specify CMP Entity Bean Name and Properties
3. **Specify Source**
4. Modify CMP Fields
5. Specify Class Files

Table from Database Connection

- Databases
  - jdbc:pointbase://localhost:9092/sample [public on PUBLIC]
    - Tables
      - CUSTOMERREVIEW
      - CUSTOMER_TBL
      - CustomerreviewEJBTable
      - DISCOUNT_CODE_TBL
      - MANUFACTURE_TBL
      - MICRO_MARKETS_TBL
      - OFFICE_TBL
      - OFFICE_TYPE_CODE_TBL
      - ORDER_TBL
      - PRODUCT_CODE_TBL
      - PRODUCT_TBL
      - ProductEJBTable
      - RESTAURANT
      - RestaurantEJBTable
      - SALES_REP_DATA_TBL
      - SALES_REP_TBL
      - SALES_TAX_CODE_TBL

12. **Click Next.**

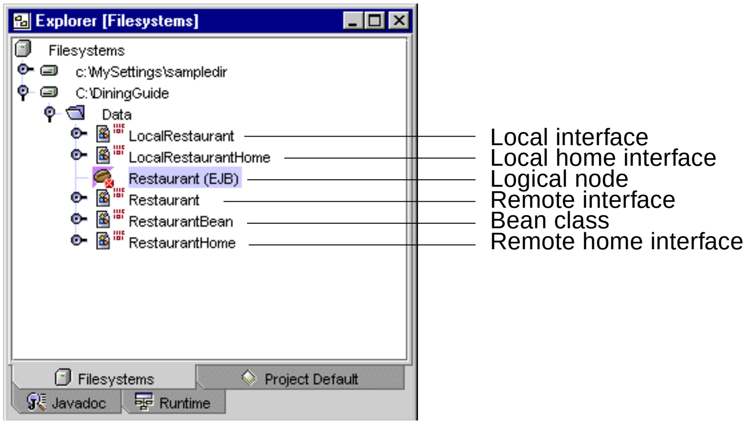
    The CMP Fields pane is displayed. You see a side-by-side display of the columns of the Restaurant database table and the corresponding Java fields that the columns will be mapped to when the wizard creates the `Restaurant` entity bean.

13. **Accept all the default labels and click Next.**

    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.

14. **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.

15. **Choose File → Save to save your work.**

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 Connection`**
    - Component Interfaces: **`Both Remote and Local Interfaces`**

3. **Click Next.**

    The Table from Database pane is displayed.

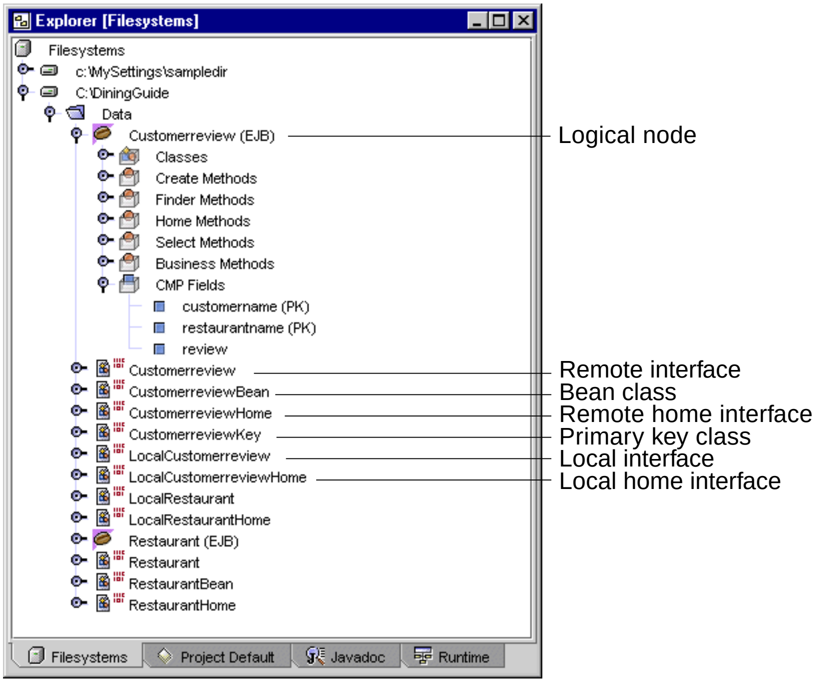
    If the `sample` database is still connected, you can tell this by the square (▣) icon, Proceed to Step 4.

    If the connection is a broken square (⊠), select the broken square and click the Connect to Database button.

4. **Open the database node and the `Tables` folder, 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 an additional component, the `CustomerreviewKey` bean. This bean is automatically created when the entity bean has a composite primary key. (See TABLE 1-4 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 (java.lang.Integer)
   ```

   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.

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;
}
```

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

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

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 RI 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	`Object(o)`
From	`Restaurant o`

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

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

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

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	`Object(o)`
From	`Customerreview o`
Where	`o.restaurantname = ?1`

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

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 `java.lang.Integer` in the Return Type field.**

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

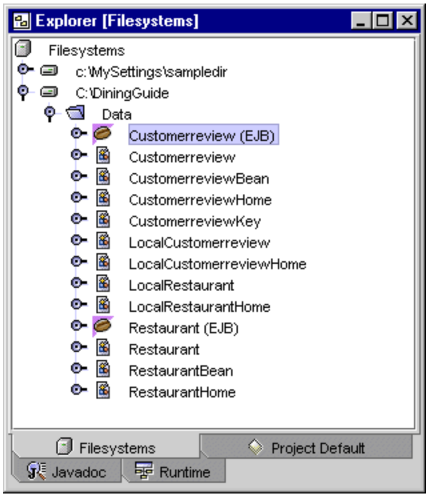
4. **Click OK.**

   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 29, 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 (Integer)
   ```

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
java.lang.Integer rating
```

(You must type in `java.lang.Integer`; you can not select it from the list of types.)

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.**

   `Data.RestaurantDetail` is displayed in the Return Type field.

4. **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. **In the Explorer, right-click the `Customerreview (EJB)` logical node and choose Add Business Method.**

7. **Type `getCustomerreviewDetail` in the Name field.**

8. **For the return type, use the Browse button to select the `CustomerreviewDetail` class.**

9. **Click OK to create the method.**

10. **Open the method in the Source Editor and add the following bold code:**

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

11. **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 Forte for Java 4 IDE includes a mechanism for testing enterprise beans without having to create your own client. This feature uses the J2EE RI as the application server. The enterprise bean is deployed as part of an application that uses JavaServer Pages technology. The test client is displayed in a web browser. Using this page, you can create instances of the bean and exercise the bean's create, finder, and business methods.

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

Creating a Test Client for an Entity Bean

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

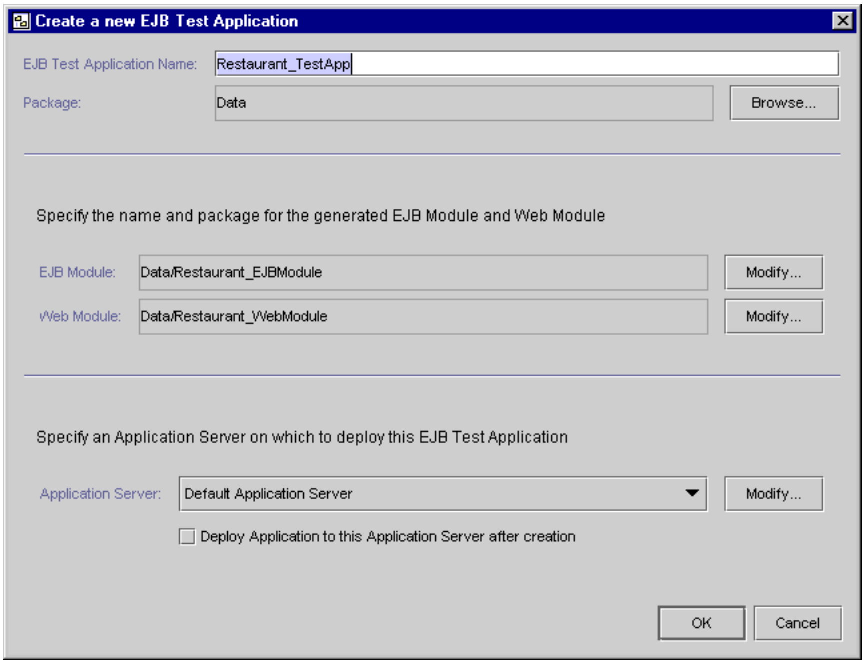
To create a test client for the `Restaurant` entity bean:

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:



Create a new EJB Test Application

EJB Test Application Name: Restaurant_TestApp

Package: Data

Specify the name and package for the generated EJB Module and Web Module

EJB Module: Data/Restaurant_EJBModule

Web Module: Data/Restaurant_WebModule

Specify an Application Server on which to deploy this EJB Test Application

Application Server: Default Application Server

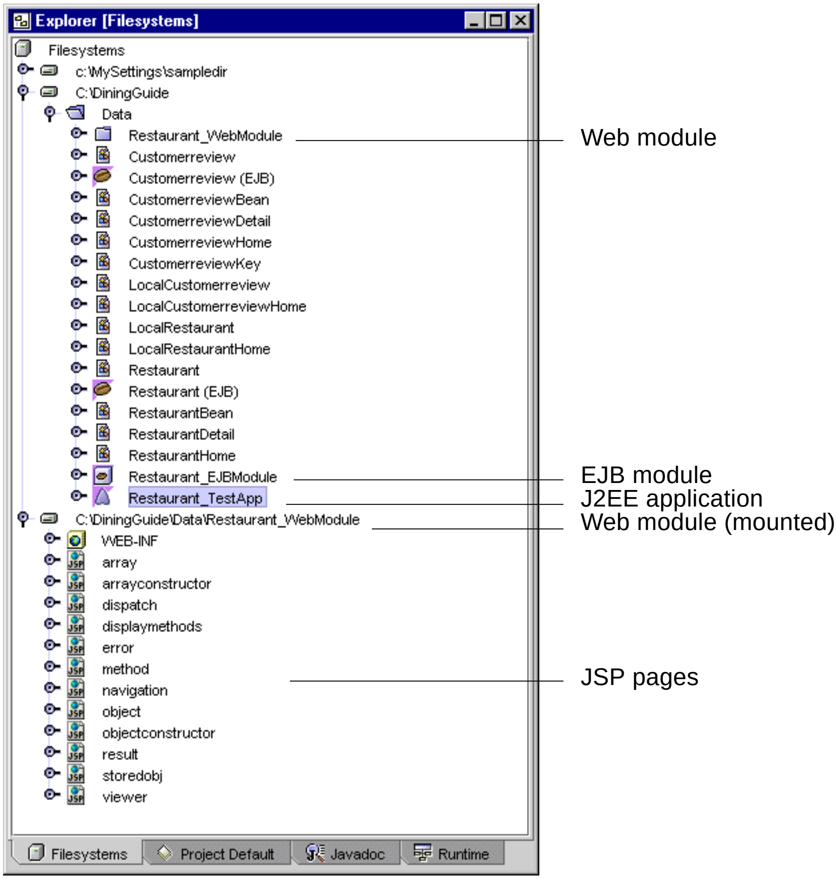
□ Deploy Application to this Application Server after creation

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.

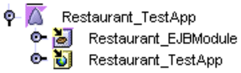
4. **View the resulting test objects in the Explorer.**

   The IDE has created an EJB module named `Restaurant_EJBModule`, a web module named `Restaurant_WebModule` (which is 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.



```
Explorer [Filesystems]
Filesystems
  c:\MySettings\sampledir
  C:\DiningGuide
    Data
      Restaurant_WebModule
      Customerreview
      Customerreview (EJB)
      CustomerreviewBean
      CustomerreviewDetail
      CustomerreviewHome
      CustomerreviewKey
      LocalCustomerreview
      LocalCustomerreviewHome
      LocalRestaurant
      LocalRestaurantHome
      Restaurant
      Restaurant (EJB)
      RestaurantBean
      RestaurantDetail
      RestaurantHome
      Restaurant_EJBModule
      Restaurant_TestApp
  C:\DiningGuide\Data\Restaurant_WebModule
    WEB-INF
    array
    arrayconstructor
    dispatch
    displaymethods
    error
    method
    navigation
    object
    objectconstructor
    result
    storedobj
    viewer
```

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`:



```
Restaurant_TestApp
  Restaurant_EJBModule
  Restaurant_TestApp
```

Providing the RI Plugin With PointBase Information

In order for the test client to find the database and log onto it, you must add PointBase information to the Reference Implementation properties of the EJB module.

To add the required information:

1. **Select the EJB module (`Restaurant_EJBModule`) in the Explorer and display its property sheet.**

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

2. **Select the J2EE RI tab of the Properties window.**

Note – If there is no J2EE RI tab on the Properties window, there is no instance of the Reference Implementation in the Server Registry. See "Verifying the Correct Default Application Server and Web Server" on page 7 for correcting this problem.

