Pro Apache XML

Poornachandra Sarang, Ph.D.

Apress®

Pro Apache XML

Copyright © 2006 by Poornachandra Sarang, Ph.D.

ISBN-13: 978-1-59059-641-8

ISBN-10: 1-59059-641-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore
Technical Reviewer: Lalitha Sanatkumar
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editor: Sharon Wilkey
Assistant Production Director: Kari Brooks-Copony
Production Editor: Janet Vail
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: April Eddy
Indexer: Valerie Perry
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 4

Apache SOAP

Chapter 3 covered the SOAP standard theoretically, without discussing the practical implementations. This chapter introduces Apache's SOAP implementation.

Several major organizations have implemented SOAP. The Apache Software Foundation is one of them. Although nothing prevents you from hand-coding SOAP requests and manually interpreting SOAP responses, it's much easier to use the standard libraries provided by these SOAP implementers to construct the SOAP requests and electronically interpret SOAP responses.

In this chapter, you will learn how to install the Apache SOAP toolkit on Tomcat. You will then see an overview of the SOAP implementation architecture. You will create both RPC- and Document-style web services and will develop the client applications that access these services. You will examine the SOAP requests and responses created by the SOAP toolkit. You will learn how to handle exceptions in web services and how to use user-defined data types in your web services implementations. Finally, you will learn about the deployment issues with web services.

The chapter covers several programming examples. If you want to try experimenting with the code, you will need to download and install the SOAP implementation. Because you need to deploy your web service on a server, you will also have to install an HTTP server with servlet support. The Apache SOAP implementation supports several popular HTTP servers, such as Apache Tomcat, BEA WebLogic, IBM's WebSphere, Sun Microsystems' Sun ONE (previously known as iPlanet), the open source Jetty, Macromedia JRun, and Caucho Technology's Resin. Because this book is mainly about Apache implementations of XML APIs, I have used Apache Tomcat for the examples in this chapter.

Installing Apache SOAP and Related Software

Installing Apache SOAP is fairly easy. However, before installing these SOAP libraries on your machine, you need to install several other software packages. Specifically, you need three packages in order to use Apache SOAP:

- *The Java Runtime Environment (JRE) 1.5*: This is required for running any Java application on your machine.
- *Apache Tomcat*: This is the web server on which you will deploy the web services.
- *Apache SOAP*: This is the Apache implementation of the SOAP specifications.

Installing JRE 1.5

Tomcat version 5.5.9[1] was used for running the web services throughout this chapter. This version of Tomcat requires JRE 1.5. Therefore, if you do not have JRE 1.5 installed on your machine, download it from the following URL:

```
http://java.sun.com/j2se
```

The installation procedure is usually very simple. For example, for the Windows platform, an installer is provided. When you download the software, you get a Windows installer named `jdk-1_5_0-windows-i586.exe`. Double-clicking on this installer guides you through the various steps of installation. These steps are not exhaustive; they simply ask you to accept the license agreement and specify the folder where you would like to install the software.

■ **Note** For those of you installing on Linux, Appendix A provides detailed installation instructions for all the chapters in this book.

Installing Apache Tomcat

The latest version of Tomcat can be downloaded from the following URL:

```
http://jakarta.apache.org/
```

The download is available in various formats supporting different platforms. The entire source is also available for download. You can download the source and build it for your desired platform.

For Windows, you can download the zip archive. If you download the `.zip` version, simply unzip the software to your desired folder. The `.zip` version (`apache-tomcat-5.5.9.zip`[2]) does not come with administrator software for the server. This has to be downloaded (`apache-tomcat-5.5.9-admin.zip`) separately and unzipped to the same folder as Tomcat. The administrator software can be downloaded from the archives listed on the Apache site.[3]

For the Windows platform, the software is also supplied as an executable that you can download (`apache-tomcat-5.5.9.exe`). Running this executable results in it asking you to select the components you want to install, the installation folder, the port number, and the user name and password for the server. After you enter this information, the installer installs Tomcat on your machine, ready for testing. If you install by using this installer, you will still need to install the administrator service separately as explained in the previous paragraph. This is required only if you need to administer the server.

1. This is the latest stable build available at the time of this writing.
2. The filename will vary depending on the version you are downloading.
3. http://jakarta.apache.org/

Note For those of you installing on Linux, Appendix A provides detailed installation instructions for all the chapters in this book.

Testing the Installation

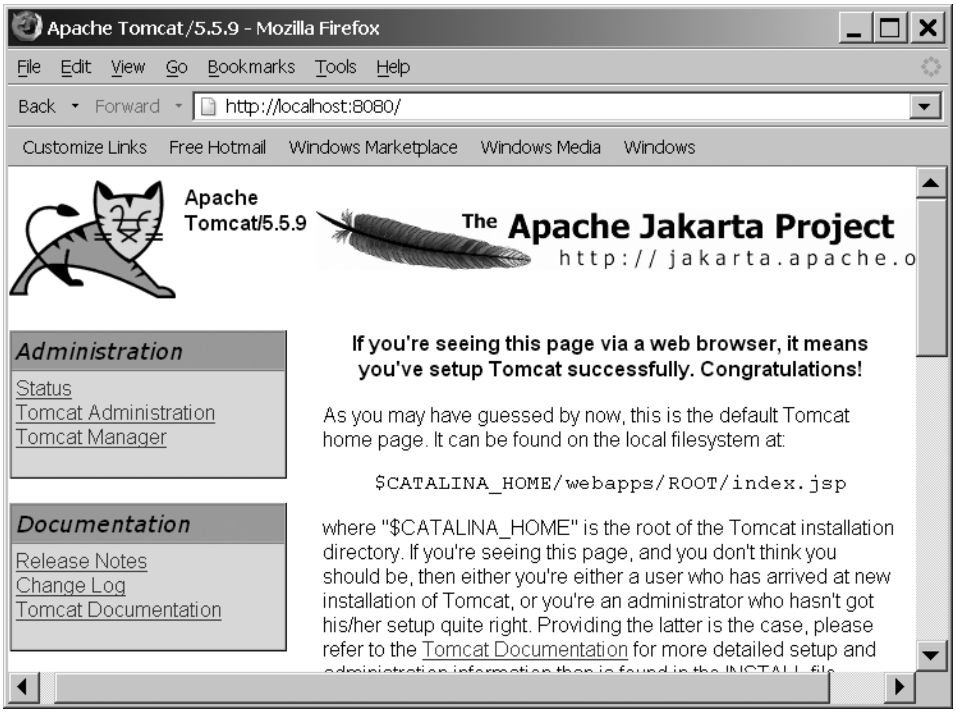
To start Tomcat, run the `startup.bat` batch file (if you are running Windows) or the `startup.sh` script (if you are running Linux).

