Chapter 1

Creating your first Web service and Web application

Chapter Contents

- Introducing Web service terminology
- Installing WebSphere Application Server and Rational Developer
- Setting up a Web project
- Creating a Web service
- Creating a Web application

Time Required

- 2 hours

Files Required

- The /solutions/chap01/samples/StockQuote.java file

Web services are network-accessible programs that use a standardized messaging protocol to communicate with other programs that want to use their functions.

Web services reside on application servers, such as WebSphere Application Server, which can be on the Internet, on your company's private intranet, or on your own computer. The functions that Web services provide can be general-purpose, such as looking up the weather forecast for a particular city, or application-specific, such as creating an order for an e-commerce application. Web services are also *self-describing*; that is, they include special descriptions of the functions that they provide and how to access those functions. Application programmers use these descriptions to find (*discover*) and use the services.

Web applications also reside on application servers. These applications consist of one or more Web pages that users access through a Web browser. They typically contain a combination of static and dynamic content, such as text, images, and programs that run on the server or in the user's Web browser. Web applications can use Web services that reside on the same server, or on any server in the network that the Web application has access to. Unless you tell them, users at a Web browser aren't aware that the work might be distributed across multiple servers.

Notice that we haven't said anything about what programming language a Web service is written in, or what operating system it's running on. That's because it doesn't matter. The Web service just needs to be available on the network that the application is using, and the service and the application need to use the correct protocol to communicate with one another. The application can be written in a different programming language than the service, and run on a different operating system.

In this chapter, you'll create a simple Web service and Web application that happen to be written in the same programming language (Java) and happen to run on the same machine (yours). In later chapters, however, you'll use publicly available Web services on the Internet, where you don't know what language the Web service is written in or what operating system it's running on.

Web services and applications communicate with each other using messages in a specific, standardized format. An application builds a message request such

as "give me the weather forecast for Atlantic, NC," and sends it to the server where the Web service resides. That server calls the Web service to process the request. When the Web service returns the result (such as "warm and sunny"), the server builds a message response and sends it back to the application. At first glance, this probably sounds very complicated—building messages and sending them where they need to go. As you'll see in this book, however, Rational Developer and WebSphere handle most of the work for you, so you can just concentrate on developing your Web service functions and application logic.

Introducing Web service terminology

Before jumping into building Web services and Web applications, you need to understand some of the terminology you'll see when you use Rational Developer and WebSphere. Like most modern technologies, Web services are an alphabet soup of technical jargon, and the acronyms can make your head spin if you're not careful. At this point, we'll look at two of the main building blocks for Web services: SOAP and WSDL.

SOAP

SOAP is the protocol that defines the format of messages used to communicate with Web services. SOAP messages are XML-formatted text strings that consist of an *envelope* with a message *body* and an optional *header*. The header is used to pass control information, and the body contains the actual request or response message content. SOAP messages typically travel over an HTTP transport, the standard network protocol for Web applications, which is what you'll use in this book. You'll rarely need to look at the actual content of SOAP messages, since Rational Developer and WebSphere handle all this for you. When you learn about more advanced features later

An acronym that isn't

SOAP is an "acronym" that doesn't actually stand for anything. It originally meant "Simple Object Access Protocol," but since the protocol works with text messages rather than objects, that meaning fell by the wayside. Some people are starting to refer to SOAP as "Service-Oriented Architecture Protocol," but that's not the official definition, at least not as of this writing.

in this book, you'll look at SOAP messages in more detail and consider some of the options you can set to control how SOAP messages are built.

WSDL

Web Services Description Language (WSDL) is the XML vocabulary that describes Web services. WSDL is stored in standard text files with a *.wsdl* extension, typically on the same application server where the Web service itself is deployed.

As a Web service developer, you use WSDL to describe the functions that your Web service provides and how other programs can access those functions. For other programmers to use your Web service, you have to give them access to two things:

- Your Web service program
- The WSDL file that describes your Web service

From the WSDL, other programmers know the URL to use to invoke your Web service, the specific format of requests that your Web service handles, parameters that need to be supplied to each request, the format of responses, and so on. Basically, the WSDL provides all the rules that a program needs to follow to use your Web service.

As a Web application developer, you use the WSDL for a particular Web service to create the code that locates, builds messages for, and invokes the Web service. Typically, all this logic is placed in a *client proxy*, which represents the Web service in your client application. You just call the proxy, and it handles all the details of finding and invoking the Web service.

Luckily, you don't have to be fluent in WSDL to create or work with Web services, because Rational Developer handles it all for you. When you use the Web Service wizard to create a new Web service, the wizard creates the corresponding WSDL file. And when you want to use a Web service in a Web application, you just point the Web Service wizard to the WSDL file, and the wizard creates the client proxy to locate and invoke the service.

What you'll build

Now that you've got the basics, let's get started with a real example. You'll play two roles in this chapter. First, you'll be a Web service developer, using Rational Developer to create a Web service that returns the last trading price for a particular stock symbol. Then, you'll be a Web application developer, using your Web service to build a Web application that shows the last trading prices for a list of stocks. Figure 1.1 shows an example of the application you'll build.

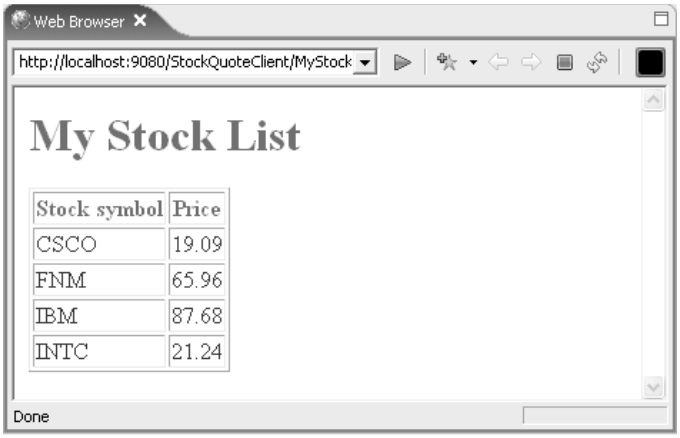


Figure 1.1: The MyStocks application in action.

Installing WebSphere Application Server and Rational Developer

To follow along with the steps in this book, you need to install IBM WebSphere Application Server 6.0. You also need to install either IBM Rational Web Developer 6.0 or IBM Rational Application Developer 6.0. Application Developer contains all the same functions you'll find in Web Developer (and much more), so you can follow the instructions with either product. For simplicity, we refer to the tools as Rational Developer in this book, but you can use the one you prefer. The screenshots in this book were taken from Web Developer, however, so if you're using Application Developer, you might see more features than in the screenshots.

You can install WebSphere Application Server and Rational Developer separately on your machine, or if you prefer, you can install WebSphere Express 6.0, which includes both WebSphere Application Server and Rational Web Developer. If you don't already have a copy of WebSphere Express 6.0, you can download a trial version at *www-106.ibm.com/developerworks/websphere/downloads/EXPRESSsupport.html*. See the appendix for installation instructions. After installing Rational Developer, make sure you use the Rational Product Updater to install the latest product updates. You'll need Rational Developer 6.0.0.1 or later for the samples in this book.

> **Note:** Make sure you install both WebSphere Application Server and Rational Developer on your machine. Rational Developer includes a WebSphere Test Environment that you'll use to test Web services and Web applications, but you'll also need a stand-alone WebSphere Application Server to set up your "production" server in the next chapter and to deploy Web services to it.

Setting up a Web project

Before you can create your first Web service, you need to do some set-up work in Rational Developer to enable the Workbench capabilities for Web services development, and to specify where your Web service will be saved.

Rational Developer is organized into a variety of *perspectives*, which are just different ways of looking at a development project based on your role on the development team. In this book, you'll mainly use the Web perspective, and within the Web perspective, your Web services and applications will be organized into projects and folders, according to their content, such as package folders for Java classes. You can switch perspectives whenever you need to. Within a particular perspective, you'll also see a variety of different editors and views that are applicable to the work done within that perspective. Don't worry—we'll explain how each of these works as we go along.

Starting Rational Developer

1. To start Rational Developer, click **Start → Programs → IBM Rational → Rational Software Development Platform**. When you start Rational Developer the first time, a window appears asking which directory you want to use for your workspace (the place where Rational Developer saves your work), as shown in Figure 1.2.

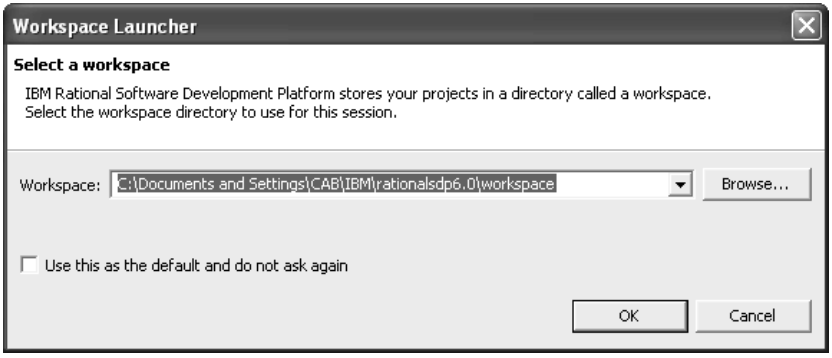


Figure 1.2: The Workspace Launcher window.