---

3. **Type `jdbc/Pointbase` in the Data Source JNDI Name field.**

Note – Make sure to spell "Pointbase" with an initial capital only. This is the way it is specified in the J2EE RI default properties file, and must be exactly the same for this property.

---

4. **Type `PBPUBLIC` in the Data Source Password field.**

   This will be displayed as asterisks.

5. **Type `PBPUBLIC` in the Data Source UserName field.**

6. **Select the value field for the SQL Deployment Settings to display the ellipsis (…) button.**

7. **Click the ellipsis button to display the SQL Deployment Settings property editor.**

   The `Restaurant` bean is displayed in the left column.

8. **Select the Restaurant bean and deselect the two options that appear in the right column.**

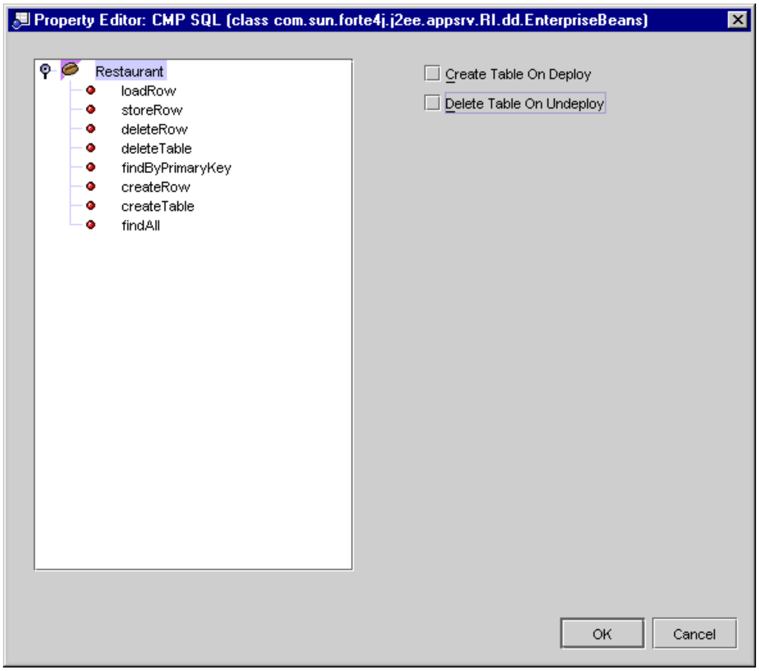
   The two options are:

   - Create Table On Deploy
   - Delete Table On Undeploy

   In "Creating the Tutorial Database Tables" on page 8, you used the database script to create the Restaurant and Customerreview tables. You do not need to recreate them every time you deploy the application.

9. **Expand the Restaurant node.**

   The SQL Deployment Settings Property Editor looks like this:



□ Create Table On Deploy

□ Delete Table On Undeploy

10. **To view the SQL generated for any method, select the method.**

    The SQL appropriate for your selected database is displayed. If you examine the SQL generated for the findAll method, you can see its relation to the EJB QL you specified for this method.

11. **Click OK to accept the changes and close the editor.**

12. **Change the value of the Use Delimited Identifiers in SQL property to False.**

    This action prevents the RI plugin from generating SQL that creates table names and column names surrounded by quotation marks. (You only need the quotation marks if your table and column names use restricted words.)

Deploying the Test Application

Note – Make sure the PointBase Server is running (see Step 1 under "Creating the Tutorial Database Tables" on page 8) before you deploy a J2EE application that accesses the database. In addition, make sure the J2EE RI server is *not* running outside the IDE. The deployment process automatically starts the J2EE RI server (or restarts it if it is already running).

To deploy the `Restaurant` test application:

1. **Right-click the `Restaurant_TestApp` J2EE application node and choose Deploy from the contextual menu.**

   A Progress Monitor window shows the J2EE RI server starting up, followed by the deployment process.

2. **Verify that the application is deployed.**

   When "Deployment of `Restaurant_TestApp` is complete" appears in the output window, the application is deployed.

Tip – If your deployment failed, check whether the J2EE RI is the default application server. See "Verifying the Correct Default Application Server and Web Server" on page 7 for more information on how to set it correctly, then redeploy.

Using the Test Client to Test the Entity Bean

Point your web browser to the test page to start the application. 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.

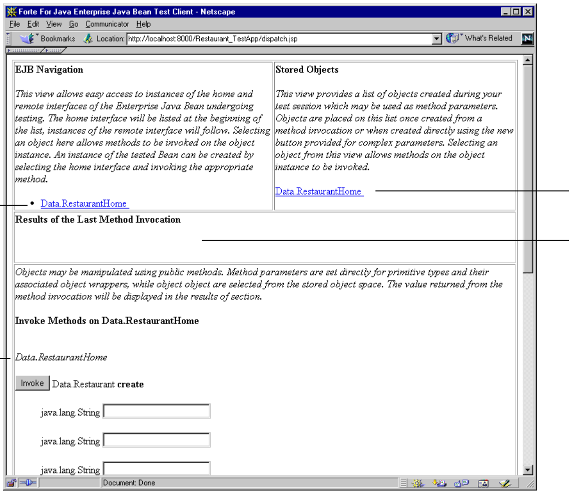
To test the `Restaurant` bean:

1. **From your operating system, launch a web browser and point it to the following URL:**

   ```
   http://localhost:8000/Restaurant_TestApp
   ```

Note – Port 8000 is the default port J2EE RI runs on. If you need to verify that this is the correct port, pointing your browser to `http://localhost:8000` should display the default page of the J2EE RI.

Your browser displays the test client.


Forte For Java Enterprise Java Bean Test Client - Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://localhost:8000/Restaurant_TestApp/dispatch.jsp
What's Related
Document: Done

EJB Navigation

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

- Data.RestaurantHome

Stored Objects

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

Data.RestaurantHome

Results of the Last Method Invocation

Objects may be manipulated using public methods. Method parameters are set directly for primitive types and their associated object wrappers, while object object are selected from the stored object space. The value returned from the method invocation will be displayed in the results of section.

Invoke Methods on Data.RestaurantHome

Data.RestaurantHome

Invoke Data.Restaurant **create**

java.lang.String

java.lang.String

java.lang.String

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

Area where you can enter parameters and invoke methods.

List of objects created during the test session.

Area where results of last method invocation are shown.

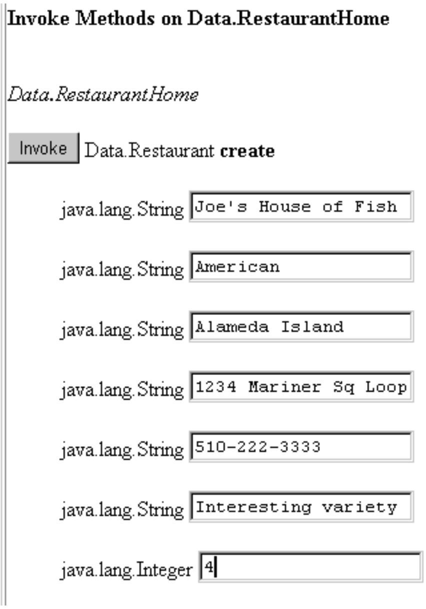
2. **Create an instance of the `Restaurant` bean by invoking the `create` method.**

   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 37).

Note – Double-click the `Restaurant.create` method 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

Data.Restaurant **create**

java.lang.String Joe's House of Fish

java.lang.String American

java.lang.String Alameda Island

java.lang.String 1234 Mariner Sq Loop

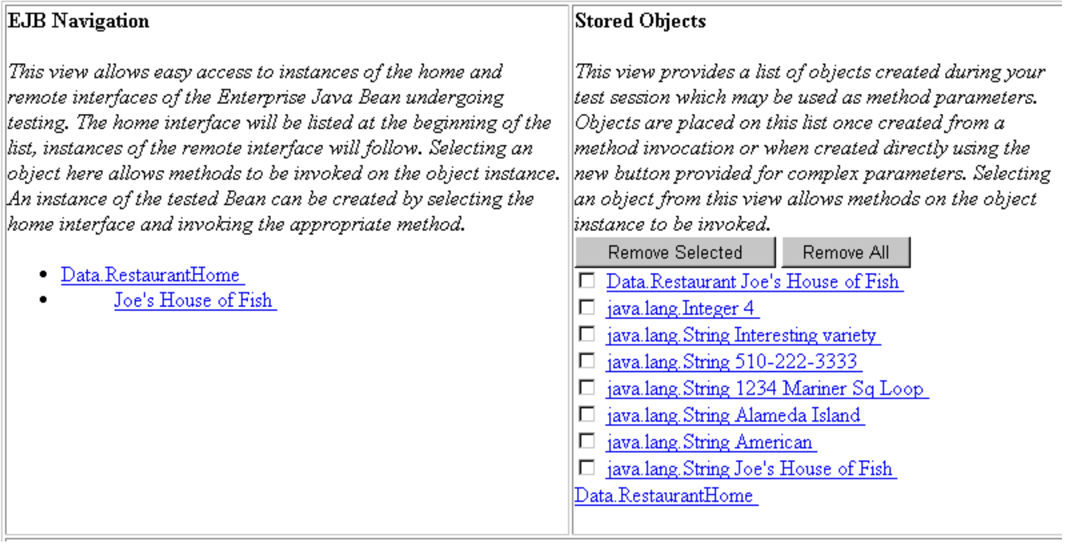
java.lang.String 510-222-3333

java.lang.String Interesting variety

java.lang.Integer 4

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.



EJB Navigation

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

- Data.RestaurantHome
- Joe's House of Fish

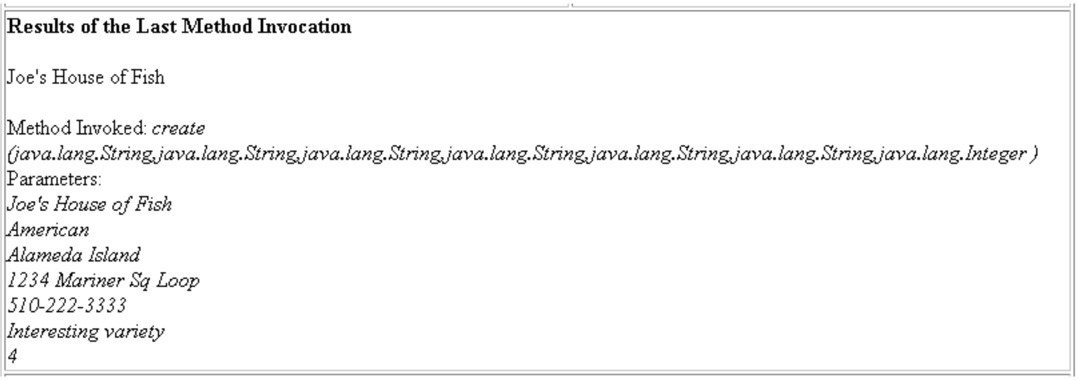
Stored Objects

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

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

Data.RestaurantHome

The results are shown in the Results area.



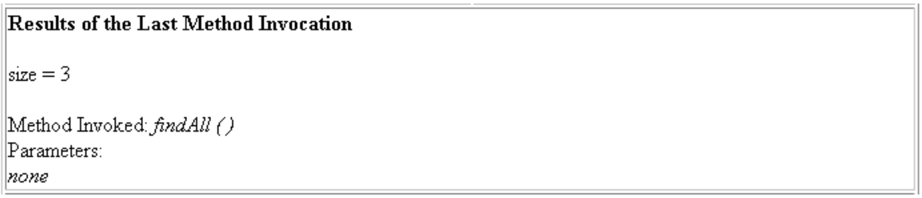
```
Results of the Last Method Invocation

Joe's House of Fish

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

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

   The results area should look like this:



```
Results of the Last Method Invocation

size = 3

Method Invoked: findAll ( )
Parameters:
none
```

   Notice that three items were returned. This demonstrates that the new database record you created in Step 3 was added to the two you created in Chapter 1.

5. **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.

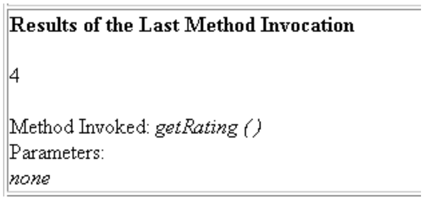
   Now, test the entity bean's business methods.

6. **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.

7. **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.

8. **When you are finished testing, stop the test client by pointing your web browser at another URL or by exiting the browser.**

   You will use the `Restaurant_TestApp` in Chapter 4. You will also need a `Customerreview_TestApp` in Chapter 4.

9. **Create a test client for the `Customerreview` entity bean and test it redoing all the steps appropriately for the `Customerreview` bean, starting with "Creating a Test Client for an Entity Bean" on page 48.**

   The test URL is: `http://localhost:8000/Customerreview_TestApp`

Checking the Additions to the Database

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

1. **Start the PointBase console.**

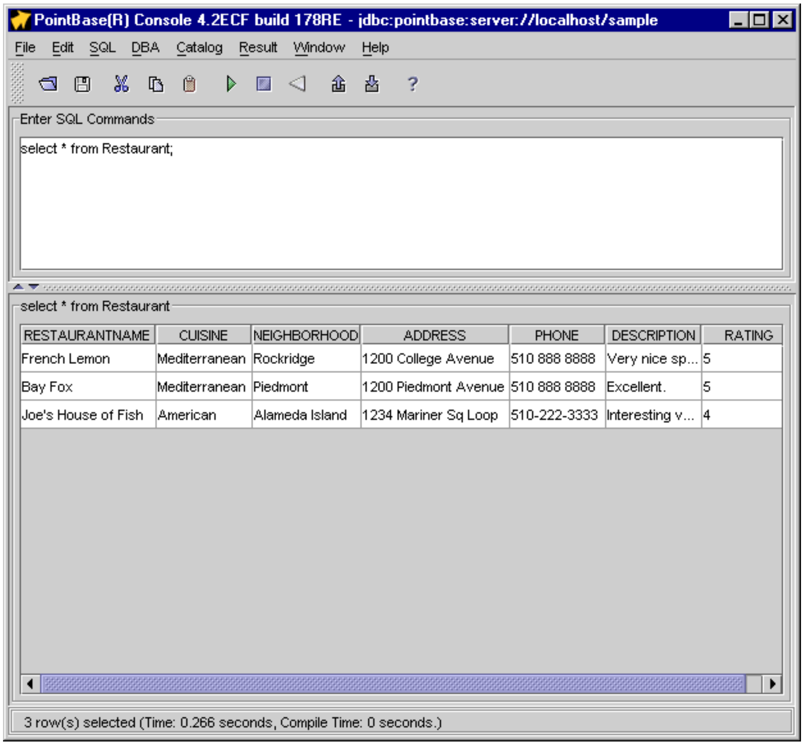
   Refer to Step 2 under "Creating the Tutorial Database Tables" on page 8.

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

   ```
   select * from Restaurant:
   ```

3. **Choose SQL → Execute to execute the statement.**

If you entered the values in Step 2 under "Using the Test Client to Test the Entity Bean" on page 53, the results should look like this:


PointBase(R) Console 4.2ECF build 178RE - jdbc:pointbase:server://localhost/sample
File Edit SQL DBA Catalog Result Window Help
Enter SQL Commands

```
select * from Restaurant;
```

select * from Restaurant

RESTAURANTNAME	CUISINE	NEIGHBORHOOD	ADDRESS	PHONE	DESCRIPTION	RATING
French Lemon	Mediterranean	Rockridge	1200 College Avenue	510 888 8888	Very nice sp...	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 v...	4

3 row(s) selected (Time: 0.266 seconds, Compile Time: 0 seconds.)

You are now ready to create the session bean.

Note – You do not need to stop the J2EE RI process. Whenever you redeploy, the IDE undeploys the application and then restarts the J2EE RI server. When you exit the IDE, a dialog box is displayed for terminating the J2EE RI instance process. However, you can manually terminate it at any time you wish by right-clicking the RI Instance 1 node in the execution window and choosing Terminate Process.

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 Forte for Java 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.**

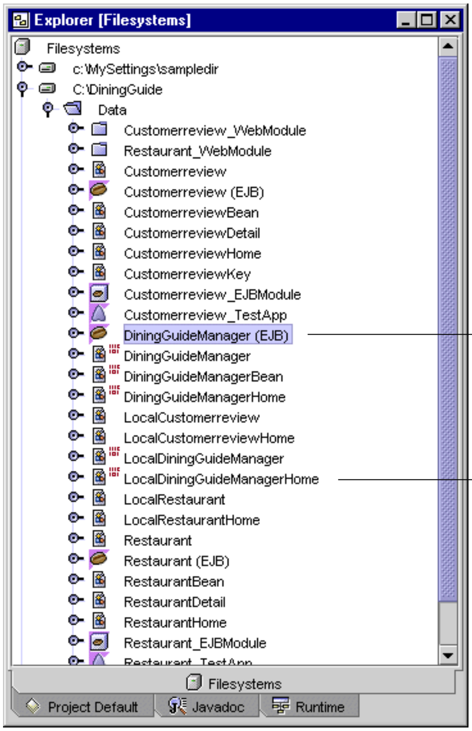
6. **Click Next.**

   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.



`DiningGuideManager` session bean

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("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.

---

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("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. **Choose File → Save All to save your work.**

   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. **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.

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

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

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

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()");
}
```

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

Creating Business Methods That Return Detail Class Types

The web service you will create in Chapter 4 is a 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 types 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 47), but the methods you create now are empty, their purpose being simply to supply the required return type.

For more information on Forte for Java 4 web services and the SOAP runtime, see *Building Web Services* in the Forte for Java 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.**

   `Data.RestaurantDetail` 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.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.**

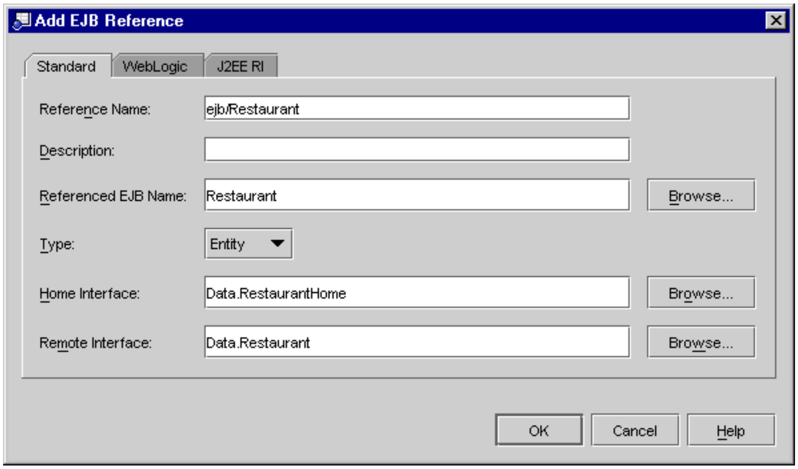
   The Select an EJB browser is displayed.

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

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

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

   The Add EJB Reference property editor looks like this:



Reference Name: ejb/Restaurant

Description:

Referenced EJB Name: Restaurant

Type: Entity

Home Interface: Data.RestaurantHome

Remote Interface: Data.Restaurant

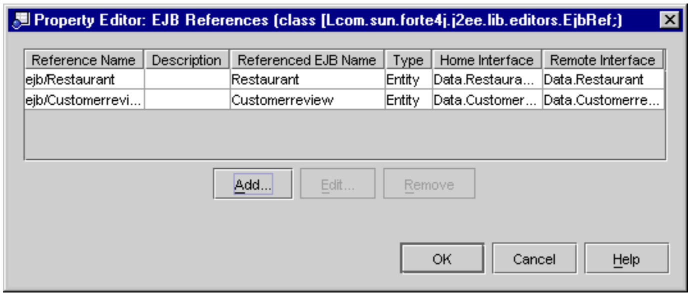
10. **Select the J2EE RI tab.**

11. **Type `jdbc/Pointbase` in the JNDI Name field and click OK.**

    Now add a reference to the `Customerreview` entity bean.

12. **Repeat Step 5 through Step 11 for the `Customerreview` entity bean.**

    The EJB References dialog box looks like this:


Property Editor: EJB References (class [Lcom.sun.forte4j.j2ee.lib.editors.EjbRef;)
Add...
Edit...
Remove
OK
Cancel
Help

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

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.**

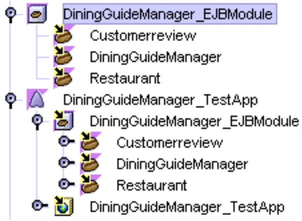
   The Add EJB to EJB Module browser is displayed.

5. **Expand the `DiningGuide` 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:



```
DiningGuideManager_EJBModule
    Customerreview
    DiningGuideManager
    Restaurant
DiningGuideManager_TestApp
    DiningGuideManager_EJBModule
        Customerreview
        DiningGuideManager
        Restaurant
    DiningGuideManager_TestApp
```

8. **Choose File → Save All.**

Providing the RI Plugin With PointBase Information

You must add PointBase information to the Reference Implementation properties of the EJB module. You performed this task with the entity bean test client in "Providing the RI Plugin With PointBase Information" on page 51.

To add the required information:

1. **Select the EJB module (`DiningGuideManager_EJBModule`) in the Explorer and display its property sheet.**

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

2. **Select the J2EE RI tab of the Properties window.**

3. **Type `jdbc/Pointbase` in the Data Source JNDI Name field.**

Note – Make sure to spell "Pointbase" with an initial capital only.

---

4. **Type `PBPUBLIC` in the Data Source Password field.**

   This will be displayed as asterisks.

5. **Type `PBPUBLIC` in the Data Source UserName field.**

6. **Select the value field for the SQL Deployment Settings property to display the ellipsis (…) button.**

7. **Click the ellipsis button to display the SQL Deployment Settings property editor.**

   The `Customerreview` and `Restaurant` beans are displayed in the left column.

8. **Select the each bean and uncheck the two options that appear in the right column:**
   - Create Table On Deploy
   - Delete Table On Undeploy
9. **Click OK to accept the changes and close the editor.**
10. **Change the value of the Use Delimited Identifiers in SQL property to False.**
11. **Save your work with File → Save All.**

Deploying the Test Application

Note – Make sure the PointBase Server is running (see Step 1 under "Creating the Tutorial Database Tables" on page 8) before you deploy a J2EE application that accesses the database. In addition, make sure the J2EE RI server is *not* running outside the IDE. The deployment process automatically starts the J2EE RI server (or restarts it if it is already running).

To deploy the `Restaurant` test application:

1. **Right-click the `DiningGuideManager_TestApp` J2EE application node and choose Deploy from the contextual menu.**

   A Progress Monitor window shows the J2EE RI server starting up, followed by the deployment process.

2. **Verify that the application is deployed.**

   When "Deployment of DiningGuideManager_TestApp is complete" appears in the output window, the application is deployed.

Tip – If your deployment failed, check whether the J2EE RI is the default application server. See "Verifying the Correct Default Application Server and Web Server" on page 7 for more information on how to set it correctly, then redeploy.

Now, test the `DiningGuideManager` session bean.

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.

Point your web browser to the test application to start the application:

1. **From your operating system, launch a web browser and point it to the following URL:**

   http://localhost:8000/DiningGuideManager_TestApp

   Your browser displays the test client, with DiningGuideManagerHome listed as the only instance in the instance list (upper left).

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

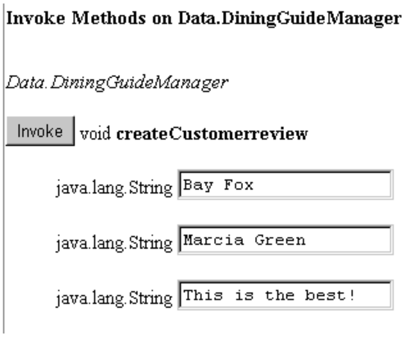
   The Data.DiningGuideManager[*x*] instance appears in the instance list. Now you can test the bean's getter methods.

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

   The getAllRestaurants and getCustomerreviewsByRestaurant methods are made available.

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

   For example:



Invoke Methods on Data.DiningGuideManager [7]

Data.DiningGuideManager

void **createCustomerreview**

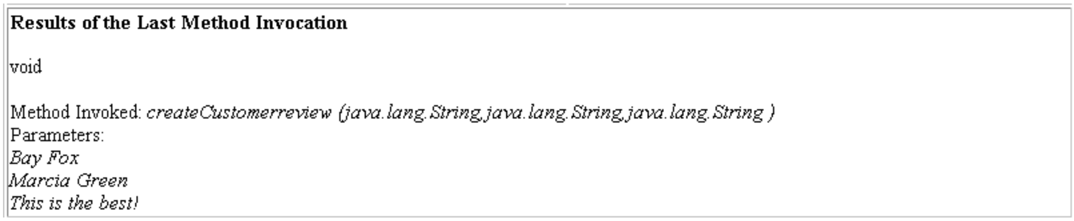
java.lang.String Bay Fox

java.lang.String Marcia Green

java.lang.String This is the best!

5. **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:



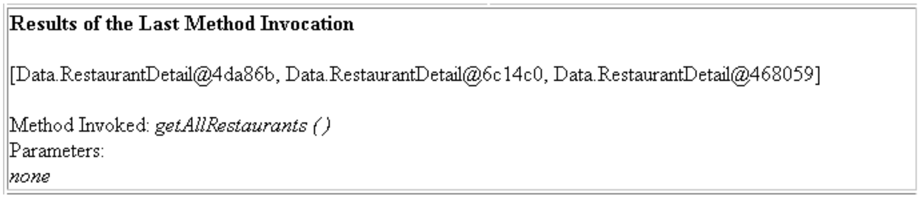
Results of the Last Method Invocation

void

Method Invoked: *createCustomerreview (java.lang.String,java.lang.String,java.lang.String)*
Parameters:
Bay Fox
Marcia Green
This is the best!