Note If you have installed Tomcat as a Windows service, you need not run the `startup.bat` file because Windows automatically starts Tomcat at boot.

Open your browser and type in the following URL:

```
http://localhost:8080
```

This should open the Apache Tomcat/5.5.9 opening screen in your browser, as shown in Figure 4-1.


Apache Tomcat/5.5.9 - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Back Forward http://localhost:8080/
Customize Links Free Hotmail Windows Marketplace Windows Media Windows
Apache Tomcat/5.5.9
The Apache Jakarta Project
http://jakarta.apache.o
Administration
Status
Tomcat Administration
Tomcat Manager
Documentation
Release Notes
Change Log
Tomcat Documentation

If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!

As you may have guessed by now, this is the default Tomcat home page. It can be found on the local filesystem at:

```
$CATALINA_HOME/webapps/ROOT/index.jsp
```

where "$CATALINA_HOME" is the root of the Tomcat installation directory. If you're seeing this page, and you don't think you should be, then either you're either a user who has arrived at new installation of Tomcat, or you're an administrator who hasn't got his/her setup quite right. Providing the latter is the case, please refer to the Tomcat Documentation for more detailed setup and

Figure 4-1. *Tomcat server welcome screen*

To run the administrator utilities, use the links listed under the Administration panel on the left side of the opening screen. When invoked, these links will ask you to enter credentials before they grant access to web server administration. For this, you will need to create an `admin` account for your server, if you have not done so during installation. You create the administrator account by modifying the `tomcat-users.xml` file under the `config` folder in your Tomcat installation. The modified configuration file is shown in Listing 4-1. You will need to add the lines shown in bold to the existing file.

Listing 4-1. *The* `tomcat-users.xml` *Configuration File*

```
<?xml version="1.0" encoding="utf-8" ?>
  <tomcat-users>
    <role rolename="tomcat" />
    <role rolename="role1" />
    <role rolename="manager" />
    <role rolename="admin" />
    <user username="tomcat" password="tomcat" roles="tomcat" />
    <user username="both" password="tomcat" roles="tomcat,role1" />
    <user username="role1" password="tomcat" roles="role1" />
    <user username="admin" password="admin" roles="admin,manager" />
  </tomcat-users>
```

To shut down the server, run `shutdown.bat` (on Windows) or `shutdown.sh` (on Unix). If you have configured Tomcat as a Windows service, you will see a Tomcat icon in the system tray. Right-clicking this icon brings up a menu with an Exit option. Select this option to stop Tomcat.

Installing Apache SOAP

To install Apache SOAP, you first need to download the latest stable version from the following URL:

`http://ws.apache.org/soap/`

Unzip the downloaded file to the desired folder. The archive contains a web application archive file called `soap.war`. This file contains the classes that are used for creating SOAP requests and parsing SOAP responses. Copy this file to the `webapps` folder of your Tomcat installation. You also need to copy the `mail.jar` and `activation.jar` files to the `\shared\lib` folder of your Tomcat installation. The `mail.jar` file contains a JavaMail implementation, and the `activation.jar` file contains classes required for supporting Mulipurpose Internet Mail Extensions (MIME) types. Because SOAP is message based, both these files are required. Both come with the J2EE[4] installation or can be downloaded as a part of the JavaMail API from Sun's website.[5]

■**Note** For those of you installing on Linux, Appendix A provides detailed installation instructions for all the chapters in this book.

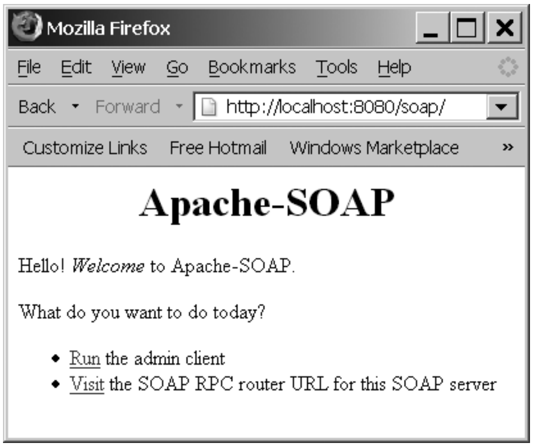
4. The latest version of J2EE is known as Java EE 5.
5. http://java.sun.com/products/javamail/downloads/index.html

Testing the Installation

Test your installation by typing the following URL in your browser:

```
http://localhost:8080/soap
```

You should see a welcome screen, as shown in Figure 4-2.


Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Back Forward
http://localhost:8080/soap/
Customize Links Free Hotmail Windows Marketplace

Apache-SOAP

Hello! *Welcome* to Apache-SOAP.

What do you want to do today?

- Run the admin client
- Visit the SOAP RPC router URL for this SOAP server

Figure 4-2. *Apache SOAP welcome screen*

The welcome screen contains two hyperlinks:

- Run the Admin Client: This hyperlink takes you to a screen that allows you to deploy and undeploy web services. The admin client screen also allows you to retrieve the list of deployed services.
- Visit the SOAP RPC router URL: This hyperlink takes you to the URL `http://localhost:8080/soap/servlet/rpcrouter`. It prints a SOAP RPC router message on your browser. This is an important URL for us. All our clients will be sending the web service requests to this URL.

You can try clicking both the links to ensure that the correct pages open in your browser. Additionally, test the message-routing servlet by typing the following URL in your browser:

```
http://localhost:8080/soap/servlet/messagerouter
```

This prints a message similar to the RPC router on your browser. You will be using this router for calling Document-style web services.

This completes our server-side installation of Apache SOAP. If you are using a different machine for client development, you will need to follow the instructions in the Apache SOAP documentation to set up your client machine. If you use the same machine for both client and server deployment (which would be the case for most of us), you are ready for some real coding of web services.

You can test your client installation by using the following command at the command prompt:

```
C:\>java org.apache.soap.server.ServiceManagerClient ➥
http://localhost:8080/soap/servlet/rpcrouter list
```

This should print the list of all deployed services (which at this stage could be none at all) on your console.