2. Leave the workspace name set to the default, and if you prefer to not be asked about it each time you launch Rational Developer in future, check the **Use this as the default and do not ask again** check box.

3. Click **OK**. After a few seconds, the Rational Developer Welcome page appears. On the Welcome page, you can click the various icons to see an overview of the Rational Software Development Platform, what's new in the current version, or to go directly to tutorials and samples to learn how to use the tool. We'll skip the Welcome information for now, but you can always come back to it by selecting **Help → Welcome** in the Workbench menu.

4. Click the **X** on the Welcome pane tab to close it. The Workbench appears, as shown in Figure 1.3. The Workbench initially opens in the Web perspective, which is the perspective you want for working with Web services and Web applications.

Application Developer users


Open a perspective

If the Workbench opens in a perspective other than Web, click the **Open a perspective** icon in the toolbar tab along the top right of the Workbench, and select **Web** from the list of perspectives. If you don't see Web in the list, click **Other…**, select **Web** in the Select Perspective window, and click **OK**. If Web still isn't present in the Select Perspective window, check the **Show all** check box.

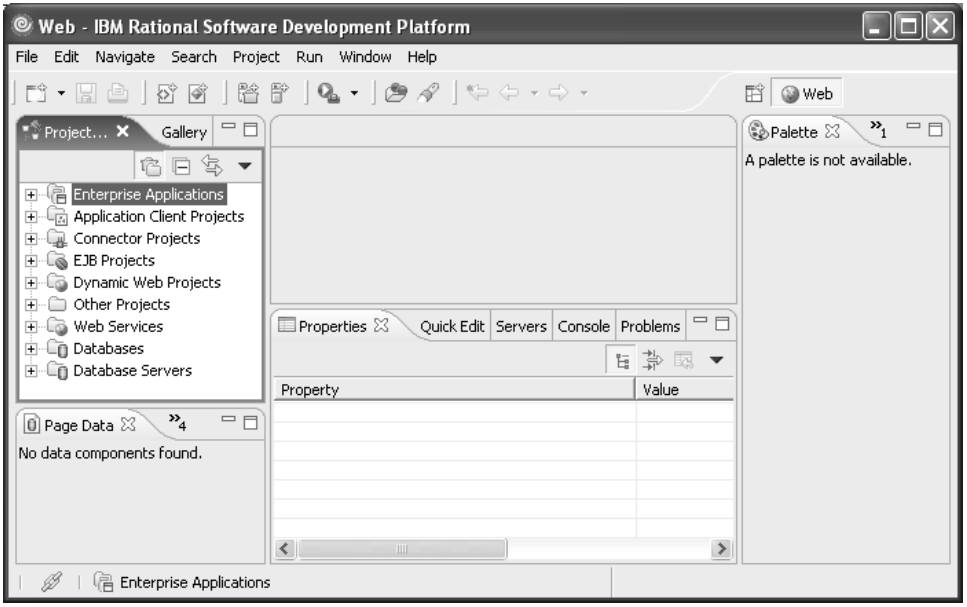


Figure 1.3: The Rational Developer Workbench.

Notice that the Workbench is divided into several toolbars and work areas. Let's take a minute to get familiar with the layout before we proceed.

Web perspective

First, the toolbar that appears along the top right tab shows an icon for each perspective that's currently open, such as the Web perspective that currently appears in the Workbench. You can have multiple perspectives open at any given time; to switch from one to another, just select its icon. You can also customize this portion of the toolbar, such as moving it to the left edge of the window or not showing text descriptions with each icon. To customize, right-click anywhere in the toolbar tab, and select the option you want from the pop-up menu.

Next, along the top of the Workbench are the pull-down menus and a toolbar for common actions for working within a particular perspective. The menus and toolbar are tailored to the functions available in a given perspective and the editors that are being used, so you'll see these items change as you switch from one perspective to another and as you work with different editors. The toolbar contains a subset of the actions available from the pull-down menus, and you can customize it to add, reorganize, or remove toolbar actions to suit your preferences.

Finally, the work areas within the Workbench are also tailored to the functions available in a given perspective and the editors and views being used. In the Web perspective, you use the Project Explorer to navigate the various folders and files in your Web projects. When you open a particular file, the editor for that file type appears in the center of the Workbench. For example, the Java Editor appears when you are working with Java source files, and the Page Designer appears when you are working with Web pages. Any associated views are shown in the other work areas.

Enabling Workbench capabilities

The Workbench comes with the capabilities enabled that a Web developer would be most likely to use, such as working with Java programs and designing Web pages. This simplifies the default options you see in wizards and pop-up menus, so your Workbench isn't cluttered with a lot of options that you don't typically use. When you need to use more advanced functions (for example, when you work with Web services or databases), additional capabilities are enabled automatically. You can also manually enable capabilities by setting your Workbench preferences, as follows:

1. Select **Window → Preferences** in the Workbench menu.
2. In the Preferences window that appears, the list in the left pane shows the default set of categories for which you can set preferences. Click the plus sign next to Workbench to expand the list of Workbench preferences, and select **Capabilities**.

3. In the right pane, you see all the possible Workbench capabilities, with the basic set for Web developers already checked. When you select a particular capability in the right pane, its description appears below the list. Select **Web Service Developer** to see its description, and then click the plus sign next to Web Service Developer to see the capabilities contained in this role. To enable Web services capabilities, check the **Web Service Developer** check box, so the window looks like Figure 1.4. Checking this check box enables both Core Database Development and Web Services Development, so you're already set for the work you'll do with databases in chapter 5.

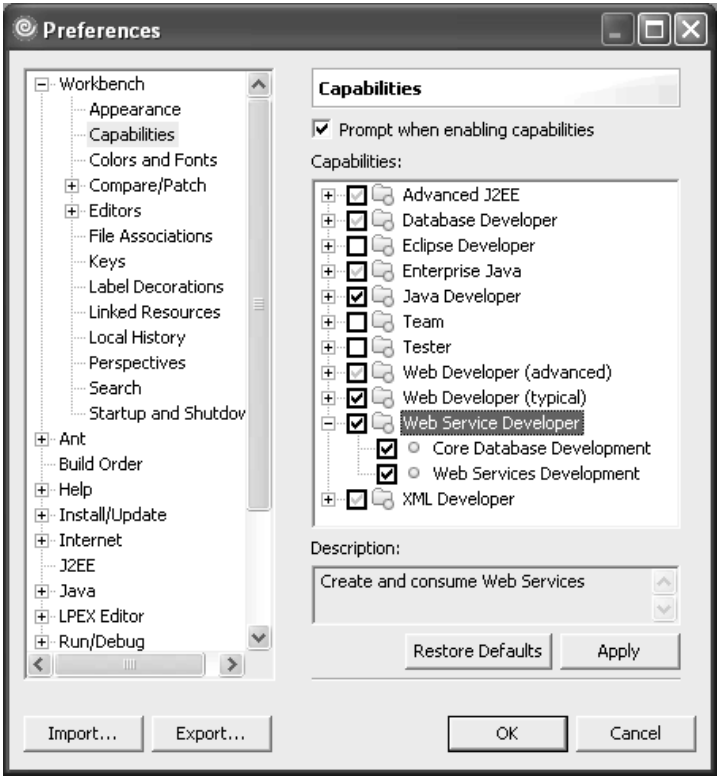


Figure 1.4: Setting Workbench capabilities.

4. As you can see in Figure 1.4, capabilities are grouped by role, such as Web Developer (advanced) and Web Service Developer. Capabilities

that apply to multiple roles, such as Web Services Development, are duplicated under the various roles. When you check the Web Services Development check box for one role, you enable the capability for all roles. Click **OK**.

> **Tip:** Once you enable the Web services capability in the Workbench, you can then set your preferences for working with Web services. Select **Window → Preferences** again, and this time you'll see Web services as one of the categories for which you can set preferences. We'll use the default preferences in this book, but you might find that you'll want to customize some of the preferences when you develop your own Web services.

Creating the project

Next, you'll use the Web perspective to create a new Web project that will contain your Web service. You'll create the project as a *Dynamic Web project*, one that contains dynamic content, such as the Java code for your Web service. You'll also specify an EAR project (called an *Enterprise Application*), which is used for deploying and testing the Web project on an application server.