6. **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 Entity Bean" on page 53), 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.



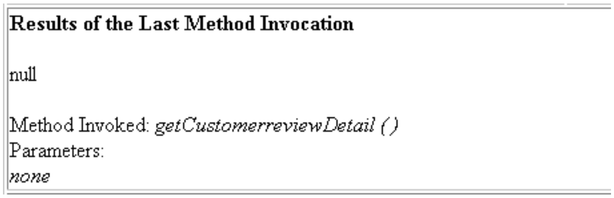
Results of the Last Method Invocation

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

Method Invoked: *getAllRestaurants ( )*
Parameters:
none

7. **Click the Invoke button on the `getCustomerreviewDetail` method.**

   The result is shown in the Results section.



Results of the Last Method Invocation

null

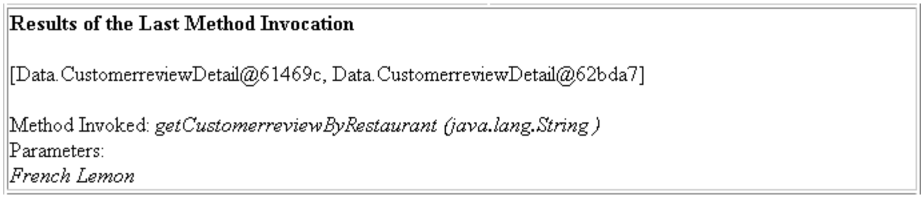
Method Invoked: *getCustomerreviewDetail ( )*
Parameters:
none

8. **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.

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

   Two `CustomerreviewDetail` records should be returned:



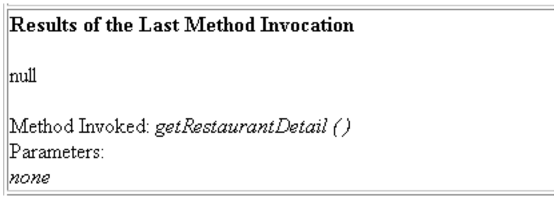
Results of the Last Method Invocation

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

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

10. **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

11. **When you are finished testing, stop the test client by pointing your web browser at another URL or by exiting the browser.**

Note – You do not need to stop the J2EE RI process. Whenever you redeploy, the IDE undeploys the application and then restarts the J2EE RI server. When you exit the IDE, a dialog box is displayed for terminating the J2EE RI instance process. However, you can manually terminate it at any time you wish by right-clicking the RI Instance 1 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:

1. **Start the PointBase console.**

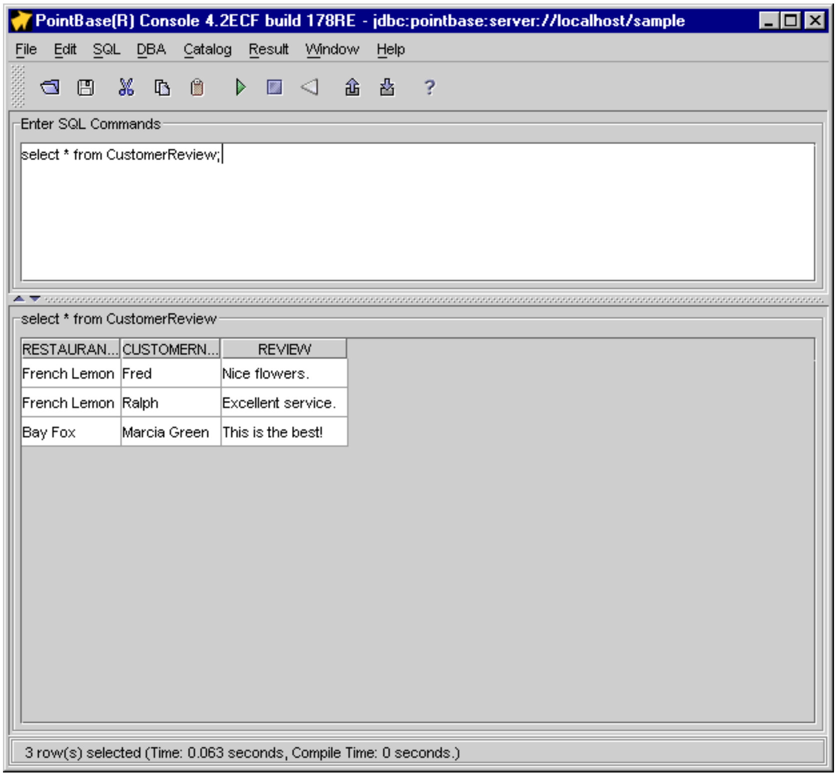
   Refer to Step 2 under "Creating the Tutorial Database Tables" on page 8.

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

```
select * from CustomerReview;
```

3. **Choose SQL → Execute to execute the statement.**

   If you entered the values in Step 4 under "Using the Test Client to Test a Session Bean" on page 74, the results should look like this:



PointBase(R) Console 4.2ECF build 178RE - jdbc:pointbase:server://localhost/sample
File Edit SQL DBA Catalog Result Window Help
Enter SQL Commands

```
select * from CustomerReview;
```

select * from CustomerReview

RESTAURAN...	CUSTOMERN...	REVIEW
French Lemon	Fred	Nice flowers.
French Lemon	Ralph	Excellent service.
Bay Fox	Marcia Green	This is the best!

3 row(s) selected (Time: 0.063 seconds, Compile Time: 0 seconds.)

   You are now ready to create the web service.

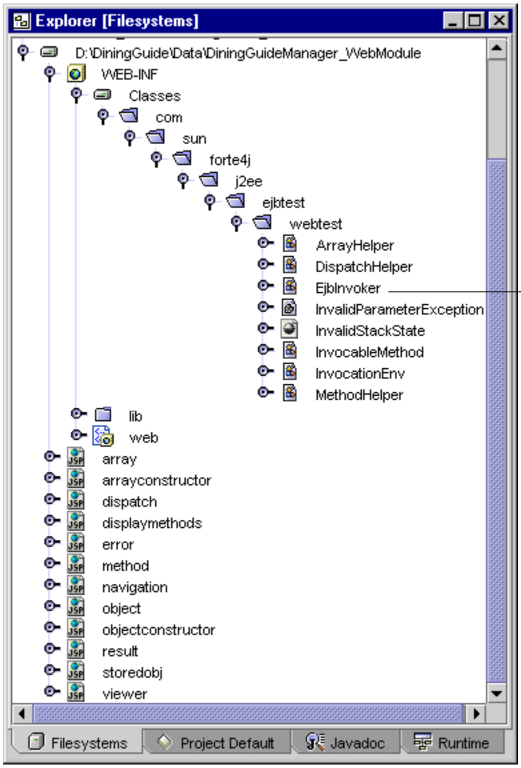
Comments on Creating a Client

Congratulations, you have successfully completed the EJB tier of the DiningGuide application. You are ready to go on to Chapter 4, to use the Forte for Java 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 Forte for Java 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.



Lookup method examples

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

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

CHAPTER 4

Creating the DiningGuide Application's Web Service

This chapter describes how to use the Forte for Java 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 Services Tier" on page 81
- "Testing the Web Service" on page 84
- "Making Your Web Service Available to Other Developers" on page 93

For a complete discussion of Forte for Java 4 web service features, see *Building Web Services* from the Forte for Java Programming series. This book is available from the Forte for Java 4 portal's Documentation page at `http://forte.sun.com/ffj/documentation/index.html`. For information on specific features, see the Forte for Java 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 `DiningGuideWebService` 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 Forte for Java 4 online help for specific web service topics and procedures.

The web service you create in the IDE is a logical entity that represents the entire set of objects in your web service, and enables you to program your web service. 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. You also use the IDE to generate supporting EJB components and any document files the web service needs to reference, such as JSP pages and HTML files. Although the DiningGuide application does not require them, for other web services you could add other types of documents, including image files.

The Runtime Classes

When you have finished programming your web service, 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 logical web service.

The Client Proxy Pages

When you deploy your web service, a client proxy is generated, which includes supporting client pages placed in the logical web service's `Documents` directory. 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 paged include a JSP page for each reference method, a JSP page to display errors for the web service, and a welcome HTML 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 page 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 Apache SOAP 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 Apache SOAP 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 Services Tier

Create the tutorial's web services tier 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 83

  The SOAP `rpcrouter` Servlet is the Apache SOAP runtime on the application server. You inform the web service of its location by specifying its URL as a property on the web service.

- "Generating the Web Service's Runtime Classes" on page 84

  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. Multitier limits web services to calling business methods only on components in an application server, whereas web-centric method calls can be on components in either an application server or a web server. 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 `DiningGuide` Filesystem and choose New → Java Package.**

   The New Package dialog box is displayed.

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

   The new `WebService` package appears under the `DiningGuide` directory.

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

   The New wizard displays the Web Service pane.

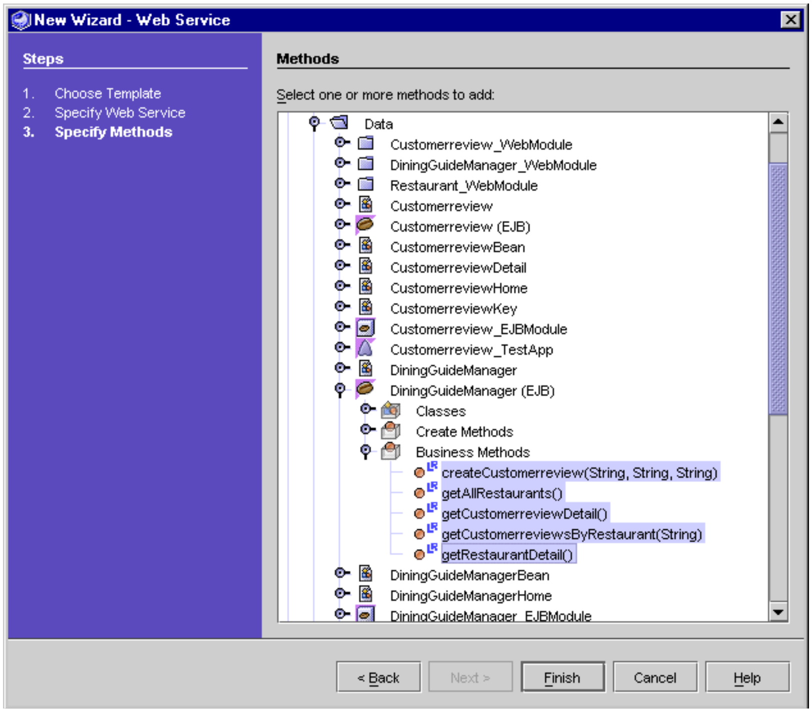
4. **Type `DiningGuideWebService` in the Name field, make sure the `Multitier` option is selected for the Architecture type, and click Next.**

   The Methods pane of the New wizard is displayed.

5. **Expand the `Data,  DiningGuide`, `DiningGuideManager(EJB)`, and `Business Methods` nodes.**

6. **Use Control-Click to select all the `DiningGuideManager`'s business methods:**

   The Methods pane looks like this:

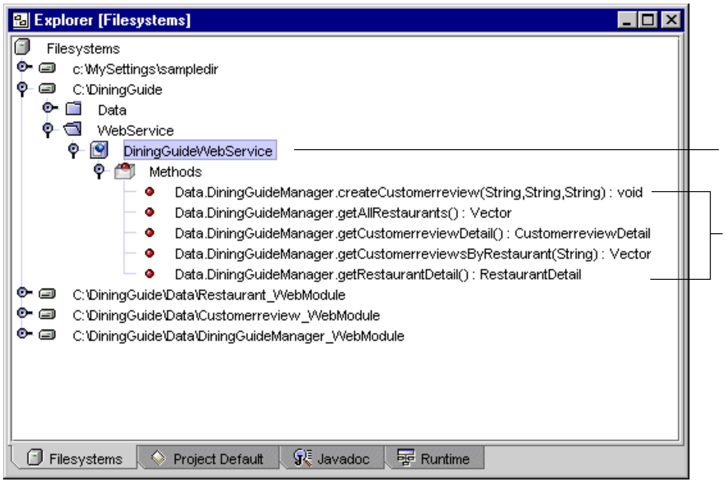


Select one or more methods to add:

- Data
  - Customerreview_WebModule
  - DiningGuideManager_WebModule
  - Restaurant_WebModule
  - Customerreview
  - Customerreview (EJB)
  - CustomerreviewBean
  - CustomerreviewDetail
  - CustomerreviewHome
  - CustomerreviewKey
  - Customerreview_EJBModule
  - Customerreview_TestApp
  - DiningGuideManager
  - DiningGuideManager (EJB)
    - Classes
    - Create Methods
    - Business Methods
      - createCustomerreview(String, String, String)
      - getAllRestaurants()
      - getCustomerreviewDetail()
      - getCustomerreviewsByRestaurant(String)
      - getRestaurantDetail()
  - DiningGuideManagerBean
  - DiningGuideManagerHome
  - DiningGuideManager_EJBModule

7. **Click Finish.**

   The new `DiningGuideWebService` 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:



```
Explorer [Filesystems]
Filesystems
  c:\MySettings\sampledir
  C:\DiningGuide
    Data
    WebService
      DiningGuideWebService
        Methods
          Data.DiningGuideManager.createCustomerreview(String,String,String) : void
          Data.DiningGuideManager.getAllRestaurants() : Vector
          Data.DiningGuideManager.getCustomerreviewDetail() : CustomerreviewDetail
          Data.DiningGuideManager.getCustomerreviewsByRestaurant(String) : Vector
          Data.DiningGuideManager.getRestaurantDetail() : RestaurantDetail
  C:\DiningGuide\Data\Restaurant_WebModule
  C:\DiningGuide\Data\Customerreview_WebModule
  C:\DiningGuide\Data\DiningGuideManager_WebModule
```

New web service

Included methods

Specifying the Web Service's SOAP RPC URL

The SOAP RPC URL property locates the SOAP `rpcrouter` Servlet of the Apache SOAP 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 86.