SOAP Implementation Architecture

As you learned in the previous chapter, in a web services implementation the entire communication between a client and a server takes place by using SOAP request/response documents. A SOAP call can be RPC-oriented or document-oriented. When the web service application receives one of these SOAP requests, the application needs to parse the XML document fragment to understand the request and then needs to convert it to an appropriate binary request to the implementing service object. When the service object returns a result to the caller, it is mapped into a SOAP response first and then dispatched to the client. This marshalling and unmarshalling of messages is exactly what the Apache SOAP server implementation provides. The implementation architecture is shown in Figure 4-3.

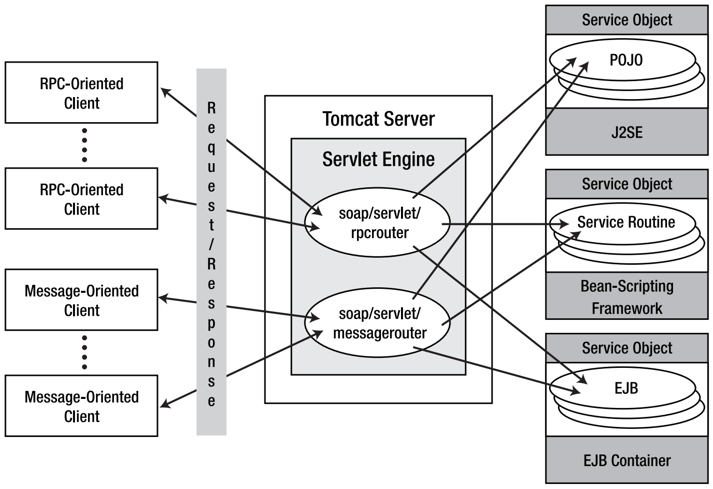


Figure 4-3. *Apache SOAP server implementation*

Apache's SOAP implementation provides two servlets called `rpcrouter` and `messsagerouter`, which together provide the desired functionality of transforming the messages from XML to binary and vice versa. The `rpcrouter` servlet services all the RPC-based requests, and the `messagerouter` servlet services all document-oriented requests. When the client makes a SOAP request to the SOAP server, it specifies the Uniform Resource Name (URN) for the service.

Note The URN is a unique identifier consisting of any text string for each defined service. At the time of deployment, this URN is mapped to the server implementation class. The deployment process is explained in depth later in this chapter.

Each deployed service has an associated unique identifier specified by this URN. The SOAP server uses this URN to invoke the call on the appropriate service object. How does the SOAP server know the appropriate service object? This is resolved with the help of a deployment descriptor (deployment descriptors are described in depth later in this chapter). The deployment descriptor associates a URN with the service object and its methods. Thus, looking at the deployment descriptor, the SOAP server knows which method on which Java class is to be invoked.

The service object itself can be implemented by using different technologies. You can implement the service object as a Plain Old Java Object (POJO), an Enterprise JavaBean (EJB), or a method written in a scripting language. As Figure 4-3 illustrates, both `rpcrouter` and `messagerouter` servlets can redirect the requests to any of these service objects. When the service object returns a response to the caller, the underlying runtime maps this into a SOAP response document and dispatches the response to the caller via the same servlets.

As you can see, the use of a Tomcat server is not unique. You can use any web server that provides the servlet support. Ensure that your web server supports the minimum servlet API version that is required by the Apache SOAP implementation.

Developing Web Services

As you learned in Chapter 3, a web service can be RPC-style or Document-style. In this section, we will develop both RPC- and Document-style web services and write clients for both.

Remember that a web service is essentially a component that exposes its interface by using standard web protocols. This component can be written as a simple Java class, a JavaBean, an EJB, a servlet, or even a .NET component or a Common Object Request Broker Architecture (CORBA) server object. To keep the component development simple so as to concentrate more on the SOAP implementation, we will be using only POJOs to develop the example web services.

Creating an RPC-Style Web Service

We will develop an RPC-style web service for our stock brokerage as discussed in Chapter 3. The web service will provide the requesting client with the latest trade price for a particular stock symbol.

The web service development consists of the following steps:

1. Developing the service that is deployed on the server
2. Developing a client that consumes this service
3. Running the client

Each of these tasks is described in detail here.

Developing the Service

Developing and deploying our web service comes as a result of completing two tasks. First we'll write the server code, and then we'll need to deploy it.

Writing the Server Code

The server code can consist of a POJO or an EJB or scripting language code. We will develop the server using POJO. The server code for our stock brokerage web service is shown in Listing 4-2.

Listing 4-2. *The Stock Quote Service Server Class (*`<working folder>\Ch04\RPC\StockQuoteService.java`*)*

```
package StockBroker;
public class StockQuoteService
  {
    // A web service method
    public float getStockQuote(String symbol)
      {
        // We return the hard-coded value for simplicity
        return ((float)25.35);
      }
  }
```

This is a simple Java class with one public method called `getStockQuote`. The method receives a parameter of `String` type that represents the stock symbol for which the trade price is sought. The method returns a `float` value to the requesting client that contains the last traded price. The method implementation simply returns a fixed value to the client. In real life, the method would read the live database of the stock exchange to retrieve the last traded price.

Compile the code in Listing 4-2 by using the following command line:

```
C:\<working folder>\Ch04\RPC>javac -d . StockQuoteService.java
```

Copy the generated `.class` file (`<working folder>\Ch04\RPC\StockBroker\StockQuoteService.class`) to the following folder:

```
<Tomcat Installation Folder>\webapps\soap\WEB-INF\classes\StockBroker
```

Deploying the Service

After you develop the service, you can choose from two techniques to deploy it on the server:

- Using a GUI-based admin tool
- Using a command-line interface

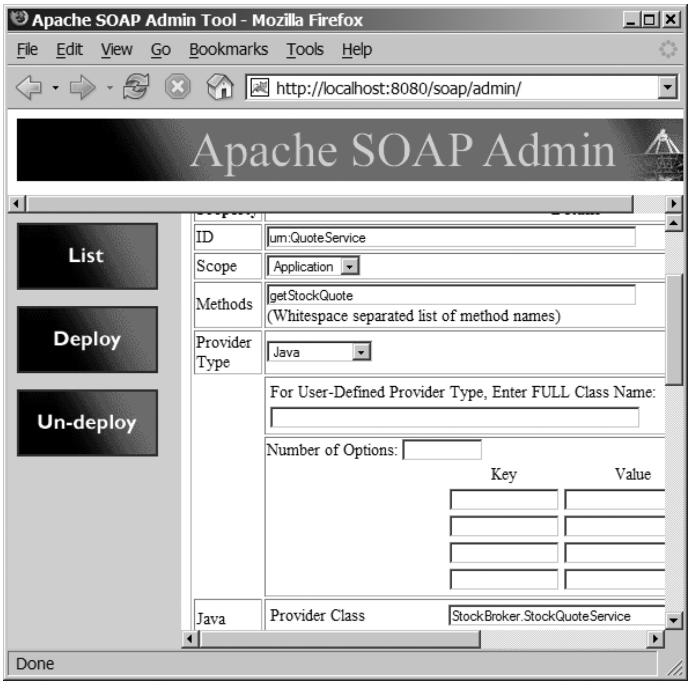
I will describe both methods of deployment.