1. To create the project, right-click **Dynamic Web Projects** in the Project Explorer, and select **New → Dynamic Web Project** from the pop-up menu.
2. Enter a project name of **StockQuote**. Notice that the project location is set to the default for your workspace (the directory specified when you first started Rational Developer).
3. For your first Web project, you'll take a closer look at some of the options that will be used to create it. Click **Show Advanced**, so the window looks like Figure 1.5. For the Web projects you create in this book, you'll take the defaults. That means your Web services and applications will use WebSphere Application Server v6.0 as their target server, your Web projects will be packaged in their own separate EAR projects, and the context root (that is, the value used to generate URLs for Web components within the project) will use the same name as your Web project.

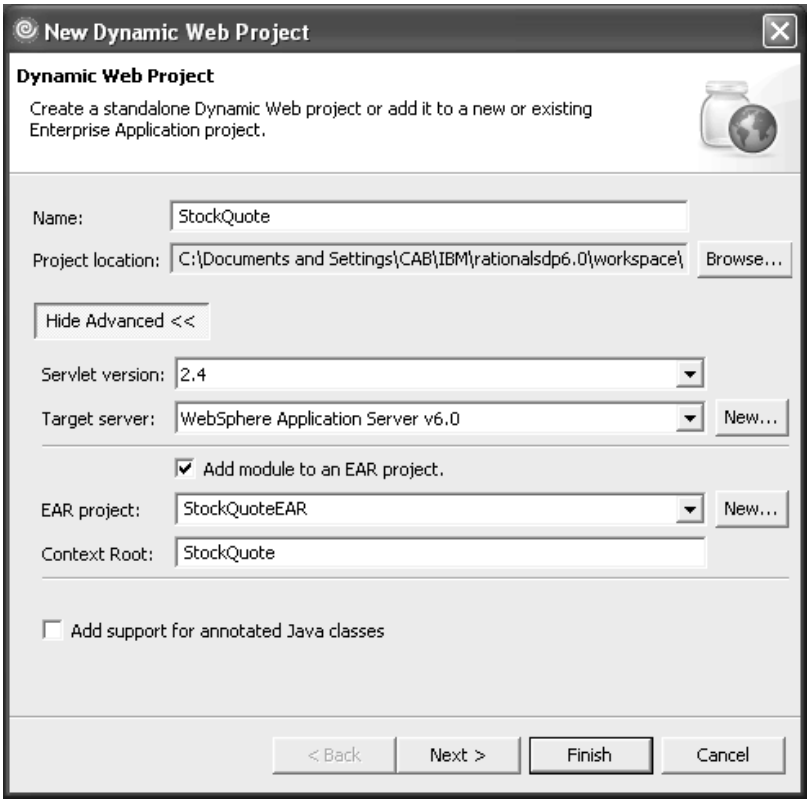


Figure 1.5: Creating a Dynamic Web project.

4. Click **Finish**.

In the Project Explorer, click the plus signs to expand the content of the folders for Dynamic Web projects and Enterprise Applications. Your StockQuote project appears as an entry under Dynamic Web Projects, and its associated StockQuoteEAR project appears as an entry under Enterprise Applications, as shown in Figure 1.6.

Rational Developer automatically created a default folder structure for your projects and placed several control files in the folders. The project structure is set up according to the specifications for packaging J2EE applications. A Web project, where you place the actual content for your Web services and applications, uses the standard structure for Web Archive (WAR) files, also called a *Web module*. An Enterprise project uses the standard structure for Enterprise Archive (EAR) files. When you create a Dynamic Web project, Rational Developer associates the Web project with an Enterprise project, and places the Web project's WAR file in the Enterprise project's EAR file, which is then deployed to an application server.

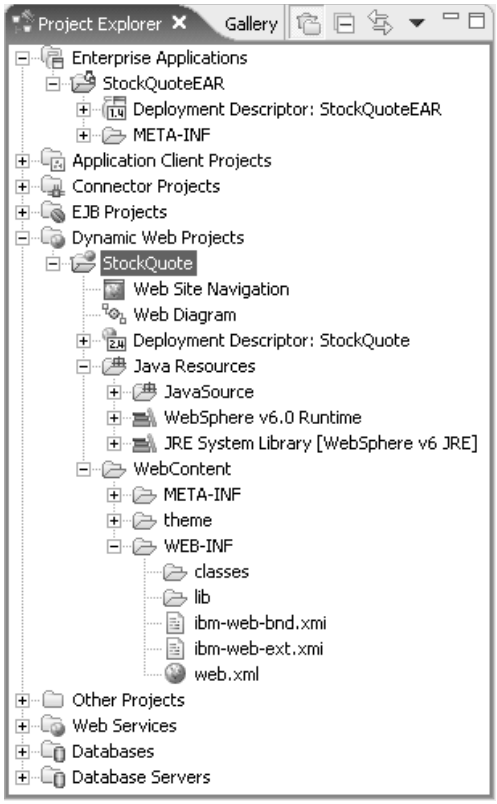


Figure 1.6: The Project Explorer.

Multiple Web projects can be associated with a particular Enterprise project. You do that to combine related Web modules into a single, larger J2EE application. Luckily, you don't have to remember all the details for WAR and EAR files, because Rational Developer does the packaging for you. You just place your Java source files in Java packages within the JavaSource folder in the Java Resources folder, and you place the content for your Web pages in the WebContent folder. Rational Developer automatically updates the necessary configuration and deployment information for you.

Creating a Web service

For your first Web service, you'll use Rational Developer to turn a Java class into a Web service. The Java class you'll use is our version of the StockQuoteService sample that's supplied with Rational Developer. Our code is in the solutions folder on the CD included with this book. You'll create a new Java package in your Web project, import the Java source file into the Workbench, and then use the Web Service wizard to turn the Java class into a Web service.

Importing the Java source

To create a new Java package in your Web project and import the Java source file, follow these steps:

1. In the Project Explorer, right-click the **JavaSource** folder in the Java Resources folder for the StockQuote project, and select **New → Package** from the pop-up menu.
2. Enter **samples** for the package name, as shown in Figure 1.7.

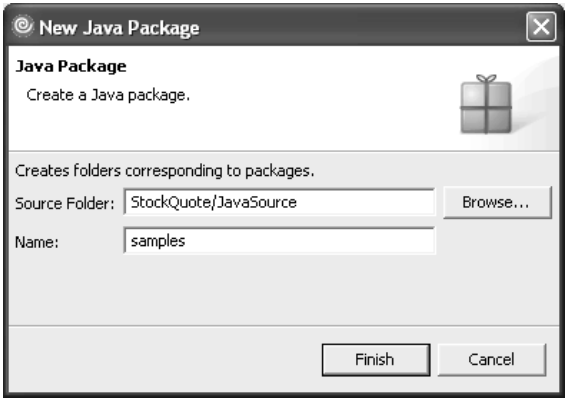


Figure 1.7: Entering the package name.

3. Click **Finish**.
4. In the Project Explorer, right-click the **samples** package that you just created, and select **Import…** from the pop-up menu.

5. Choose **File system** as the import source, and click **Next**.
6. Click the **Browse…** button next to the From Directory field, and browse to the /solutions/chap01/samples folder on the CD included with this book.
7. Click **OK**.
8. Select **StockQuote.java** as the file to import, and make sure that **Create selected folders only** is selected, as shown in Figure 1.8.