To set the SOAP RPC URL property:

1. **Display the properties of the `DiningGuideWebService` node.**

   Select the `DiningGuideWebService` 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 string `DiningGuideWebService` in the URL to `DiningGuideContext`, so that the entire URL is:**

   ```
   http://localhost:8000/DiningGuideContext/servlet/rpcrouter
   ```

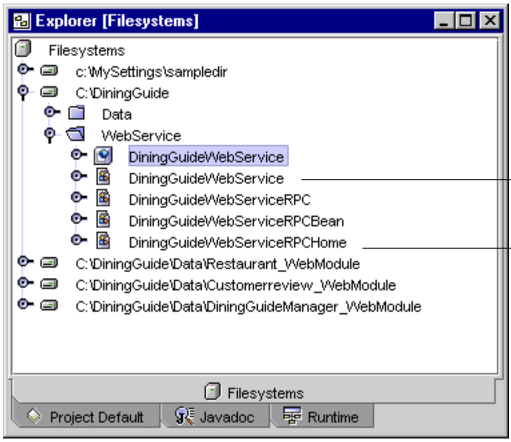
Generating the Web Service’s Runtime Classes

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 `DiningGuideWebService` node and choose Generate/Compile Java File.**

  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:



Explorer [Filesystems]

```
Filesystems
  c:\MySettings\sampledir
  C:\DiningGuide
    Data
    WebService
      DiningGuideWebService
      DiningGuideWebService
      DiningGuideWebServiceRPC
      DiningGuideWebServiceRPCBean
      DiningGuideWebServiceRPCHome
  C:\DiningGuide\Data\Restaurant_WebModule
  C:\DiningGuide\Data\Customerreview_WebModule
  C:\DiningGuide\Data\DiningGuideManager_WebModule
```

EJB components for implementing and testing the web service

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

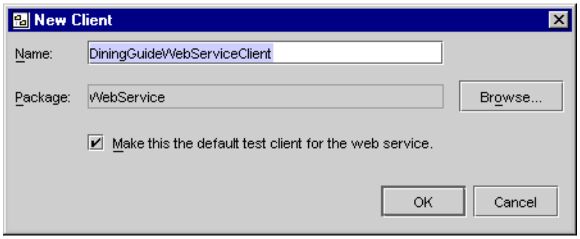
To test your web service, create a test client and a J2EE application. Add the EJB modules and the web service module to the J2EE application.

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 `DiningGuideWebService` node ( ) and choose New Client.**

   The New Client dialog box is displayed.

   New Client

   Name: DiningGuideWebServiceClient

   Package: WebService

   ☑ Make this the default test client for the web service.

   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.

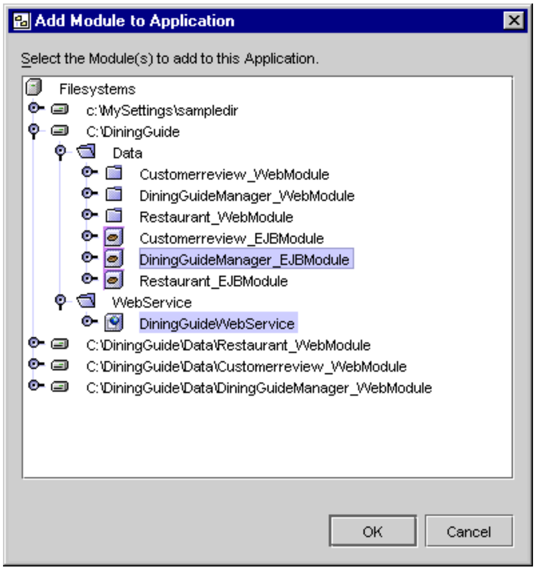
5. **Right-click the `DiningGuideApp` node and choose Add Module.**

   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 `DiningGuideWebService` nodes.**

   The dialog box looks like this:



Add Module to Application

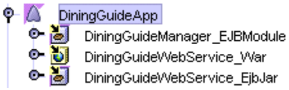
Select the Module(s) to add to this Application.

```
Filesystems
  c:\MySettings\sampledir
  C:\DiningGuide
    Data
      Customerreview_WebModule
      DiningGuideManager_WebModule
      Restaurant_WebModule
      Customerreview_EJBModule
      DiningGuideManager_EJBModule
      Restaurant_EJBModule
    WebService
      DiningGuideWebService
  C:\DiningGuide\Data\Restaurant_WebModule
  C:\DiningGuide\Data\Customerreview_WebModule
  C:\DiningGuide\Data\DiningGuideManager_WebModule
```

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

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

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



```
DiningGuideApp
  DiningGuideManager_EJBModule
  DiningGuideWebService_War
  DiningGuideWebService_EjbJar
```

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 83.

1. **Display the Properties window of the `DiningGuideWebService_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 `DiningGuideContext` for the property value.**

3. **Choose File → Save All.**

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

Deploying the Test Application

Note – Make sure the PointBase Server is running (see Step 1 under "Creating the Tutorial Database Tables" on page 8) before you deploy a J2EE application that accesses the database. In addition, make sure the J2EE RI server is *not* running outside the IDE. The deployment process automatically starts the J2EE RI server (or restarts it if it is already running).

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.

2. **Verify that the application is deployed.**

   You can read the progress of deployment in the IDE's output window, on the RI Application tab. At the end, the text there should show that deployment of `DiningGuideApp` has completed.

   The Execution window of the Explorer displays an `RI Instance 1` node.

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

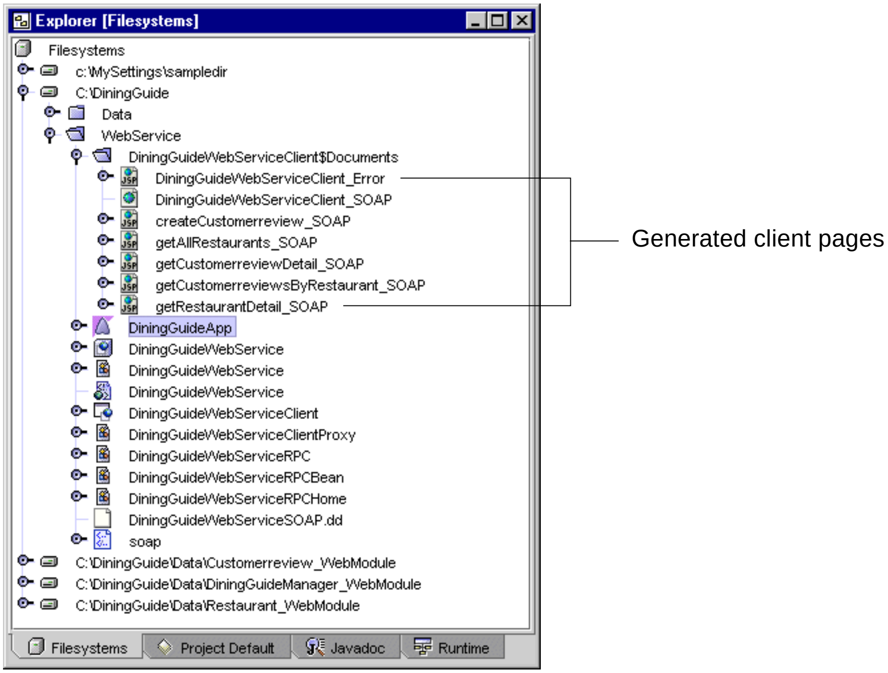
   A new `DiningGuideWebServiceClientProxy` appears in the Explorer.

4. **Expand the `DiningGuideWebServiceClient$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:



- Filesystems
  - c:\MySettings\sampledir
  - C:\DiningGuide
    - Data
    - WebService
      - DiningGuideWebServiceClient$Documents
        - DiningGuideWebServiceClient_Error
        - DiningGuideWebServiceClient_SOAP
        - createCustomerreview_SOAP
        - getAllRestaurants_SOAP
        - getCustomerreviewDetail_SOAP
        - getCustomerreviewsByRestaurant_SOAP
        - getRestaurantDetail_SOAP
      - DiningGuideApp
      - DiningGuideWebService
      - DiningGuideWebService
      - DiningGuideWebService
      - DiningGuideWebServiceClient
      - DiningGuideWebServiceClientProxy
      - DiningGuideWebServiceRPC
      - DiningGuideWebServiceRPCBean
      - DiningGuideWebServiceRPCHome
      - DiningGuideWebServiceSOAP.dd
      - soap
  - C:\DiningGuide\Data\Customerreview_WebModule
  - C:\DiningGuide\Data\DiningGuideManager_WebModule
  - C:\DiningGuide\Data\Restaurant_WebModule

Generated client pages

These files are also referenced under the `Generated Documents` node under the `DiningGuideWebServiceClient` node.

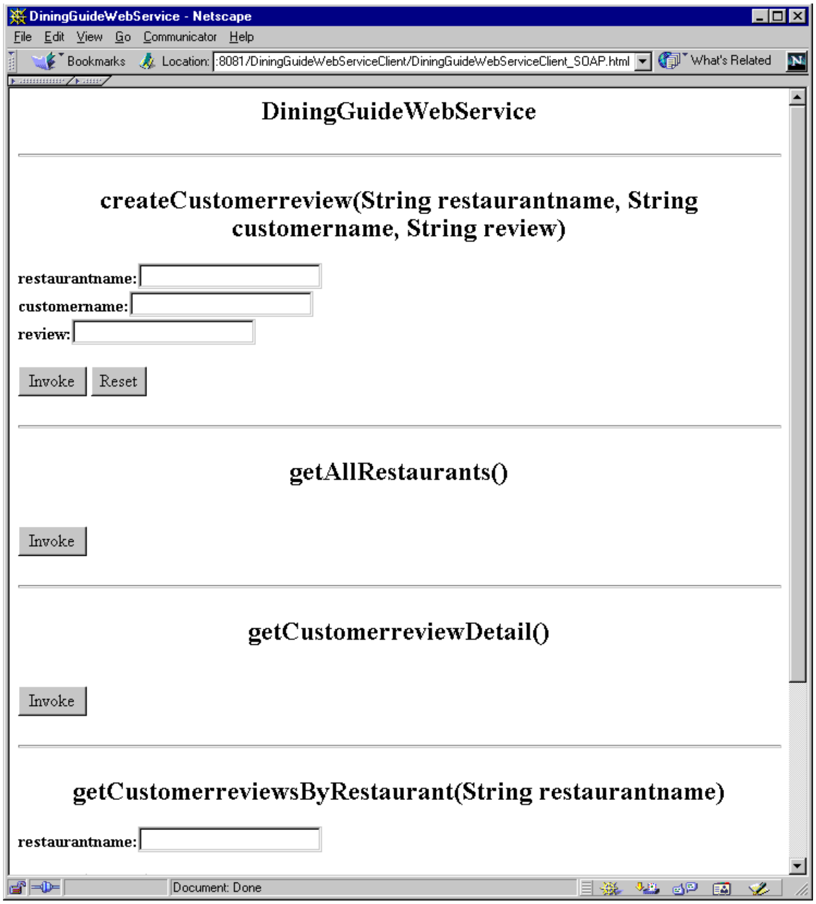
Using the Test Application to Test the Web Service

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 Forte for Java 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 `DiningGuideWebServiceClient` node ( ) and choose Execute.**

   The IDE automatically starts the built-in Tomcat web server, launches the default web browser, and displays the client's generated welcome page (`DiningGuideWebServiceClient_SOAP.html`):



DiningGuideWebService - Netscape
File Edit View Go Communicator Help
Bookmarks Location: :8081/DiningGuideWebServiceClient/DiningGuideWebServiceClient_SOAP.html
What's Related
Document: Done

DiningGuideWebService

createCustomerreview(String restaurantname, String customername, String review)

restaurantname:

customername:

review:

getAllRestaurants()