Deploying by Using a GUI tool

To deploy the service, you use the Admin tool. Type the following URL in your browser to invoke the Admin tool:

```
http://localhost:8080/soap/admin/
```

Click the Deploy button on the Admin screen. This opens a screen as shown in Figure 4-4. All the fields on this screen are initially blank. The screen output shows the values that you will be filling in.


Apache SOAP Admin Tool - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/soap/admin/
Apache SOAP Admin
List
Deploy
Un-deploy
Done

ID: urn:QuoteService
Scope: Application
Methods: getStockQuote
(Whitespace separated list of method names)
Provider Type: Java
For User-Defined Provider Type, Enter FULL Class Name:
Number of Options:
Key Value
Java Provider Class: StockBroker.StockQuoteService

Figure 4-4. *Deployment wizard*

Complete the following steps to fill in the displayed form and deploy the web service:

1. In the ID field, type **urn:QuoteService**. The client supplies this URN in the request. The `rpcrouter` servlet redirects the client request to this URN on the server side.

2. Set the Scope combo box to Application. This option decides the lifetime of the server object. The lifetime may be `Request`, `Session`, or `Application`.

3. In the Methods text box, type **getStockQuote**. Remember, this is the method that our web service wants to expose as a web method that can be invoked by using SOAP.

4. As the Provider Type, select Java. This is the default selection. You can select the Script provider type if your server implementation uses a script code.

5. In the Java Provider group, set the Provider Class as `StockBroker.StockQuoteService`. Note that this is the fully qualified name of our web service class.
6. Use defaults for the rest of the selections.
7. Click the Deploy button on the left side of the screen to deploy the service.
8. If the deployment is successful, you will see a "Service urn:QuoteService deployed" message on your browser.
9. You can click the List button at any time to list all the deployed services.

Deploying by Using a Command Interface

The Apache SOAP implementation provides a Java application called `ServiceManagerClient` that allows you to deploy and undeploy services and to list the available services. To deploy the service by using this command-line utility, you first need to write a deployment descriptor. The deployment descriptor for `QuoteService` is given in Listing 4-3.

Listing 4-3. *Deployment Descriptor for* `QuoteService` *(*`<working folder>\Ch04\RPC\DeploymentDescriptor.xml`*)*

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:QuoteService">
  <isd:provider type="java"
                scope="Application"
                methods="getStockQuote">
    <isd:java
      class="Apress.XMLBook.StockBroker.StockQuoteService" static="false"
    />
  </isd:provider>
  <isd:faultListener>
     org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
</isd:service>
```

The `service` element defines an `id` attribute and sets its value to a namespace that is unique within the current deployment environment. In our case, this is set to `urn:QuoteService`. The `provider` element defines `type`, `scope`, and the web `methods` to be called. If the service exposes more than one method as a web method, all such methods are listed in the `methods` attribute, each separated by a space from the other. The `class` attribute in the `isd:java` element defines the name of the POJO class that implements the service. The deployment descriptors are discussed in depth later in the chapter.

After you write the deployment descriptor for your service, you can deploy the service by using the following command line:

```
C:\<working folder>\Ch04\RPC>java org.apache.soap.server.ServiceManagerClient➥
 http://localhost:8080/soap/servlet/rpcrouter deploy DeploymentDescriptor.xml
```

You can verify that the service has been deployed successfully by listing all the deployed services via the following command line:

```
C:\<working folder>\Ch04\RPC>java org.apache.soap.server.ServiceManagerClient ➥
http://localhost:8080/soap/servlet/rpcrouter list
```

The program will list all the deployed services, as shown here:

```
Deployed Services:

        urn:QuoteService
        urn:AddressFetcher
        urn:Hello
```

To undeploy a deployed service, use the following command line:

```
C:\<working folder>\Ch04\RPC>java org.apache.soap.server.ServiceManagerClient ➥
 http://localhost:8080/soap/servlet/rpcrouter undeploy urn:QuoteService
```

The URN (such as `urn:QuoteService` shown in the preceding statement) uniquely identifies the service to be undeployed.

Developing the Client

Developing a client is a simple process that requires you to write an application that constructs a call to the web service and invokes it. Such a client application can be written in any programming language of your choice. For this example, we will use Java.

To construct a call, you do not need to construct a SOAP request. You simply use provided Java classes to input the desired information for the web service. The underlying SOAP implementation converts the call to an XML SOAP request. Similarly, when the web service returns a response to the client, it does so as a SOAP response. The SOAP response document is interpreted by the SOAP runtime, and the result is returned to the client as a Java object.

Writing the Client Code

We will write a console-based Java application for the purpose of developing a client for our web service. Listing 4-4 provides the complete listing for the console application.

Listing 4-4. *Client Program for Stock Quote Service (*`<working folder>\Ch04\RPC\StockQuoteRequest.java`*)*

```
package StockClient;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
```

```
public class StockQuoteRequest {
  public static void main (String[] args) throws Exception {
    if (args.length != 1)
      {
        System.err.println ("Usage: java StockQuoteRequest symbol");
        System.exit (1);
      }

      String ServiceURL = "http://localhost:8080/soap/servlet/rpcrouter";
      URL url = new URL (ServiceURL);
      String symbol = args[0];

      // Build the call.
      Call call = new Call ();
      call.setTargetObjectURI ("urn:QuoteService");
      call.setMethodName ("getStockQuote");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      Vector params = new Vector ();
      params.addElement (new Parameter("symbol", String.class, symbol, null));
      call.setParams (params);

      Response resp = call.invoke (/* router URL */ url, /* actionURI */ "" );

      // Check the response.
      if (resp.generatedFault ())
         {
          Fault fault = resp.getFault ();