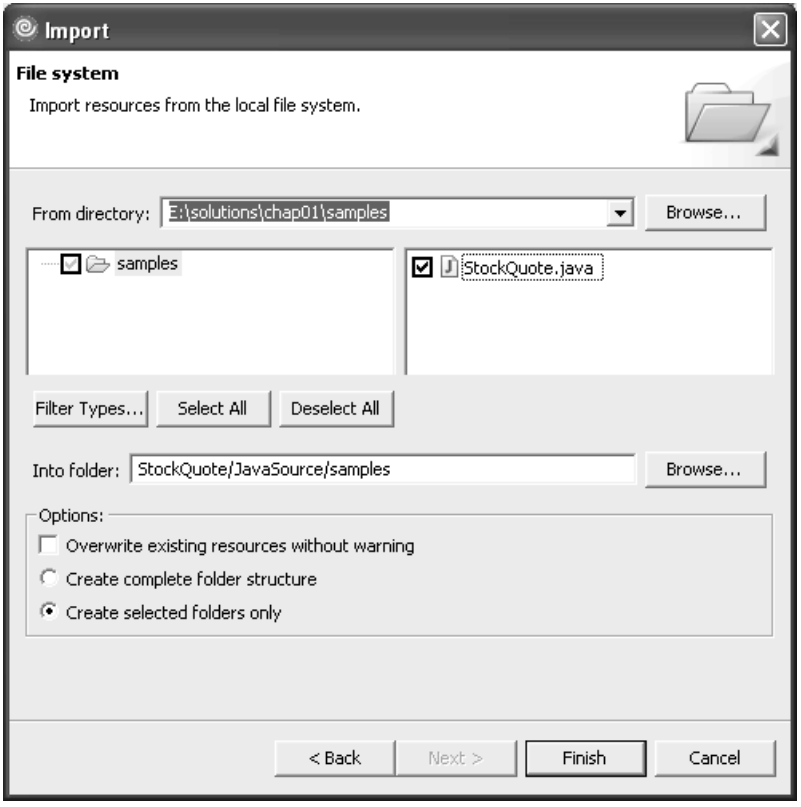


Figure 1.8: Importing the Java source code for StockQuote.java

9. Click **Finish**.

We'll take a quick look at the code you just imported before proceeding. In the Project Explorer, expand the content of the samples package, right-click **StockQuote.java**, and select **Open** from the pop-up menu (or just double-click **StockQuote.java**). The Java Editor opens with the source code shown in Listing 1.1.

```
package samples;

import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;
import java.io.StreamTokenizer;
import java.net.URL;

/**
 * Java class that gets a stock price quote given a symbol.
 */
public class StockQuote
{

 /**
  * Gets the price for a stock.
  *
  * @param the String stock symbol, e.g. "IBM"
  * @return the stock price
  */

 public float getQuote(String symbol) throws Exception {

  URL url = new URL(BASE_URL + symbol);

  // get the quote as a comma separated value string, as in this example:
  // "IBM",80.85,"11/6/2002","2:20pm",-0.68,80.80,81.500,80.10,5697700

  InputStream is = url.openStream();
  Reader reader = new InputStreamReader(is);
  StreamTokenizer st = new StreamTokenizer(reader);

  // get the symbol string token, e.g. IBM
  st.nextToken();
  String outSymbol = st.sval;
  if (!symbol.equalsIgnoreCase(outSymbol)) {
    throw new Exception("Wrong symbol received: " + outSymbol);
  }
```

Listing 1.1: The source code for the StockQuote sample, part 1 of 2.

```
  // skip the comma token and get the price number token, e.g. 80.85
  st.nextToken();
  st.nextToken();
  float price = (float) st.nval;

  reader.close();
  return price;
 }

 private String BASE_URL =
     "http://finance.yahoo.com/d/quotes.csv?f=sl1d1t1c1ohgv&e=.csv&s=";
}
```

Listing 1.1: The source code for the StockQuote sample, part 2 of 2.

As you can see, the StockQuote class is fairly simple. The caller passes a stock symbol to the getQuote(String) method. This method uses the stock symbol to build a URL to get the price information from the *finance.yahoo.com* Web site. The method then extracts the last trading price from the string returned by the Web site, and returns that value to the caller. When you're finished looking at the source code, close the Java Editor.

> **Tip:** The StockQuote sample uses the comma-separated value (CSV) interface provided by the *finance.yahoo.com* Web site. The format of the string returned by the Web site is subject to change, so if you have problems running the code, check *help.yahoo.com/help/us/fin/quote/quote-05.html* to see if the interface has changed, and update the sample accordingly.

Using the Web Service wizard

Next, you'll use the Web Service wizard to turn the StockQuote class into a Web service. As part of the wizard's processing, it deploys the Web service to your WebSphere Test Environment, so you'll start the server before running the wizard. If you forget to start the server before running the wizard, that's OK, because the wizard will start the server for you—we just like to do things in an orderly process to keep track of when the server's started, so that we remember to stop it when we're done.

1. Click the **Servers** tab in the Workbench, and select **WebSphere Application Server v6.0**. This server is the WebSphere Test Environment that you'll use to run your Web services and applications.
2. Right-click **WebSphere Application Server v6.0**, and select **Start** from the pop-up menu. After a few moments, you'll see the server's status change to "Starting" and then "Started."

> **Tip:** To see the server's message log, click the **Console** tab. This shows all the detailed start-up messages, as well as runtime messages when your Web service and applications are deployed to the server and run. The log file itself is called SystemOut.log, and you'll find it in the runtimes/base_v6/profiles/default/logs/server1 folder under the folder where you installed Rational Developer (for example, C:/Rational60).

3. To create your Web service, right-click **StockQuote.java** in the Project Explorer, and select **Web Services → Create Web service** from the pop-up menu.
4. The Web service's Options window appears. Make sure the Web service type is set to **Java bean Web Service**, the **Start Web service in Web project** check box is checked, and the **Create folders when necessary** check box is checked, as shown in Figure 1.9. (These should all be the default settings, unless you've changed your preferences.)

> **Note:** The Web Service wizard has many options to control how a Web service is created, to test and publish the Web service, and to generate and test the client proxy that applications call when they want to use the service. You'll see many of the other options later in this book. For now, you're just creating the Web service using Rational Developer's defaults. That means your Web service will be set up to run in the WebSphere Application Server v6.0 runtime environment, which is the default runtime environment for Rational Developer.

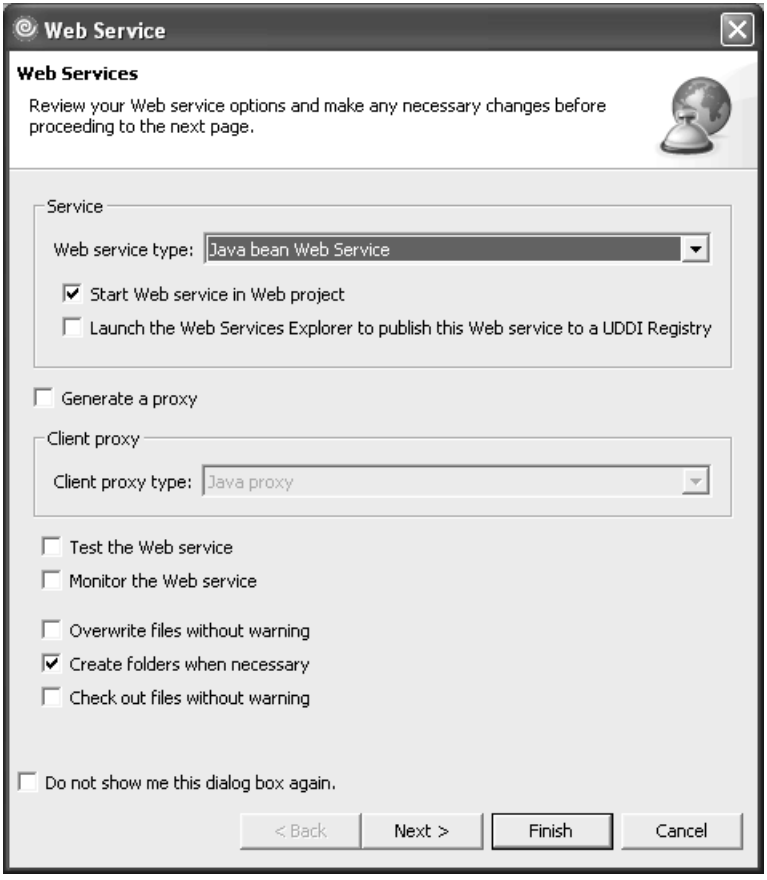


Figure 1.9: The Web service's Options window.

5. Click **Finish**.

After a few seconds the wizard completes. Let's take a closer look at what it just did for you. First, notice that several new files and folders have been added to the StockQuote project, as shown in Figure 1.10.

What you see are the control files that WebSphere Application Server needs to process your StockQuote class as a Web service, so that, when an application sends a message to invoke the Web service, the server knows what to do. The control files include the information that applications need to discover and use the Web service—namely, the StockQuote.wsdl file in the WebContent/WEB-INF/wsdl folder. The WSDL file describes the operations that the Web service provides and the URL that applications use to access the Web service. You'll use this file in the next section to create a client proxy (that is, the code that your application calls in order to use the Web service).

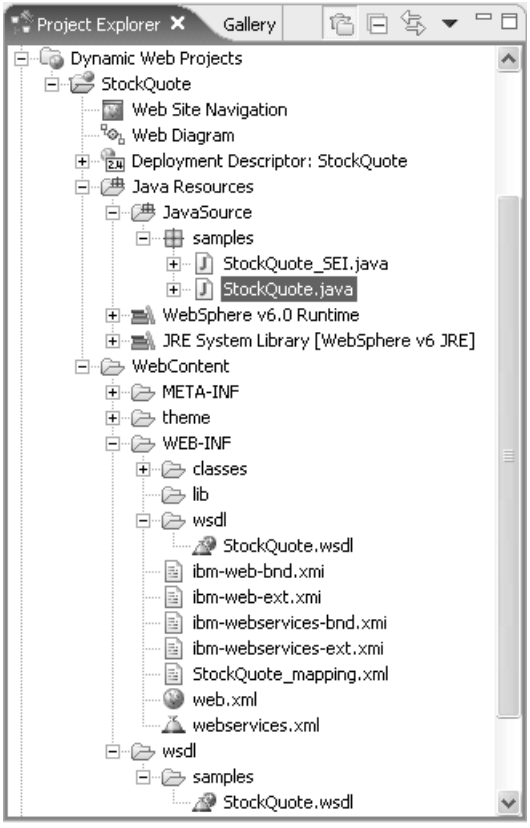


Figure 1.10: Project Explorer after running the Web Service wizard.