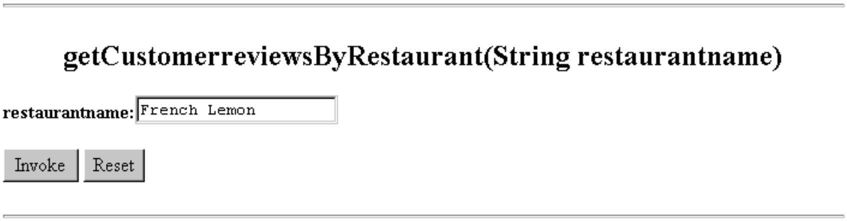
getCustomerreviewDetail()

getCustomerreviewsByRestaurant(String restaurantname)

restaurantname:

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

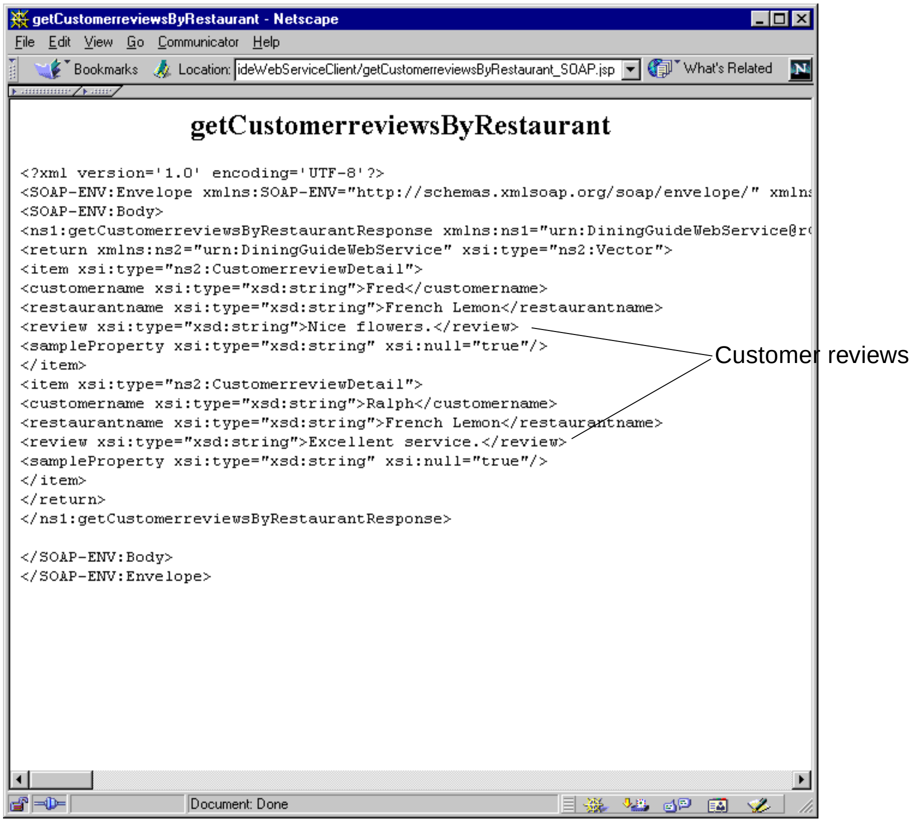
2. **Test the `getCustomerreviewsByRestaurant` by typing `French Lemon` in the text field and clicking the Invoke button.**



getCustomerreviewsByRestaurant(String restaurantname)

restaurantname: French Lemon

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.


getCustomerreviewsByRestaurant - Netscape
File Edit View Go Communicator Help
Bookmarks
Location: ideWebServiceClient/getCustomerreviewsByRestaurant_SOAP.jsp
What's Related
Document: Done
Customer reviews

getCustomerreviewsByRestaurant

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns
<SOAP-ENV:Body>
<ns1:getCustomerreviewsByRestaurantResponse xmlns:ns1="urn:DiningGuideWebService@r
<return xmlns:ns2="urn:DiningGuideWebService" xsi:type="ns2:Vector">
<item xsi:type="ns2:CustomerreviewDetail">
<customername xsi:type="xsd:string">Fred</customername>
<restaurantname xsi:type="xsd:string">French Lemon</restaurantname>
<review xsi:type="xsd:string">Nice flowers.</review>
<sampleProperty xsi:type="xsd:string" xsi:null="true"/>
</item>
<item xsi:type="ns2:CustomerreviewDetail">
<customername xsi:type="xsd:string">Ralph</customername>
<restaurantname xsi:type="xsd:string">French Lemon</restaurantname>
<review xsi:type="xsd:string">Excellent service.</review>
<sampleProperty xsi:type="xsd:string" xsi:null="true"/>
</item>
</return>
</ns1:getCustomerreviewsByRestaurantResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The data includes all the records entered for the French Lemon restaurant. Refer to TABLE 1-6 to verify the data. Or you can verify the data by starting the PointBase console and running the following SQL statement:

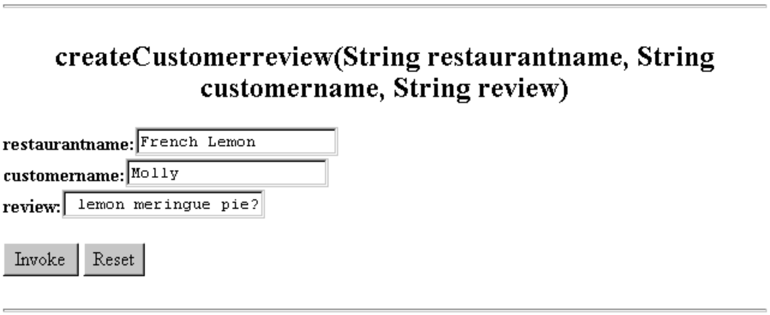
```
select * from CustomerReview;
```

The results show what CustomerReview records you have entered.

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

4. **Test the `createCustomerreview` operation by typing `French Lemon` in the `restaurantname` field, and whatever you want in the other two fields.**

   For example:



createCustomerreview(String restaurantname, String customername, String review)

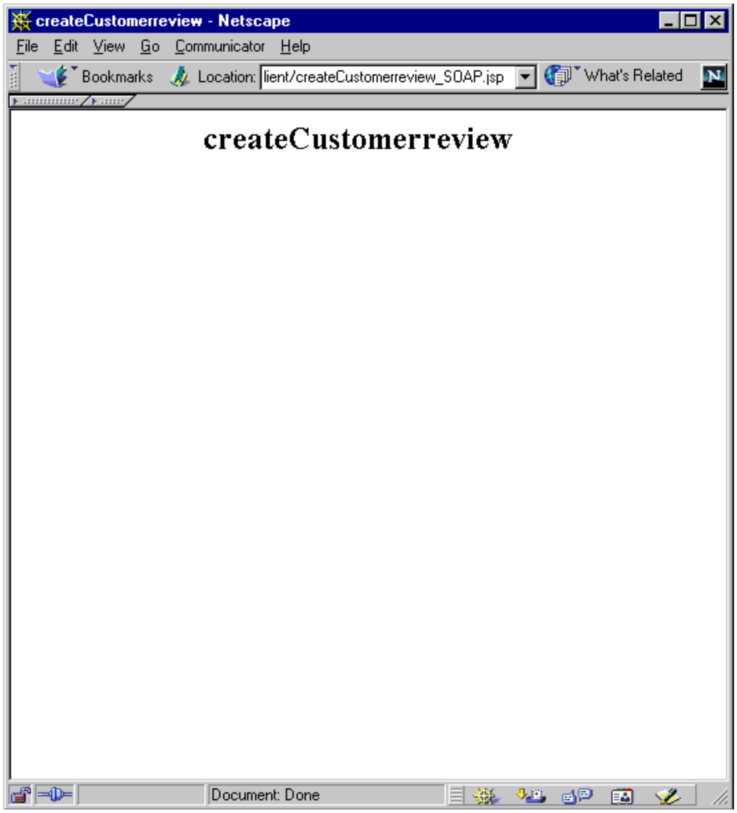
restaurantname: French Lemon

customername: Molly

review: lemon meringue pie?

5. **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 XML and passed to the SOAP protocol, which sends a request to the web service tier. The service receives the request and enters it into the database. This method returns a void, so the JSP page is blank:



createCustomerreview - Netscape

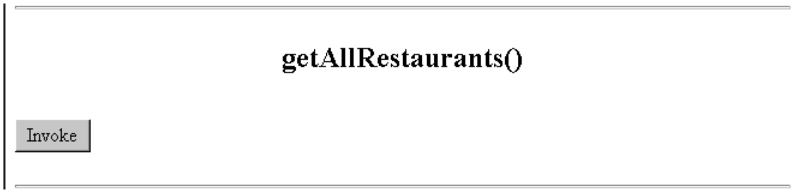
Location: lient/createCustomerreview_SOAP.jsp

createCustomerreview

Document: Done

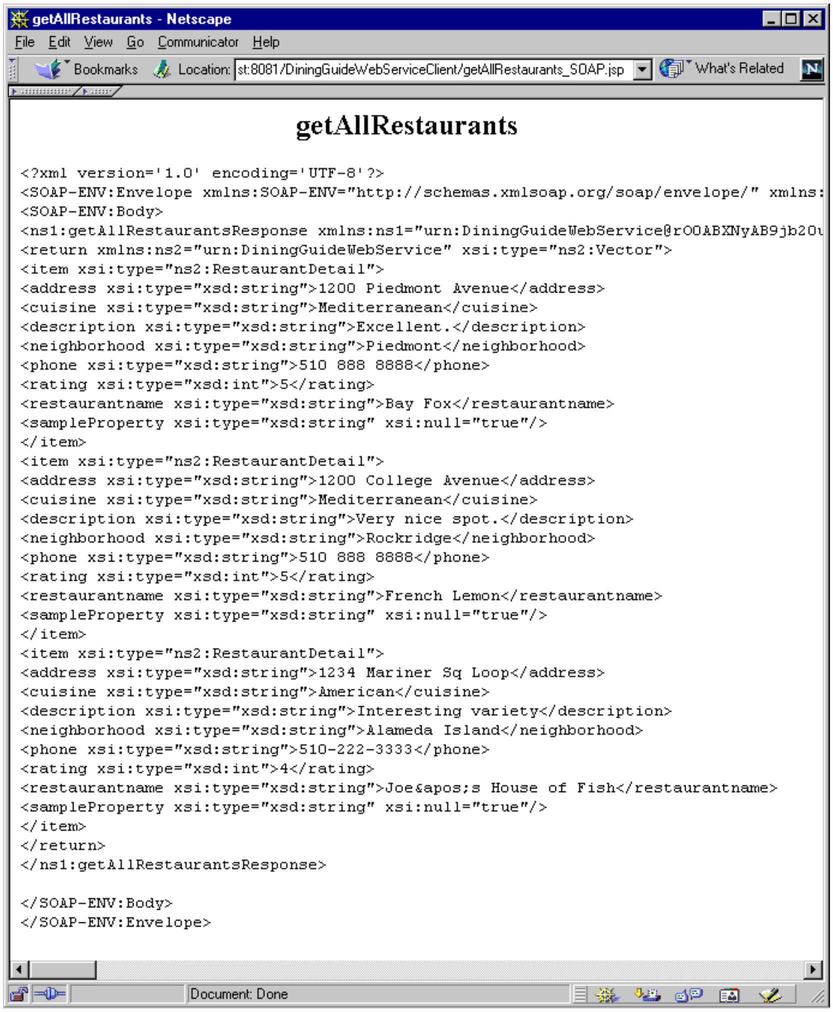
6. **Use the Back button on your browser to return to the welcome page.**

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



getAllRestaurants()

This method does not require an input parameter. It returns a collection of restaurant data, which the `getAllRestaurants_SOAP.jsp` page displays as XML:



getAllRestaurants - Netscape
File Edit View Go Communicator Help
Bookmarks
Location:
What's Related
Document: Done

Location: st:8081/DiningGuideWebServiceClient/getAllRestaurants_SOAP.jsp

getAllRestaurants

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:
<SOAP-ENV:Body>
<ns1:getAllRestaurantsResponse xmlns:ns1="urn:DiningGuideWebService@rO0ABXNyAB9jb20
<return xmlns:ns2="urn:DiningGuideWebService" xsi:type="ns2:Vector">
<item xsi:type="ns2:RestaurantDetail">
<address xsi:type="xsd:string">1200 Piedmont Avenue</address>
<cuisine xsi:type="xsd:string">Mediterranean</cuisine>
<description xsi:type="xsd:string">Excellent.</description>
<neighborhood xsi:type="xsd:string">Piedmont</neighborhood>
<phone xsi:type="xsd:string">510 888 8888</phone>
<rating xsi:type="xsd:int">5</rating>
<restaurantname xsi:type="xsd:string">Bay Fox</restaurantname>
<sampleProperty xsi:type="xsd:string" xsi:null="true"/>
</item>
<item xsi:type="ns2:RestaurantDetail">
<address xsi:type="xsd:string">1200 College Avenue</address>
<cuisine xsi:type="xsd:string">Mediterranean</cuisine>
<description xsi:type="xsd:string">Very nice spot.</description>
<neighborhood xsi:type="xsd:string">Rockridge</neighborhood>
<phone xsi:type="xsd:string">510 888 8888</phone>
<rating xsi:type="xsd:int">5</rating>
<restaurantname xsi:type="xsd:string">French Lemon</restaurantname>
<sampleProperty xsi:type="xsd:string" xsi:null="true"/>
</item>
<item xsi:type="ns2:RestaurantDetail">
<address xsi:type="xsd:string">1234 Mariner Sq Loop</address>
<cuisine xsi:type="xsd:string">American</cuisine>
<description xsi:type="xsd:string">Interesting variety</description>
<neighborhood xsi:type="xsd:string">Alameda Island</neighborhood>
<phone xsi:type="xsd:string">510-222-3333</phone>
<rating xsi:type="xsd:int">4</rating>
<restaurantname xsi:type="xsd:string">Joe&apos;s House of Fish</restaurantname>
<sampleProperty xsi:type="xsd:string" xsi:null="true"/>
</item>
</return>
</ns1:getAllRestaurantsResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Notice the Restaurant record you entered when you tested the Restaurant entity bean (see "Using the Test Client to Test the Entity Bean" on page 53) 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 `DiningGuideWebService` node () and choose Generate WSDL.**

   A WSDL file (the node with a green sphere) named `DiningGuideWebService` is created under the `WebService` package.