         System.err.println("Generated fault: " + fault);
        }
      else
        {
         Parameter result = resp.getReturnValue ();
         System.out.println (symbol + " last trade: $" + result.getValue ());
        }
    }
}
```

The program sets the service URL in the following statement:

```
String ServiceURL = "http://localhost:8080/soap/servlet/rpcrouter";
```

This is the URL of our `rpcrouter` servlet. The client sends a request to this URL, and the router servlet redirects the request to our POJO providing the service. The underlying implementation converts the request to a binary request required by our POJO service object. Next, the program constructs a `URL` object that points to this service URL, which will be used while invoking this service:

```
URL url = new URL (ServiceURL);
```

Next, the program constructs a Call object for invoking the service:

```
Call call = new Call ();
```

We call the setTargetObjectURI method on the call object to set the target URI:

```
call.setTargetObjectURI ("urn:QuoteService");
```

Note that this is the ID used while deploying the service. Using this information, the rpcrouter servlet routes the requests to the appropriate service.

We call the setMethodName method on the call object to set the desired method call on the web service:

```
call.setMethodName ("getStockQuote");
```

The getStockQuote is the web method exposed by our stock quote web service.

We set the encoding style URI for the request by using the predefined constant:

```
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
```

The value of this predefined constant is http://schemas.xmlsoap.org/soap/encoding/.

Next, we construct the parameter to our web method by using following code snippet:

```
Vector params = new Vector ();
params.addElement (new Parameter("symbol", String.class, symbol, null));
call.setParams (params);
```

The code constructs the parameters for the method by creating a Vector and adding Parameter objects to it. Thus, for multiple parameters, you construct the appropriate desired number of Parameter objects and add them to the Vector object.

Now, you are ready to invoke the call on the web service. You invoke the web service by calling the invoke method on the call object:

```
    Response resp = call.invoke (/* router URL */ url, /* actionURI */ "" );
```

The invoke method takes two parameters. The first parameter is the router URL, which is the URL of our rpcrouter servlet. The second parameter is the action URI, which is set to a null string. This parameter is currently not used and will be implemented in future versions. The invoke method returns a Response object to the caller. The Response object may contain valid data returned by the service or a SOAP fault.

The program checks for the exception by using the following code snippet:

```
    if (resp.generatedFault ())
      {
        Fault fault = resp.getFault ();
        System.err.println("Generated fault: " + fault);
      }
```

If the generatedFault method of the Response class returns true, the server has reported a fault condition. We retrieve the fault information by calling the getFault method on the response object. The program simply prints the fault information on the user console.

On success from the server, the program retrieves the returned value and prints it on the console by using the following code snippet:

```
Parameter result = resp.getReturnValue ();
System.out.println (symbol + " last trade: $" + result.getValue ());
```

Running the Client

Compile the client code by using the following command line:

```
C:\<working folder>Ch04\RPC>javac -d . StockQuoteRequest.java
```

To run the client, you use a `java` command in your command line with the appropriate parameter that represents a stock symbol. The client invocation and its output are shown here:

```
C:\<working folder>Ch04\RPC>java StockClient.StockQuoteRequest MSFT
MSFT last trade: $25.35
```

Investigating the RPC Implementation

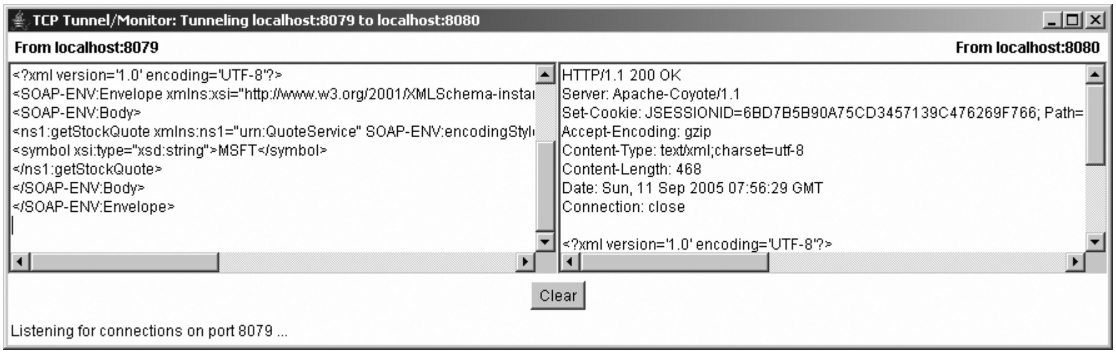
As seen in both the client and server code examples, the developer does not have to deal with any SOAP code. The underlying implementation on both sides takes care of marshalling and unmarshalling the SOAP request and response. If you want to look up the SOAP request and response generated by the underlying implementation, you need to snoop around the network traffic. Fortunately, Apache's SOAP implementation provides a tool for this very purpose.

Running the Network Traffic Interceptor

You invoke the network snooping tool by running the provided Java console application via the following command line:

```
C:\ >java org.apache.soap.util.net.TcpTunnelGui 8079 localhost 8080
```

When you run this program, the screen will appear as shown in Figure 4-5.


TCP Tunnel/Monitor: Tunneling localhost:8079 to localhost:8080
From localhost:8079
From localhost:8080
Clear

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instai
<SOAP-ENV:Body>
<ns1:getStockQuote xmlns:ns1="urn:QuoteService" SOAP-ENV:encodingStyl
<symbol xsi:type="xsd:string">MSFT</symbol>
</ns1:getStockQuote>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=6BD7B5B90A75CD3457139C476269F766; Path=
Accept-Encoding: gzip
Content-Type: text/xml;charset=utf-8
Content-Length: 468
Date: Sun, 11 Sep 2005 07:56:29 GMT
Connection: close

<?xml version='1.0' encoding='UTF-8'?>
```

Listening for connections on port 8079 ...

Figure 4-5. *Network traffic sniffer*

Note When you run the TcpTunnelGui tool, both the panes initially will be blank. Only when you modify and rerun your client application (as described later in this section) will you see the SOAP request and response displayed in the two panes, as seen in Figure 4-5.

This TCP Tunnel/Monitor program sets up a server socket to listen to incoming requests. In our case, we specify 8079 as the port on which this server socket should be listening. When a request is received, the program dumps the request contents in the edit field on the left side of the screen. Then it forwards the request to the URL and the port specified by the second (`localhost`) and the third (`8080`) parameters on the program invocation command line. When the service returns a response, the program dumps the response contents in the second edit field on the right side of the screen.