Note that there's also a second copy of the WSDL file in the WebContent/wsdl/samples folder. This copy is used by tools such as the Web Services Explorer and Web Service Discovery Dialog, both of which you'll use later in this book. (We'll examine the WSDL file in more detail in the next chapter. However, if you want to take a peek now, feel free. Just right-click the file name, and select **Open With → WSDL Editor** from the pop-up menu. You'll see all the control information that describes your StockQuote Web service and its getQuote operation.)

In addition to creating all the necessary control files, the wizard also deployed your Web service to the WebSphere Test Environment. To see that, either click the Console tab or browse the messages in the server's SystemOut.log file. That file is in the runtimes/base_v6/profiles/default/logs/server1 folder under the folder where you installed Rational Developer (for example, C:/Rational60).

Your Web service is now ready to use. In the next section, you'll change roles from being a Web service developer, to being a Web application developer. In that role, you'll create a Web application to test your Web service.

Creating a Web application

The Web application you'll create to test your Web service is a JavaServer Page (JSP). It contains an HTML table with your favorite stock symbols and their last trading prices. You'll create the application in a new Web project using Rational Developer's Page Designer and the Web Service wizard. Then, you'll run the JSP in the WebSphere Test Environment (the same server where your Web service is currently deployed).

What's a JSP?

A *JavaServer Page* (JSP) is a dynamic Web page that contains a combination of static HTML elements and Java code. A JSP file is saved as a standard text file with a *.jsp* extension. A JSP is compiled and run on an application server when a user requests the page from a Web browser, and the resulting output is returned to the user.

Some of the elements you'll commonly see in JSP files are beans, scriptlets, and expressions. *Beans* are instances of Java classes that you reference within the JSP. *Scriptlets* are Java code executed at the point they appear in the JSP file. *Expressions* are Java functions that return string values, which are inserted into the resulting Web page at the point they appear in the JSP file.

Setting up the JSP

1. Create a new Dynamic Web project named *StockQuoteClient* by right-clicking **Dynamic Web Projects** in the Project Explorer and selecting **New → Dynamic Web Project** from the pop-up menu. Enter **StockQuoteClient** as the Web project name, accept the defaults in the advanced options, and click **Finish**. Note that you could have specified the existing StockQuoteEAR project as the EAR project name for your new Web project—that would have placed both the Web service and the Web application in the same EAR file. You're not doing that, because in the next chapter, you'll deploy just the Web service (and not the Web application) to your "production" application server.

2. In the Project Explorer, right-click the **WebContent** folder for the StockQuoteClient project, and select **New → JSP File** from the pop-up menu.


Create a JSP File

> **Tip:** Make sure you select the WebContent folder in the StockQuote Client project, and not in the StockQuote project. Also, note that instead of using the pop-up menu, you can create a new JSP file by selecting the folder and clicking the **Create a JSP File** icon in the Web perspective's toolbar. Rational Developer gives you several ways to accomplish common tasks, so you can choose the method you prefer.

3. Enter **MyStocks.jsp** for the JSP file name, so the window looks like Figure 1.11.

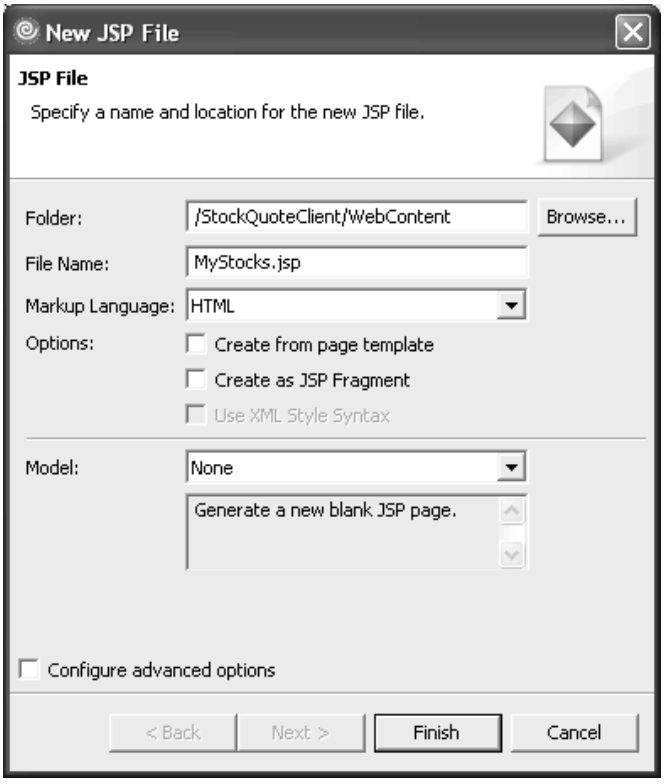


Figure 1.11: Creating a JSP file.

4. Click **Finish**.

The MyStocks.jsp file opens in the Page Designer, as shown in Figure 1.12. (If you see HTML source code instead of the design surface, click the **Design** tab.)

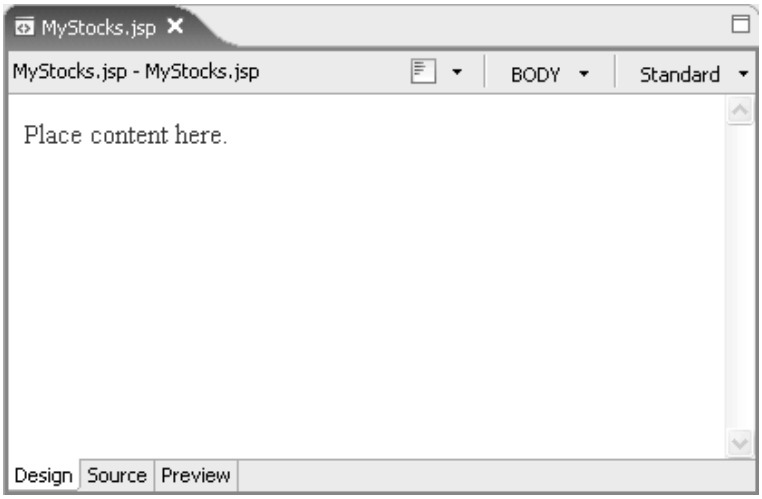


Figure 1.12: A new JSP file in the Page Designer.

Tip: To see the HTML and JSP source for your Web page, click the **Source** tab in the Page Designer. To see a preview of what the HTML elements in your Web page will look like in a Web browser, click the **Preview** tab. The Preview tab won't show you what the final JSP elements will look like, however, because it's not actually running in a real application server.

Using the Page Designer

You use the Page Designer much like any other graphical editor. To add or change text in the Web page, just type on the design surface. To change the attributes for a particular text string, select the text on the design surface, and then set the value you want in the Properties view. The Properties view appears at the bottom of the Workbench, and its layout changes based on the element that's currently selected on the design surface.

To add more complex elements to the Web page, such as tables, images, forms, or JSP logic, select the element you want in the Palette (Figure 1.13), drop it on

the design surface, and set its attributes in the Properties view. The Palette contains all the commonly used elements for Web pages, organized into related groups called *drawers*. You click the name of a drawer to open it.

> **Tip:** The Properties view should already appear in the Workbench (click the **Properties** tab to see it), and the Palette should automatically appear in the Workbench when you open a file with the Page Designer. To see a view that isn't already open, use **Window → Show View**.

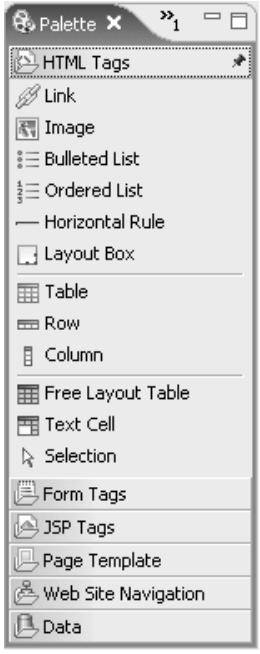
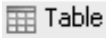


Figure 1.13: The HTML drawer in the Palette.

Now, you'll use the Page Designer to create the static content for your Web application:

1. Select **Place content here**. on the design surface, and change the text to **My Stock List**.
2. In the Properties view, select **Heading 1** as the Paragraph value for the text.
3. Click **HTML Tags** in the Palette to open the drawer for HTML elements.



Table

4. Select a **Table** in the Palette's HTML drawer, and drop it on the design surface below the heading. When prompted to enter the number of table rows and columns, leave the columns set to **2**, and set the rows to the number of stocks you want to display plus one for the table header, as shown in Figure 1.14. You can set some table properties here, or if you prefer, you can set the properties after the table has been dropped on the design surface, by editing the values in the Properties view.