   You can find this file on your operating system's file system, named `DiningGuideWebService.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 `DiningGuideWebService.wsdl` file in it.**

2. **In the Forte for Java 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 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.**

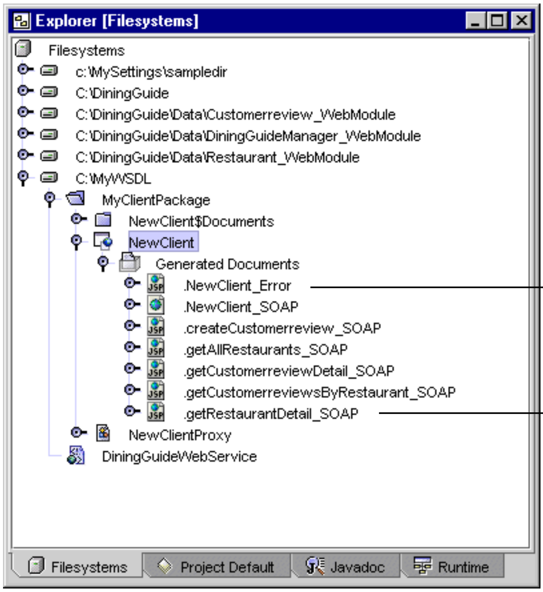
    The Select Local WSDL File pane of the New wizard is displayed.

11. **Select the `DiningGuideWebService` WSDL file in the `MyClientPackage` package under your mounted directory 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 and a `MyClientProxy` bean are generated in the Explorer. The expanded `Generated Documents` node reveals the JSP pages and welcome page required for the client, as shown:



Explorer [Filesystems]

```
Filesystems
  c:\MySettings\sampledir
  C:\DiningGuide
  C:\DiningGuide\Data\Customerreview_WebModule
  C:\DiningGuide\Data\DiningGuideManager_WebModule
  C:\DiningGuide\Data\Restaurant_WebModule
  C:\MyWSDL
    MyClientPackage
      NewClient$Documents
      NewClient
        Generated Documents
          .NewClient_Error
          .NewClient_SOAP
          .createCustomerreview_SOAP
          .getAllRestaurants_SOAP
          .getCustomerreviewDetail_SOAP
          .getCustomerreviewsByRestaurant_SOAP
          .getRestaurantDetail_SOAP
      NewClientProxy
    DiningGuideWebService
```

Generated support files

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 88.

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. Forte for Java 4 provides a single-user internal UDDI registry to test this process, and the StockApp example, available from the Forte for Java 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*.

CHAPTER 5

Creating a Client for the Tutorial Application

This chapter shows you how to run the DiningGuide application using a provided Swing client that communicates with the web service you created in Chapter 4.

The provided client contains two Swing classes, `RestaurantTable` and `CustomerReviewTable`. The code for these classes is provided in Appendix A. You will create two classes and replace their default code with the code you copy and paste from Appendix A. You 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 99
- "Examining the Client Code" on page 102

Creating the Client With the Provided Code

To use the provided code to create your client, create two classes, then replace their entire code with the source code provided in 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.

To create the two classes:

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

   The `RestaurantTable` class should compile without errors.

8. **Repeat Step 5, copying all the code from "`CustomerReviewTable.java` Source" on page 130 and the following three pages into the** `CustomerReviewTable` **class body.**

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

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

    The `CustomerReviewTable` 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.

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**, and the** J2EE Reference Implementation 1.3.1 **nodes.**

   Because you deployed your web service previously (see "Deploying the Test Application" on page 87), you do not need to redeploy it. However, the server where your web service is deployed must be running. Step 3 takes care of this issue.

3. **Right-click the** RI Instance 1 **node and choose Start Server if the command is activated.**

   If the command is not activated (dimmed), do nothing, because the J2EE RI server is already running.

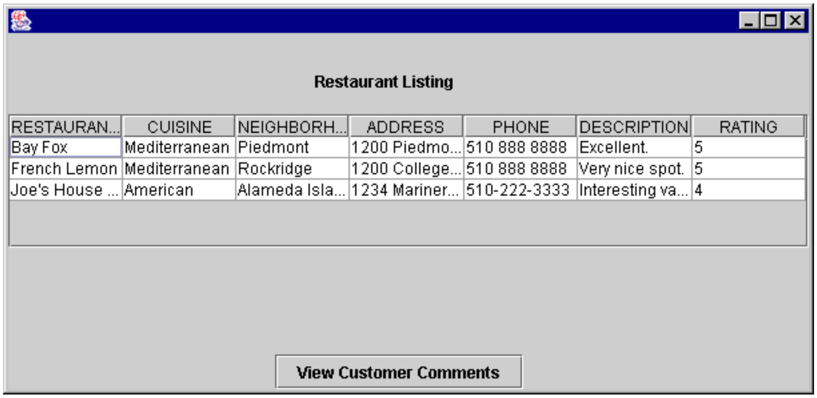
   The PointBase server must also be running. If your PointBase server is already running, skip Step 4 and continue with Step 5.

4. **If the PointBase Network Server is not running, choose Tools → PointBase Network Server → Start Server.**

5. **Click the Filesystems tab of the Explorer.**

6. **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:

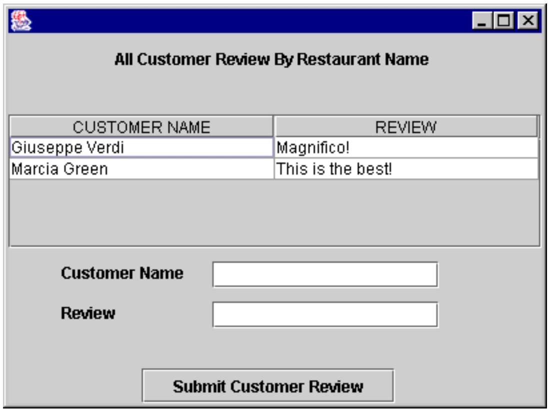


Restaurant Listing

RESTAURAN...	CUISINE	NEIGHBORH...	ADDRESS	PHONE	DESCRIPTION	RATING
Bay Fox	Mediterranean	Piedmont	1200 Piedmo...	510 888 8888	Excellent.	5
French Lemon	Mediterranean	Rockridge	1200 College...	510 888 8888	Very nice spot.	5
Joe's House ...	American	Alameda Isla...	1234 Mariner...	510-222-3333	Interesting va...	4

7. **Select any restaurant in the table and Click the View Customer Comments button.**

   For example, select the Bay Fox 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.

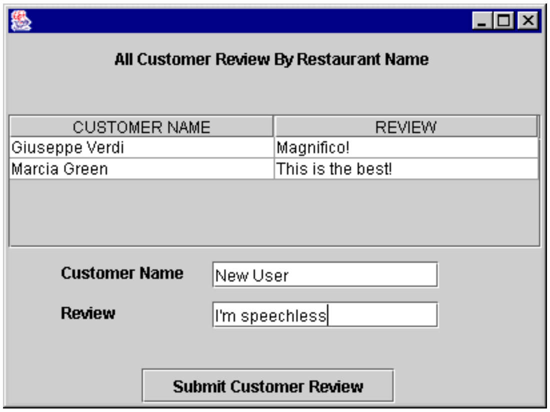


All Customer Review By Restaurant Name

CUSTOMER NAME	REVIEW
Giuseppe Verdi	Magnifico!
Marcia Green	This is the best!

8. **Type a something in the `Customer Name` field and in the `Review` field and click the Submit Customer Review button.**

   For example:



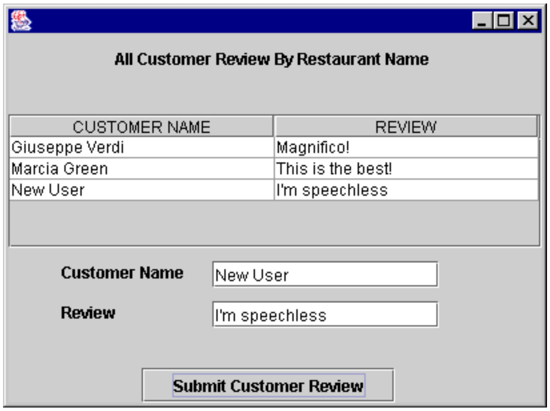
All Customer Review By Restaurant Name

CUSTOMER NAME	REVIEW
Giuseppe Verdi	Magnifico!
Marcia Green	This is the best!

Customer Name: New User

Review: I'm speechless

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



All Customer Review By Restaurant Name

CUSTOMER NAME	REVIEW
Giuseppe Verdi	Magnifico!
Marcia Green	This is the best!
New User	I'm speechless

Customer Name: New User

Review: I'm speechless

9. **Play around with the features, as described in "User's View of the Tutorial Application" on page 15.**

10. **Quit the application by closing any window.**

    After you quit the application, the execution window shows that the J2EE RI server process is still running. You need not stop the J2EE RI 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 J2EE RI server or the Tomcat 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 102
- "Displaying Customer Review Data for a Selected Restaurant" on page 104
- "Creating a New Customer Review Record" on page 106

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 {
        DiningGuideWebServiceClientProxy restaurantCP = new
        DiningGuideWebServiceClientProxy();
        restList =
(java.util.Vector)restaurantCP.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) {//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
```

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 {
        DiningGuideWebServiceClientProxy custCP = new
        DiningGuideWebServiceClientProxy();
        custList =
(java.util.Vector)custCP.getCustomerreviewsByRestaurant(restaura
ntname);
    }
    catch (Exception ex) {
        System.err.println("Caught an exception." );
        ex.printStackTrace();
    }
    return custList;
}
```

4. The review data is passed up to the `CustomerReviewTable.putDataToTable` method, which iterates through it and populates the table.

```
public void putDataToTable(java.lang.String restaurantname) {
    RestaurantName = restaurantname;
    java.util.Vector customerList =
getCustomerReviewByName(restaurantname);
    Iterator j=customerList.iterator();
    while (j.hasNext()) {
        CustomerreviewDetail ci = (CustomerreviewDetail)j.next();
        String[] str = {ci.getCustomername(),ci.getReview()
        };
        TableModel.addRow(str);
    }
}
```

5. The `RestaurantTable.jButton1ActionPerformed` method then displays the data.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent
evt) {//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
```

Creating a New Customer Review Record

When the user types a name and review comments on the Customer Review window and clicks the Submit Customer Review button, the `CustomerReviewTable`'s `jButton1ActionPerformed` method creates the review record in the database by means of the 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.jButton1ActionPerformed` 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 jButton1ActionPerformed(java.awt.event.ActionEvent
evt) {
    try {
        DiningGuideWebServiceClientProxy reviewCP = new
        DiningGuideWebServiceClientProxy();
        reviewCP.createCustomerreview(RestaurantName,
customerNameField.getText(),reviewField.getText());
    }
    catch (Exception ex) {
        System.err.println("Caught an exception." );
        ex.printStackTrace();
    }
    refreshView();
}
```

2. This same method (`jButton1ActionPerformed`) calls the `CustomerReviewTable.refreshView` method.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent
evt) {
    try {
        DiningGuideWebServiceClientProxy reviewCP = new
        DiningGuideWebServiceClientProxy();
        reviewCP.createCustomerreview(RestaurantName,
customerNameField.getText(),reviewField.getText());
    }
    catch (Exception ex) {
        System.err.println("Caught an exception." );
        ex.printStackTrace();
    }
    refreshView();
}
```

3. The CustomerReviewTable.refreshView method calls the putDataToTable method, passing it the restaurant name.