Intercepting network traffic by using this tool gives us an opportunity to examine the generated SOAP request and response. For this, you will need to modify the client application and set the `ServiceURL` port to `8079` from the earlier value of `8080`. You do this by modifying the source to the following:

```
String ServiceURL = "http://localhost:8079/soap/servlet/rpcrouter";
```

Now, when you run the program (you will need to open another command prompt), the request will be sent to port 8079 on which the Tunnel/Monitor program is listening. The Tunnel/Monitor tool then redirects the request to port `8080`.

This modified program is provided with the source download at the path `<working folder>\Ch04\RPC\StockQuoteRequestNTS.java`. You can compile this program and run it on a separate command window to see the SOAP request and response displayed in the TCP Tunnel/Monitor.

Examining SOAP Requests

When you capture the SOAP request by using the monitor tool, you will see the request in the left panel of the network traffic sniffer window. Listing 4-5 shows this request.

Listing 4-5. *SOAP Request with HTTP Header*

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: localhost:8079
Content-Type: text/xml;charset=utf-8
Content-Length: 452
SOAPAction: ""
Accept-Encoding: gzip

<?xml version="1.0" encoding="UTF-8" ?>
```

```
<SOAP-ENV:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
>
  <SOAP-ENV:Body>
    <ns1:getStockQuote
      xmlns:ns1="urn:QuoteService"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <symbol xsi:type="xsd:string">
        MSFT
      </symbol>
    </ns1:getStockQuote>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

You may want to refer to Chapter 3 to understand the various parts of the SOAP request shown in Listing 4-5. The SOAP request starts with an `xml` declaration that is followed by an `Envelope` element. The `Envelope` element encapsulates the `Body` element and does not contain an optional header. The SOAP body contains a `getStockQuote` element; this is our web method. Note the declaration of `urn:QuoteService` as an XML namespace. The parameter to the method is specified by the `symbol` element of type `xsd:string`. The element text contents are set to `MSFT`.

At the top of the XML declaration, note the use of an HTTP header (refer to the "SOAP over HTTP" section in Chapter 3). The HTTP header contains the required information for transporting the XML document payload to the destination. We use HTTP *post* to invoke the web service. You cannot use HTTP *get* for this purpose. The request is posted to the `rpcrouter` at `localhost:8079`. From this port, it is redirected to port `8080` by the TCP Tunnel/Monitor tool.

Examining the SOAP Response

After executing the request, the server returns a SOAP response to the client. This response indicates either success of the request's execution or failure (in which case a SOAP fault is generated due to an application exception or network error). Listing 4-6 shows a SOAP response document that results from successful execution of the SOAP request in Listing 4-5. The response appears in the right-hand panel of the network traffic sniffer window.

Listing 4-6. *SOAP Response with HTTP Header*

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=8BA93045F0253C0A654A1458D15A3A29; Path=/soap
Accept-Encoding: gzip
Content-Type: text/xml;charset=utf-8
Content-Length: 468
Date: Sun, 11 Sep 2005 07:47:04 GMT
Connection: close
```

```
<?xml version="1.0" encoding="UTF-8" ?>

<SOAP-ENV:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
>
  <SOAP-ENV:Body>
    <ns1:getStockQuoteResponse
      xmlns:ns1="urn:QuoteService"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    >
      <return
        xsi:type="xsd:float"
      >25.35</return>
    </ns1:getStockQuoteResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The XML document is identified by the `xml` declaration as in the previous example. The `Envelope` element is the root element and encapsulates the mandatory `Body` element. The optional `Header` element is not used. The `Body` element contains the return value of `float` type. The return value is identified by the `return` element, whose value is set to `25.35` in the current example.

As in the SOAP request (Listing 4-5), an HTTP header is on top of the XML payload in the SOAP response (Listing 4-6). The header contains an HTTP `200` response code, which indicates an `OK` response. It also contains information on the server that returned the response. The server has also sent a cookie to the client as indicated by the `Set-Cookie` element. The server indicates the date and time of response.

Creating a Document-Style Web Service

Continuing with our stock brokerage example, we will now create a Document-style web service. This service will process a purchase order sent by a client as an XML document fragment. Our web service will receive the purchase order as part of a SOAP request. The web service will parse the request to extract the XML document fragment containing the purchase order. We will parse the XML fragment by using the DOM API (discussed in Chapter 2) to extract the order details. The web service will then generate a confirmation response to the client. The response will be another XML document fragment.

We will write a client that dispatches the purchase order to our web service. We will use a messaging API to send the XML document. Listing 4-7 shows the XML code that we will use as the input document containing the purchase order to be placed on our server.

Listing 4-7. *The* `po.xml` *Document (*`<working folder>\Ch04\Messaging\BrokerApp\po.xml`*)*

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
<s:Body >
<purchaseOrder xmlns="urn:po-processor"
               xmlns:NYSE="http://www.example.com/NYSE/Stocks/Tradeorder">
    <NYSE:scrip>
      MSFT
    </NYSE:scrip>
    <NYSE:quantity>
      1000
    </NYSE:quantity>
    <NYSE:price>
      25.25
    </NYSE:price>
</purchaseOrder>
</s:Body>
</s:Envelope>
```

Developing the Service

Like the RPC-style web service, the development of a document-style web service consists of two parts: writing the server code and deploying the service. Both tasks are described in this section.

Writing the Server Code

We will implement the server in a POJO, as we did for the RPC-style web service. In the RPC-style web service, the Java class was registered with the `rpcrouter` servlet. In this case we will register the class with the `messagerouter` servlet. After receiving the message, the `messagerouter` servlet passes it to the registered Java class. Listing 4-8 shows the complete program for the server code.

Listing 4-8. *Document-Style Web Service (*`<working folder>\Ch04\Messaging\BrokerApp\StockOrderProcessor.java`*)*

```
package StockBroker;

import java.util.Vector;
import org.w3c.dom.Attr;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.apache.soap.*;
import org.apache.soap.rpc.SOAPContext;

/* The StockOrderProcessor defines a method called purchaseOrder that
 * is SOAP-aware
 */
```

```
public class StockOrderProcessor {
  // purchaseOrder method is a SOAP-method
  public void purchaseOrder(Envelope env, SOAPContext reqCtx,
          SOAPContext resCtx)
          throws Exception {

    // Create variable for storing node values
    String scripName = null;
    String quantity = null;
    String price = null;