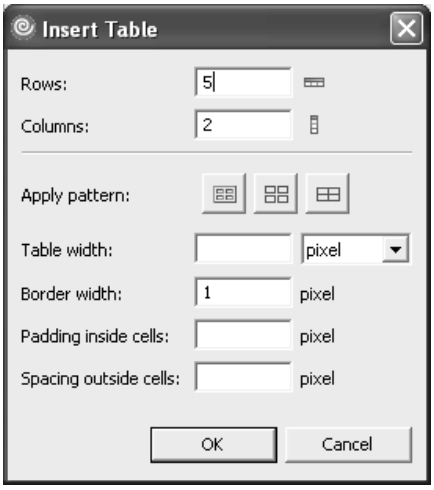


Figure 1.14: The Insert Table window.

5. Click **OK**. The design surface should look something like Figure 1.15.

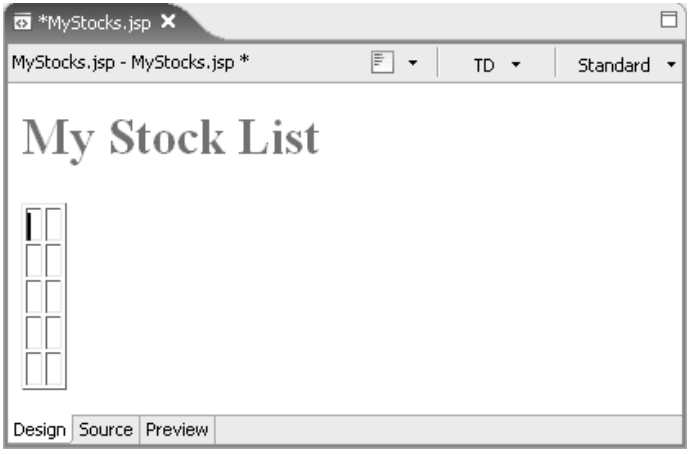


Figure 1.15: Adding an HTML table to the design surface.

6. Next, you'll set the values for the table header. In the design surface, enter **Stock symbol** for the text in the first table cell (the first row and first column). The first cell should already be selected; if not, just click in it to select it.

7. In the Properties view, select **Header** as the cell type.
8. Select the next cell in the design surface (the first row and second column). Enter **Price** for the text in this cell, and in the Properties view, select **Header** as the cell type.
9. Enter text values for the stock symbols you want to include in the table, so the design surface looks something like Figure 1.16.

Figure 1.16: Setting the static text for the HTML table.

The static content for your Web application is now complete.

Creating a client proxy for the Web service

In this section, you'll use the Web Service wizard to create a client proxy for your StockQuote Web service, that is, the code that a Web application uses to call the Web service. To create the client proxy for your Web service, you could browse to the WSDL file in your StockQuote project and launch the Web Service wizard from the file in that project, but we'll show you a more convenient way to access the Web services in your Workbench.

1. Scroll down in the Project Explorer until you see the Web Services folder, and expand the content of the folder. Notice that the folder

contains sub-folders for Services and Clients. Within the Services folder are all the Web services in your Workbench (currently just your StockQuote Web service), and within the Clients folder are all the Web service client proxies in your Workbench (none currently, but that will change in a moment).

2. Right-click **StockQuoteService** in the Services folder, and select **Generate Client** from the pop-up menu. The Web Service Client Options window appears, as shown in Figure 1.17.

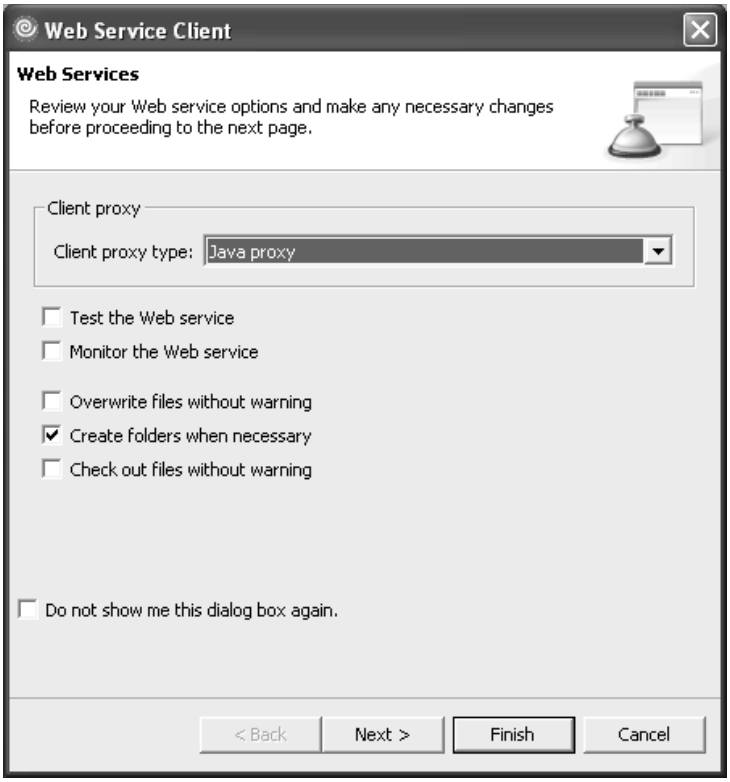


Figure 1.17: The Web Service Client Options window.

3. Make sure the client proxy type is set to **Java proxy** and the **Create folders when necessary** check box is checked, and click **Next**. The Web

Service Selection page appears for you to select the WSDL file from which you want to create a client proxy, as shown in Figure 1.18.

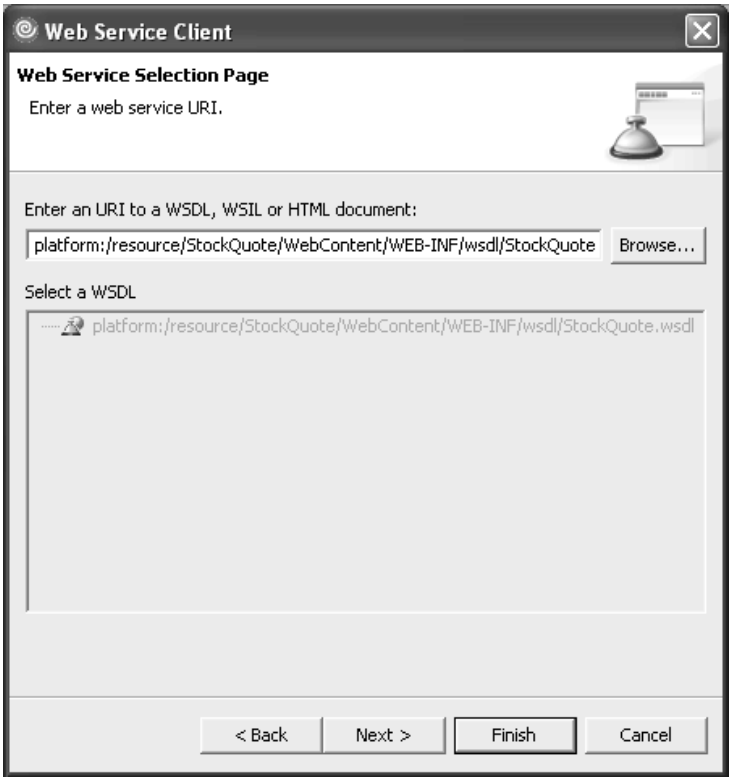


Figure 1.18: The Web Service Selection page.

4. The URI is already set to the WSDL file for your StockQuote Web service, so just click **Next**. The Client Environment Configuration page appears, to let you select the runtime environment for the client (that is, the runtime environment where your JSP runs). Make sure the Web service runtime is set to **IBM WebSphere**, the server is set to **WebSphere Application Server v6.0**, and the J2EE version is set to **1.4**. Also, make sure **StockQuoteClient** is selected as the Client project (the project where the wizard will place the client proxy), so the window looks like Figure 1.19.

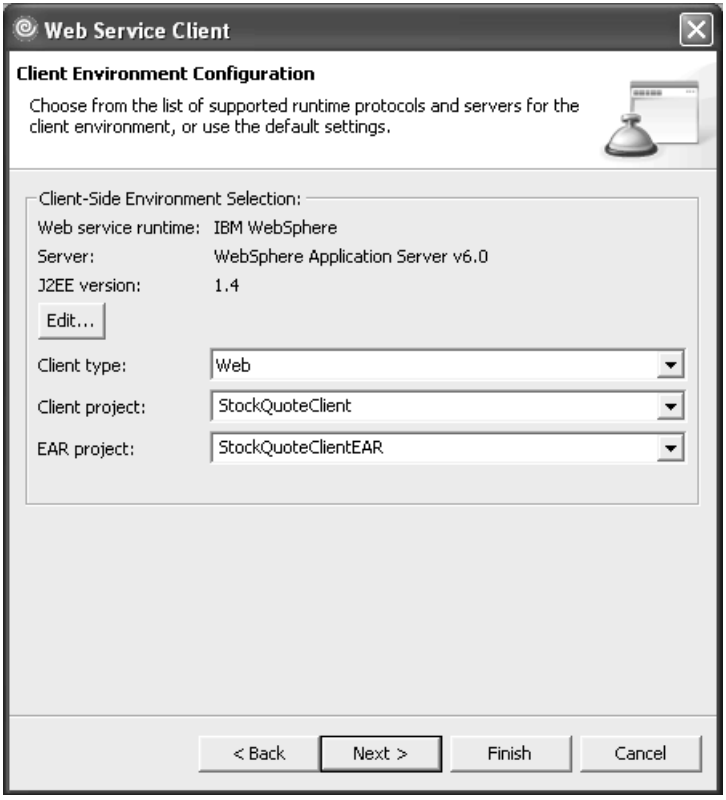


Figure 1.19: The Client Environment Configuration page.