```
void refreshView() {
    try{
        while(TableModel.getRowCount()>0) {
            TableModel.removeRow(0);
        }
        putDataToTable(RestaurantName);
        repaint();
    }
    catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

4. The CustomerReviewTable.putDataToTable method populates the table.

```
public void putDataToTable(java.lang.String restaurantname) {
    RestaurantName = restaurantname;
    java.util.Vector customerList =
getCustomerReviewByName(restaurantname);
    Iterator j=customerList.iterator();
    while (j.hasNext()) {
        CustomerreviewDetail ci = (CustomerreviewDetail)j.next();
        String[] str = {ci.getCustomername(),ci.getReview()
        };
        TableModel.addRow(str);
    }
}
```

5. Then the `CustomerReviewTable.refreshView` method repaints the window, showing the new data.

```
void refreshView() {
    try{
        while(TableModel.getRowCount()>0) {
            TableModel.removeRow(0);
        }
        putDataToTable(RestaurantName);
        repaint();
    }
    catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

APPENDIX **A**

DiningGuide Source Files

This appendix displays the code for the following DiningGuide components:

- EJB tier components:
  - "`RestaurantBean.java` Source" on page 112
  - "`RestaurantDetail.java` Source" on page 115
  - "`CustomerreviewBean.java` Source" on page 119
  - "`CustomerreviewDetail.java` Source" on page 121
  - "`DiningGuideManagerBean.java` Source" on page 123
- Client components:
  - "`RestaurantTable.java` Source" on page 126
  - "`CustomerReviewTable.java` Source" on page 130

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 Forte for Java 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.

`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 java.lang.Integer getRating();
    public abstract void setRating(java.lang.Integer 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,
java.lang.Integer rating) throws javax.ejb.CreateException {
        if (restaurantname == null) {
            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, java.lang.Integer 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 Integer 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, java.lang.Integer 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 Integer getRating() {
        return this.rating;
    }

    /** Setter for property rating.
     * @param rating New value of property rating.
     */
    public void setRating(Integer 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)) {
            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("Restaurant");
                myRestaurantHome =
                (RestaurantHome)javax.rmi.PortableRemoteObject.narrow(result,
                RestaurantHome.class);
            }
            catch (Exception e) {System.out.println("Error: "+ e); }
        }
        Context crc = null;
        Object crresult = null;
        if (this.myCustomerreviewHome == null) {
            try {
                crc = new InitialContext();
                result = crc.lookup("Customerreview");
                myCustomerreviewHome =
             (CustomerreviewHome)javax.rmi.PortableRemoteObject.narrow(result,
                CustomerreviewHome.class);
            }
            catch (Exception e) {System.out.println("Error: "+ e); }
        }
    }

    public java.util.Vector getAllRestaurants() {
        System.out.println("Entering
DiningGuideManagerEJB.getAllRestaurants()");
        java.util.Vector restaurantList = new java.util.Vector();
        try {
            java.util.Collection rl = myRestaurantHome.findAll();
            if (rl == null) { restaurantList = null; }
            else {
                RestaurantDetail rd;
                java.util.Iterator rli = rl.iterator();
                while ( rli.hasNext() ) {
                    rd =((Restaurant)rli.next()).getRestaurantDetail();
                    System.out.println(rd.getRestaurantname());
                    System.out.println(rd.getRating());
                    restaurantList.addElement(rd);
                }
            }
        } catch (Exception e) {
            System.out.println("Error in
DiningGuideManagerEJB.getAllRestaurants(): " + e);
        }
      System.out.println("Leaving DiningGuideManagerEJB.getAllRestaurants()");
        return restaurantList;
    }

    public java.util.Vector getCustomerreviewsByRestaurant(java.lang.String
restaurantname) {
        System.out.println("Entering
DiningGuideManagerEJB.getCustomerreviewsByRestaurant()");
        java.util.Vector reviewList = new java.util.Vector();
        try {
            java.util.Collection rl =
            myCustomerreviewHome.findByRestaurantName(restaurantname);
            if (rl == null) { reviewList = null; }
            else {
                CustomerreviewDetail crd;
                java.util.Iterator rli = rl.iterator();
                while ( rli.hasNext() ) {
                    crd =
                    ((Customerreview)rli.next()).getCustomerreviewDetail();
                    System.out.println(crd.getRestaurantname());
                    System.out.println(crd.getCustomername());
                    System.out.println(crd.getReview());
                    reviewList.addElement(crd);
                }
            }
        } catch (Exception e) {
            System.out.println("Error in
DiningGuideManagerEJB.getCustomerreviewsByRestaurant(): " + e);
        }
        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) {
            System.out.println("Error in
DiningGuideManagerEJB.createCustomerreview(): " + e);
        }
        System.out.println("Leaving
DiningGuideManagerEJB.createCustomerreview()");
    }

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

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

RestaurantTable.java Source

```
package WebService;
import javax.swing.table.*;
import  java.util.*;
import Data.*;

public class RestaurantTable extends javax.swing.JFrame {
    /** Creates new form RestaurantTable */
    public RestaurantTable() {
        initComponents();
        restaurantList=getAllRestaurants();
        putDataToTable();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    private void initComponents() {//GEN-BEGIN:initComponents
        jButton1 = new javax.swing.JButton();
        jScrollPane1 = new javax.swing.JScrollPane();
        jTable1 = new javax.swing.JTable();
        jLabel1 = new javax.swing.JLabel();

        getContentPane().setLayout(new
org.netbeans.lib.awtextra.AbsoluteLayout());

        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
        });

        jButton1.setText("View Customer Comments");
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });

        getContentPane().add(jButton1, new
org.netbeans.lib.awtextra.AbsoluteConstraints(200, 240, -1, -1));

        TableModel = (new javax.swing.table.DefaultTableModel (
            new Object [][] {

            },
            new String [] {
             "RESTAURANT NAME", "CUISINE", "NEIGHBORHOOD", "ADDRESS", "PHONE",
"DESCRIPTION", "RATING"
            }
        ) {
            Class[] types = new Class [] {
                java.lang.String.class, java.lang.String.class,
java.lang.String.class,
java.lang.String.class,java.lang.String.class,java.lang.String.class,java.lang
.String.class
            };

            public Class getColumnClass (int columnIndex) {
                return types [columnIndex];
            }
        });
        jTable1.setModel(TableModel);
        jScrollPane1.setViewportView(jTable1);

        getContentPane().add(jScrollPane1, new
org.netbeans.lib.awtextra.AbsoluteConstraints(0, 60, 600, 100));

        jLabel1.setText("Restaurant Listing");
        getContentPane().add(jLabel1, new
org.netbeans.lib.awtextra.AbsoluteConstraints(230, 20, 110, 30));

        pack();
    }//GEN-END:initComponents

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_jButton1ActionPerformed
        int r =jTable1.getSelectedRow();
                int c = jTable1.getSelectedColumnCount();

                String i =(String)TableModel.getValueAt(r,0);
                CustomerReviewTable crt = new CustomerReviewTable();
                crt.putDataToTable(i);
                crt.show();
             System.out.println(i);
    }//GEN-LAST:event_jButton1ActionPerformed

    /** Exit the Application */
    private void exitForm(java.awt.event.WindowEvent evt)
{//GEN-FIRST:event_exitForm
        System.exit(0);
    }//GEN-LAST:event_exitForm

     private void putDataToTable()
    {
        Iterator j=restaurantList.iterator();
            while (j.hasNext()) {
                RestaurantDetail ci = (RestaurantDetail)j.next();
                String strRating = null;
                String[] str = 
{ci.getRestaurantname(),ci.getCuisine(),ci.getNeighborhood(),ci.getAddress(),
ci.getPhone(),ci.getDescription(),
                                strRating.valueOf(ci.getRating()),
                               };
                TableModel.addRow(str);
            }
    }
      private Vector getAllRestaurants()
    {
         Vector restList = new Vector();
        try
        {
            DiningGuideWebServiceClientProxy restaurantCP = new 
DiningGuideWebServiceClientProxy();
            restList = (java.util.Vector)restaurantCP.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
        {
            DiningGuideWebServiceClientProxy restaurantCP = new 
DiningGuideWebServiceClientProxy();
            reviewList = 
(java.util.Vector)restaurantCP.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 Data.*;

public class CustomerReviewTable extends javax.swing.JFrame {

    /** Creates new form CustomerReviewTable */
    public CustomerReviewTable() {
        initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    private void initComponents() {//GEN-BEGIN:initComponents
        jScrollPane1 = new javax.swing.JScrollPane();
        jTable1 = new javax.swing.JTable();
        jButton1 = new javax.swing.JButton();
        customerNameLabel = new javax.swing.JLabel();
        customerNameField = new javax.swing.JTextField();
        reviewLabel = new javax.swing.JLabel();
        reviewField = new javax.swing.JTextField();
        jLabel1 = new javax.swing.JLabel();
        getContentPane().setLayout(new
org.netbeans.lib.awtextra.AbsoluteLayout());

        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
        });

        TableModel = (new javax.swing.table.DefaultTableModel (
            new Object [][] {

            },
            new String [] {
                "CUSTOMER NAME", "REVIEW"
            }
        ) {
            Class[] types = new Class [] {
                java.lang.String.class,java.lang.String.class
            };

            public Class getColumnClass (int columnIndex) {
                return types [columnIndex];
            }
        });
        jTable1.setModel(TableModel);
        jScrollPane1.setViewportView(jTable1);

        getContentPane().add(jScrollPane1, new
org.netbeans.lib.awtextra.AbsoluteConstraints(0, 60, 400, 100));

        jButton1.setText("Submit Customer Review");
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });

        getContentPane().add(jButton1, new
org.netbeans.lib.awtextra.AbsoluteConstraints(100, 250, 190, -1));

        customerNameLabel.setText("Customer Name");
        getContentPane().add(customerNameLabel, new
org.netbeans.lib.awtextra.AbsoluteConstraints(40, 170, -1, -1));

        getContentPane().add(customerNameField, new
org.netbeans.lib.awtextra.AbsoluteConstraints(153, 170, 170, -1));

        reviewLabel.setText("Review");
        getContentPane().add(reviewLabel, new
org.netbeans.lib.awtextra.AbsoluteConstraints(40, 200, 80, -1));

        getContentPane().add(reviewField, new
org.netbeans.lib.awtextra.AbsoluteConstraints(153, 200, 170, 20));

        jLabel1.setText("All Customer Review By Restaurant Name");
        getContentPane().add(jLabel1, new
org.netbeans.lib.awtextra.AbsoluteConstraints(80, 10, 240, -1));

        pack();
    }//GEN-END:initComponents

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_jButton1ActionPerformed

        try {
            DiningGuideWebServiceClientProxy reviewCP = new
DiningGuideWebServiceClientProxy();
            reviewCP.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 {
            DiningGuideWebServiceClientProxy custCP = new
DiningGuideWebServiceClientProxy();
            custList =
(java.util.Vector)custCP.getCustomerreviewsByRestaurant(restaurantname);
        }
        catch (Exception ex) {
            System.err.println("Caught an exception." );
            ex.printStackTrace();
        }
        return custList;
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        new CustomerReviewTable().show();
    }

    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JLabel reviewLabel;
    private javax.swing.JButton jButton1;
    private javax.swing.JScrollPane jScrollPane1;
    private javax.swing.JTextField customerNameField;
    private javax.swing.JTable jTable1;
    private javax.swing.JLabel customerNameLabel;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JTextField reviewField;
    // End of variables declaration//GEN-END:variables
    private DefaultTableModel TableModel;
    private java.lang.String RestaurantName = null;
    //private java.util.Vector restaurantList = null;
}
```

APPENDIX **B**

DiningGuide Database Script

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

```
drop table CustomerReview;
drop table Restaurant;

create table Restaurant(
    restaurantNamevarchar(80),
    cuisine varchar(25),
    neighborhoodvarchar(25),
    address varchar(30),
    phone  varchar(12),
    descriptionvarchar(200),
    rating tinyint,
constraint pk_Restaurant primary key(restaurantName));

create table CustomerReview(
    restaurantNamevarchar(80) not null references Restaurant(restaurantName),
    customerName varchar(25),
    review varchar(200),
constraint pk_CustomerReview primary key(customerName, restaurantName));

insert into Restaurant (restaurantName, cuisine, neighborhood, address, phone,
description, rating) values ('French Lemon','Mediterranean','Rockridge', '1200
College Avenue', '510 888 8888', 'Very nice spot.', 5);
insert into Restaurant (restaurantName, cuisine, neighborhood, address, phone,
description, rating) values
('Bay Fox','Mediterranean','Piedmont', '1200 Piedmont Avenue', '510 888 8888',
'Excellent.', 5);

insert into CustomerReview (restaurantName, customerName, review) values
('French Lemon','Fred', 'Nice flowers.');
insert into Customerreview (restaurantname, customername, review) values
('French Lemon','Ralph', 'Excellent service.');
```

Index