5. Click **Finish**.

6. When prompted to enable overwriting for the web.xml file, click **Yes**. This lets the wizard add the necessary control information for packaging your Web application. That's it—your client proxy has been created.

Note: We'll skip this last step for the rest of the client proxies that you create in this book, by having you check the **Overwrite files without warning** check box on the first page of the wizard. For this first client proxy, we wanted you to see all the steps that take place.

Before proceeding, let's take a closer look at what the wizard just did for you. Expand the contents of the JavaSource folder in the StockQuoteClient project's Java Resources folder. You'll see that it now contains a "samples" package with six Java source files. These files are the client proxy code that the wizard generated for you from the WSDL file using the default client runtime, that is, the WebSphere Application Server v6.0 runtime environment. If you had chosen a different client runtime, the generated code would look somewhat different, but the concept would still be the same—these are the classes that your application uses to access the Web service.

The client proxy code is very simple to use. First, you create an instance of the proxy class that represents your Web service—that's StockQuoteProxy for your StockQuote Web service. Then, you call the proxy's getQuote(String) method, passing the stock symbol whose trading price you want. That's it. The proxy classes do all the work to locate your Web service, send the request message, and return the response (the stock price) to you.

Using the client proxy in the Web application

Let's go back to the Page Designer to add the JSP logic that uses the client proxy to get the stock prices. Click anywhere on the design surface to return the focus to the Page Designer, and then follow these steps:

1. Click **JSP Tags** in the Palette to open the drawer for JSP elements.
2. You need an instance of the StockQuoteProxy class (or bean). Select a **Bean** in the Palette, and drop it on the design surface somewhere before the table. (Don't worry if it doesn't land on the right spot—you can move it later.)

Bean

3. In the Insert JSP Bean window that appears, enter **proxy** for the ID. (This is the name you'll use to reference the bean.) Enter **samples.StockQuoteProxy** for the Class name, so the window looks like Figure 1.20.

Tip: If you don't want to type in the full class name, you can use the **Browse…** button to locate it. In the Class Selection window, enter the first part of the class name, such as **Stock**. A list of all classes in the project's build path that match that pattern are presented in a list, from which you can select the class and package name you want to use.

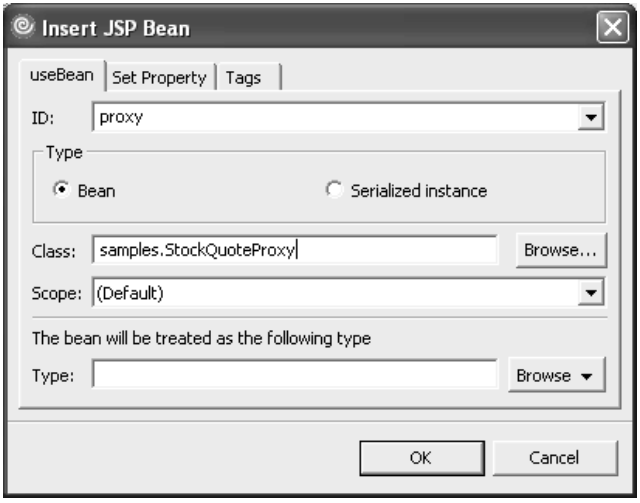


Figure 1.20: Adding the StockQuoteProxy bean to the design surface.

4. Click **OK**. The design surface should now look something like Figure 1.21.

Figure 1.21: The StockQuoteProxy bean added to the design surface.

Now that you have an instance of the StockQuoteProxy bean, you'll add Java code to call the proxy's getQuote(String) method, which gets the price for each stock symbol and inserts the resulting values into your Web page.

> **Tip:** If the bean landed somewhere else, such as within the table, use the mouse to move it. You'll be referencing this bean within the table rows to get the stock prices, and the bean instance needs to be created before it can be used.

Expression
Expression

5. Select an **Expression** in the Palette, and drop it on the design surface in the table cell for the first stock price. In the Properties view, enter **String.valueOf(proxy.getQuote("*symbol*"))** for the expression value, where *symbol* is the stock symbol, such as **String.valueOf(proxy.getQuote("CSCO"))** for the stock shown in the first row of our example. This code calls the client proxy to get the last trading price for the stock symbol specified, and then converts the float value returned by the proxy to a String value.

6. Repeat the previous step for each stock symbol in the table. Your design surface should now look something like Figure 1.22, and the Web page source should look like Listing 1.2.

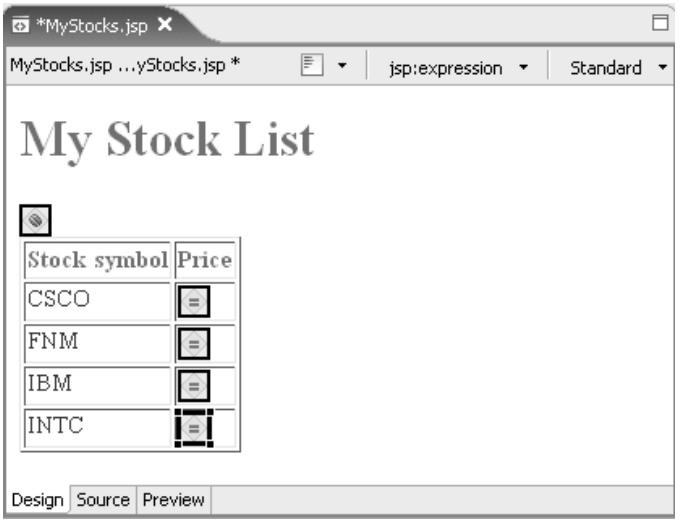


Figure 1.22: The design surface with all elements added.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<HTML>
<HEAD>
<%@ page language="java" contentType="text/html; charset=ISO-
  8859-1"%>
<META http-equiv="Content-Type" content="text/html;
      charset=ISO-8859-1">
<META name="GENERATOR" content="IBM Software Development
   Platform">
<META http-equiv="Content-Style-Type" content="text/css">
<LINK href="theme/Master.css" rel="stylesheet" type="text/css">
<TITLE>MyStocks.jsp</TITLE>
</HEAD>
<BODY>
<H1>My Stock List</H1>
<jsp:useBean id="proxy"
  class="samples.StockQuoteProxy"></jsp:useBean>
```

Listing 1.2: The final source code for the MyStocks Web application, part 1 of 2.

```
<TABLE border="1">
  <TBODY>
        <TR>
                <TH>Stock symbol</TH>
                <TH>Price</TH>
        </TR>
        <TR>
                <TD>CSCO</TD>
                <TD><%=String.valueOf(proxy.getQuote("CSCO"))%></TD>
        </TR>
        <TR>
                <TD>FNM</TD>
                <TD><%=String.valueOf(proxy.getQuote("FNM"))%></TD>
        </TR>
        <TR>
                <TD>IBM</TD>
                <TD><%=String.valueOf(proxy.getQuote("IBM"))%></TD>
        </TR>
        <TR>
                <TD>INTC</TD>
                <TD><%=String.valueOf(proxy.getQuote("INTC"))%></TD>
        </TR>
  </TBODY>
</TABLE>
</BODY>
</HTML>
```

Listing 1.2: The final source code for the MyStocks Web application, part 2 of 2.

7. To save your changes, right-click the design surface, and select **Save** from the pop-up menu. If there are any errors in your Web page, they'll be listed in the Problems view at the bottom of the Workbench. Compare your JSP file with Listing 1.2 to correct any errors.

Tip: As you make changes in the Workbench, you might notice messages at the bottom of the window about "Building Workspace" or "Publishing EAR file" functions. These are background tasks that Rational Developer is doing for you. You might sometimes see messages that your user operations are blocked by the background tasks. This is normal, because some operations need the background tasks to complete before they can proceed, such as getting the Workspace in sync with work you've done. There might also be some cases when

your operation might not be blocked, but you'll still want to wait for the background task to complete, such as waiting for an EAR file to be published before running the Web application or Web service that's contained in it.

Running the Web application

To run your Web application, first make sure you have an active Internet connection, so that your Web service can access the *finance.yahoo.com* Web site. Also, make sure your WebSphere Test Environment is still started (both your Web service and Web application run in this server). Right-click **MyStocks.jsp** in the Project Explorer, and select **Run → Run on Server…** from the pop-up menu. When prompted to select a server to launch, make sure that **WebSphere Application Server v6.0** is selected, and click **Finish**.

Tip: You can skip the Server Selection window when running Web applications in this Web project in future by checking the **Set server as project default** check box.

Your Web application starts in a Web browser window, as shown in Figure 1.23.

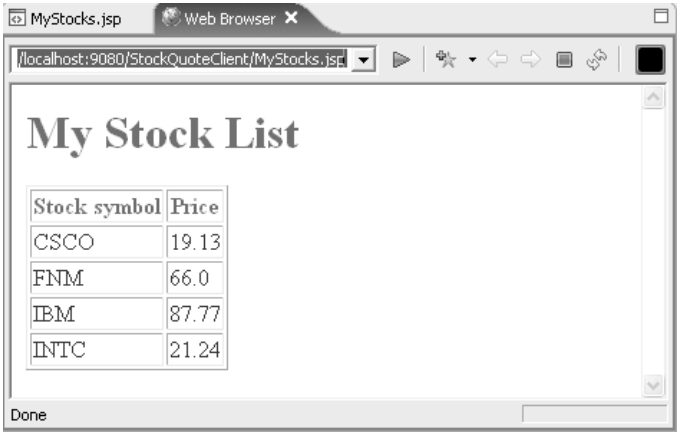


Figure 1.23: MyStocks.jsp running in a Web browser window.

Notice that Rational Developer uses the standard URL format for your Web page, which is *http://host:port/context-root/alias*. The URL prefix, *http://localhost:9080*, directs the Web browser to your WebSphere Application Server that's listening on port 9080 for HTTP requests. The context root is the value you specified when you created the Web project (StockQuoteClient, the same name as the Web project). Since you created the JSP file in the WebContent folder, the alias portion of the URL is just the name of your JSP file (MyStocks.jsp).

> **Tip:** If you're running within an intranet that uses a firewall to access the Internet, you might need to specify a proxy server in your Workbench preferences in order to access the *finance.yahoo.com* Web site. To specify a proxy server, select **Window → Preferences**, and then select **Internet – Proxy Settings** in the list of Preferences. Check the **Enable proxy** check box and enter the proxy host name and port (the same values you'd specify in a Web browser to access the Internet). If the proxy is a SOCKS server, check the **Use SOCKS** check box. If necessary, also enter authentication values for the proxy.

Importing the solution files

If you ran into any problems following the instructions in this chapter, import the solution files in the /solutions/chap01 subdirectory on the CD included with this book. There are two solution files, one for the Web service and another for the Web application. Follow these steps to import them:

1. Right-click **Enterprise Applications** in the Project Explorer, and select **Import… → EAR file** from the pop-up menu to import the solution file for the Web service.
2. Click the **Browse…** button, and browse to the /solutions/chap01 folder on the CD. Select **Chap01StockQuoteEAR.ear**, and click **Open**, so the Import window looks like Figure 1.24. (The project name will be automatically filled in.)

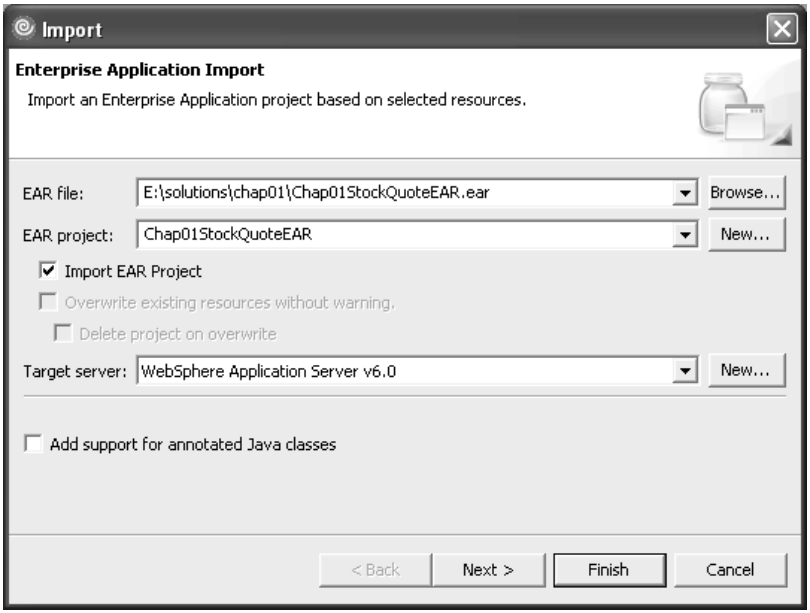


Figure 1.24: Importing the solution file for the StockQuote Web service.

3. Click **Finish**. If you are prompted to switch to the J2EE perspective, click **No**. (To avoid this prompt in future, check the **Remember my decision** check box.) Two new projects appear in the Project Explorer: Chap01StockQuote in the Dynamic Web Projects folder and Chap01StockQuoteEAR in the Enterprise Applications folder, corresponding to your own StockQuote and StockQuoteEAR projects.
4. Repeat steps 1-3 to import the Chap01StockQuoteClientEAR.ear solution file for the Web application.
5. To deploy the imported projects to your WebSphere Test Environment, click the **Servers** tab. Right-click **WebSphere Application Server v6.0**, and select **Add and remove projects…** from the pop-up menu.
6. Click **Add All** to move the new projects to the list of Configured projects, so the window looks like Figure 1.25.

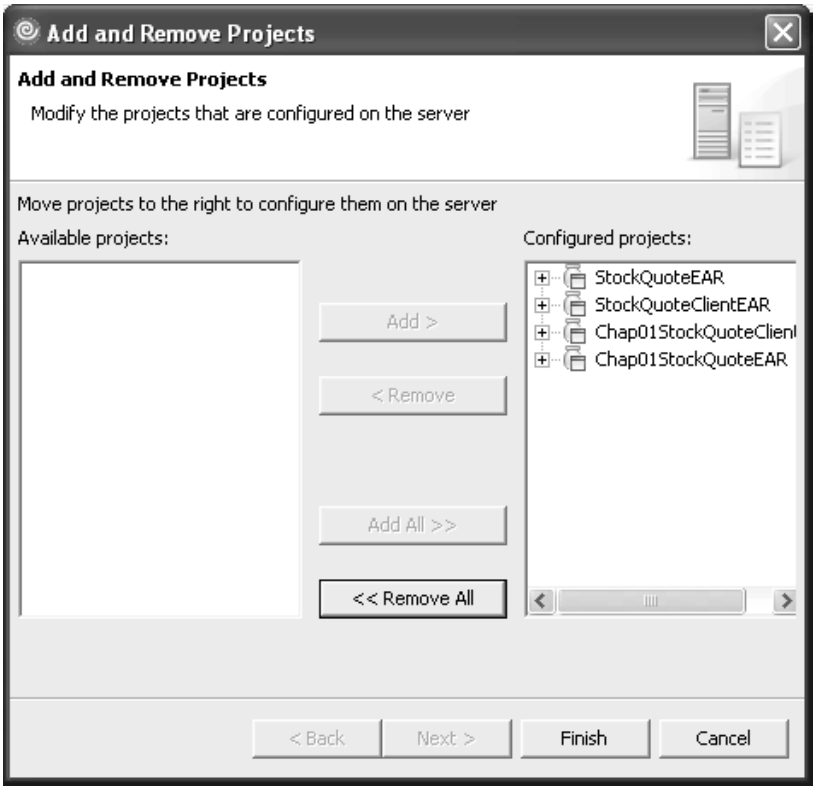


Figure 1.25: Adding the imported projects to the server configuration.

7. Click **Finish**.

> **Tip:** You can also use the Add and Remove Projects wizard to uninstall applications from the WebSphere Test Environment, by removing the associated EAR file from the list of configured projects.

8. To run the Web application, right-click **MyStocks.jsp** in the WebContent folder of the Chap01StockQuoteClient project, and select **Run → Run on Server…** from the pop-up menu. When prompted to select a server to launch, make sure that **WebSphere Application Server v6.0** is selected, and click **Finish**.

Stopping the WebSphere Test Environment

The WebSphere Test Environment doesn't automatically stop when you exit Rational Developer, so to free up resources on your computer, you should try to remember to stop the server when you no longer need it for testing. Follow these steps:

1. Click the **Servers** tab.
2. Right-click **WebSphere Application Server v6.0** in the Servers view, and select **Stop** from the pop-up menu.
3. If prompted with a message saying that the server is not responding, click **OK** to terminate the server. The server's status changes from "Started" to "Stopped."

In review

In this chapter, you learned some of the basic terminology and concepts related to Web services. You used Rational Developer to create a simple Web service and a Web application that uses the Web service, and then you ran the Web service and application with the WebSphere Application Server (Rational Developer's built-in Test Environment). In the next chapter, you'll create another WebSphere Application Server to act as your production server, deploy your Web service to the new server, and then publish it to make it available for other programmers to discover and use.

References

IBM WebSphere Express Trial Program:
www-106.ibm.com/developerworks/websphere/downloads/EXPRESSsupport.html