    // Extract SOAP body
    Body b = env.getBody();
    // Get all the entries in the body and iterate through the list
    Vector entries = b.getBodyEntries();
    for (int i = 0; i < entries.size(); i++) {
      // get the element
      Element e = (Element) entries.elementAt(i);
      // Read the node name
      String nodeName = e.getNodeName();
      // Check if it is purchaseOrder
      if (nodeName.equals("purchaseOrder")) {
        // Iterate through the list of child nodes
        NodeList children = e.getChildNodes();
        for (int j = 0; j < children.getLength(); j++) {
          Node n = children.item(j);
          switch (n.getNodeType()) {
            // for each type of element node, extract the
            // text contents into appropriate variable
            case Node.ELEMENT_NODE:
              if (n.getNodeName().equals("NYSE:scrip"))
                scripName = n.getTextContent();
              else if (n.getNodeName().equals("NYSE:quantity"))
                quantity = n.getTextContent();
              else if (n.getNodeName().equals("NYSE:price"))
                price = n.getTextContent();
              else
                throw new Exception("Unknown element: " + n.getNodeName());
              break;
            case Node.ATTRIBUTE_NODE:
              break;
          }
        }
      }
    }
    // Create a buffer for user response
    StringBuffer response = new StringBuffer(1024);
    // Create SOAP response for the client
    response.append(Constants.XML_DECL)
    .append("<SOAP-ENV:Envelope ➥
             xmlns:SOAP-ENV=\"http://schemas.xmlsoap.org/soap/envelope/\">")
    .append("<SOAP-ENV:Body>")
    .append("<purchaseOrderResponse xmlns=\"urn:po-processor\">")
    .append("<return>")
    .append("Thanks, Received Order for ")
    .append(scripName)
    .append(" quantity= ")
    .append(quantity)
    .append(" price= " + price)
    .append("</return>")
    .append("</purchaseOrderResponse>")
    .append("</SOAP-ENV:Body>")
    .append("</SOAP-ENV:Envelope>");
    resCtx.setRootPart(response.toString(), "text/xml");
  }
}
```

We call our class `StockOrderProcessor` and define a method called `purchaseOrder` in it:

```
public class StockOrderProcessor {
    // purchaseOrder method is a SOAP-method
    public void purchaseOrder(Envelope env, SOAPContext reqCtx,
            SOAPContext resCtx)
            throws Exception {
```

The method is SOAP aware, that is, it is invoked with a SOAP request and returns a SOAP response to the client. The request/response process is achieved through the method parameters.

We expect that the SOAP request will contain the stock name, desired trade quantity, and price elements. The `purchaseOrder` method retrieves the values of these elements from the request document and stores them into the class variables declared as follows:

```
        // Create variable for storing node values
        String scripName = null;
        String quantity = null;
        String price = null;
```

The method then extracts the SOAP body by calling the `getBody` method of the `Envelope` class:

```
        // Extract SOAP body
        Body b = env.getBody();
```

The program retrieves all the entries in the body by calling its `getBodyEntries` method:

```
// Get all the entries in the body and iterate through the list
Vector entries = b.getBodyEntries();
for (int i = 0; i < entries.size(); i++) {
```

The `getBodyEntries` method retrieves a `Vector` object containing all the entries in the body. These entries consist of elements and their attributes. The program reads the element at each node in the vector and retrieves its name:

```
   // get the element
    Element e = (Element) entries.elementAt(i);
    // Read the node name
    String nodeName = e.getNodeName();
```

If the node name equals `purchaseOrder`, we obtain its children by calling the `getChildNodes` method of the `Element` class:

```
    if (nodeName.equals("purchaseOrder")) {
        // Iterate through the list of child nodes
        NodeList children = e.getChildNodes();
```

The `getChildNodes` method returns a `NodeList` object. The `NodeList` object contains a list of child nodes. We iterate through this list to retrieve each child node:

```
        for (int j = 0; j < children.getLength(); j++) {
            Node n = children.item(j);
```

A `switch` statement is used to distinguish between the different node types. The node may be an element node or an attribute node:

```
           switch (n.getNodeType()) {
                case Node.ELEMENT_NODE:
```

Our request document elements do not contain any attributes. Thus, we process only element nodes. We check the name of each node and copy the contents into an appropriate string variable depending on its name:

```
  if (n.getNodeName().equals("NYSE:scrip"))
      scripName = n.getTextContent();
  else if (n.getNodeName().equals("NYSE:quantity"))
      quantity = n.getTextContent();
  else if (n.getNodeName().equals("NYSE:price"))
      price = n.getTextContent();
  else
      throw new Exception("Unknown element: " + n.getNodeName());
```

After the SOAP body is processed, we will generate a SOAP response to the client. For this, first we declare a buffer for storing the response:

```
// Create a buffer for user response
StringBuffer response = new StringBuffer(1024);
```

Then we build the response by adding appropriate XML statements into the buffer. First, we add the XML declaration (available in the Constants class) to indicate that the current document is an XML document:

```
// Create SOAP response for the client
response.append(Constants.XML_DECL)
```

Next, we add the Envelope element indicating that this is going to be a SOAP document:

```
.append("<SOAP-ENV:Envelope xmlns:SOAP- ➥
              ENV=\"http://schemas.xmlsoap.org/soap/envelope/\">")
```

Next, we add the Body element:

```
.append("<SOAP-ENV:Body>")
```

Inside the Body element, we create a purchaseOrderResponse element and another subelement, return, within it:

```
.append("<purchaseOrderResponse xmlns=\"urn:po-processor\">")
.append("<return>")
```

Within the return element, we add the scripName, quantity, and price details obtained earlier:

```
.append("Thanks, Received Order for ")
.append(scripName)
.append(" quantity= ")
.append(quantity)
.append(" price= " + price)
```

Finally, all the tags are closed in the appropriate order, and the created buffer is copied to the response context:

```
.append("</return>")
.append("</purchaseOrderResponse>")
.append("</SOAP-ENV:Body>")
.append("</SOAP-ENV:Envelope>");
resCtx.setRootPart(response.toString(), "text/xml");
```

This completes our server code. Next, we will compile and deploy this server code on the Tomcat server.

Deploying the Service

Compile the server code by using a javac compiler and the following command line:

```
C:\<working folder>\Ch04\Messaging\BrokerApp>javac -d . StockOrderProcessor.java
```

Copy the generated .class file (<working folder>\Ch04\Messaging\BrokerApp\StockBroker\StockOrderProcess.class) to the folder <Tomcat Installation Folder>\webapps\soap\WEB-INF\classes\StockBroker.

You can deploy the server by using the GUI tool discussed earlier or you can deploy it from the command line. To deploy the server from the command line, use the following command:

```
C:\<working folder>\Ch04\Messaging\BrokerApp>java org.apache.soap.server.➥
ServiceManagerClient http://localhost:8080/soap/servlet/rpcrouter deploy ➥
DeploymentDescriptor.xml
```

You will need to set the appropriate Internet Protocol (IP) address and the port number for your server. You will also need the deployment descriptor shown in Listing 4-9.

Listing 4-9. *Deployment Descriptor for Deploying* `StockOrderProcessor` *Application (*`<working folder>\Ch04\Messaging\BrokerApp\DeploymentDescriptor.xml`*)*

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:po-processor" type="message">
  <isd:provider type="java"
                scope="Application"
                methods="purchaseOrder">
    <isd:java class="StockBroker.StockOrderProcessor"
              static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
</isd:service>
```

Note that the service type is declared as `message` by specifying the value of the `type` attribute in the `service` element. The service provider is declared by the `provider` element. The attributes for this element declare the provider type as `java`. The `methods` attribute lists all the service methods separated by a space. In our service class, we provide only one method called `purchaseOrder` that we want to expose as a SOAP-aware method.

The name of the Java class is defined in the `isd:java` element by setting its `class` attribute. This is set to our Java class called `StockOrderProcessor` with its fully qualified name.

The `faultListener` element declares `DOMFaultListener` as the Java class that listens to generated faults.

Our next task is to write a client application that transmits the purchase order XML document to the server.

Developing the Client

Developing the client consists of two parts: writing the client code and running it. Both tasks are described in this section.

Writing the Client Code

Our client is a console-based Java application called `SendMessage` and is shown in Listing 4-10.

Listing 4-10. *Java Client Application That Consumes Document-Style Web Service (*`<working folder>\Ch04\Messaging\BrokerApp\SendMessage.java`*)*

```
package StockClient;

import java.io.*;
import java.net.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import org.apache.soap.*;
import org.apache.soap.messaging.*;
import org.apache.soap.transport.*;
import org.apache.soap.util.xml.*;

public class SendMessage {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println
              ("Usage: java SendMessage SOAP-router-URL envelope-file");
      System.exit(1);
    }

    // Read input XML document file
    FileReader reader = new FileReader(args[1]);
    // Build document tree
    DocumentBuilder builder = XMLParserUtils.getXMLDocBuilder();
    Document doc = builder.parse(new InputSource(reader));
    if (doc == null) {
      throw new SOAPException(Constants.FAULT_CODE_CLIENT, "parsing error");
    }
    // get SOAP Envelope
    Envelope msgEnv = Envelope.unmarshall(doc.getDocumentElement());

    // send the message
    Message msg = new Message();
    msg.send(new URL(args[0]), "urn:action-uri", msgEnv);

    // receive response
    SOAPTransport st = msg.getSOAPTransport();
    BufferedReader br = st.receive();
    // Dump the response to user screen
    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  }
}
```

The client application is a command-line Java application called SendMessage:

```
public class SendMessage {
  public static void main (String[] args) throws Exception {
```

The main method receives two command-line parameters:

```
    if (args.length != 2) {
      System.err.println
              ("Usage: java SendMessage SOAP-router-URL envelope-file");
      System.exit (1);
    }
```

The first command-line argument specifies the URL for the messagerouter servlet, and the second parameter specifies the name of the XML document to be dispatched to the server.

The method reads the input document file by creating a FileReader object on it:

```
    // Read input XML document file
    FileReader reader = new FileReader (args[1]);
```

We create the DocumentBuilder object by calling the getXMLDocBuilder static method of the XMLParserUtils class. We parse the input document by calling the parse method on the builder object:

```
    // Build document tree
    DocumentBuilder builder = XMLParserUtils.getXMLDocBuilder();
    Document doc = builder.parse (new InputSource (reader));
```

The parse method on its successful completion returns the root node in the Document object. The program then extracts the envelope from the document by calling the unmarshall method of the Envelope class:

```
    // get SOAP Envelope
    Envelope msgEnv = Envelope.unmarshall (doc.getDocumentElement ());
```

To send the envelope in a message, we create a Message object:

```
    // send the message
    Message msg = new Message ();
```

The envelope is dispatched in a message by calling the send method on the Message object:

```
    msg.send (new URL (args[0]), "urn:action-uri", msgEnv);
```

The client application now waits for the server response by obtaining the SOAPTransport object and calling the receive method on it:

```
    // receive response
    SOAPTransport st = msg.getSOAPTransport ();
    BufferedReader br = st.receive ();
```

The receive method is a blocking call that waits until the server response is received. After receiving the response, the application dumps its contents on the user console:

```
    // Dump the response to user screen
    String line;
    while ((line = br.readLine ()) != null) {
      System.out.println (line);
    }
```

Running the Client

Compile the application by using the `javac` compiler. Copy the generated `.class` file to the Tomcat installation as described in the previous example.

Run the client application by using the following command line:

```
C:\<working folder>\Ch04\Messaging\BrokerApp>java StockClient.SendMessage ➥
http://localhost:8080/soap/servlet/messagerouter po.xml
```

When you run the application successfully, you should see the following output on your console:

```
<?xml version='1.0' encoding='UTF-8'?><SOAP-ENV:Envelope xmlns:SOAP-ENV="http://
schemas.xmlsoap.org/soap/envelope/"><SOAP-ENV:Body><purchaseOrderResponse xmlns=
"urn:po-processor"><return>Thanks, Received Order for
      MSFT
     quantity=
      1000
     price=
      25.25
    </return></purchaseOrderResponse></SOAP-ENV:Body></SOAP-ENV:Envelope>
```

Exception Handling

The Apache SOAP server provides an exception handler for processing any errors that may occur while invoking the web service. You have seen such exception handlers in our earlier example, where the exception handler class was listed in the deployment descriptor as the value of the `faultListener` element. The code fragment is reproduced here:

```
<isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
```

The `DOMFaultListener` class augments the SOAP fault message with additional information about the fault. The Apache SOAP server also provides another class called `ExceptionFaultListener`. This class wraps the root exception in a parameter.

You can provide your own classes for exception handling to generate application-specific messages. To illustrate this, we will modify our `StockOrderProcessor` class from the previous example to use a custom exception handler. The order processor will check the order value, and will generate a custom exception message to the client if the value exceeds a preset credit limit.

To begin, we will write a custom exception handler.

Writing a Custom Exception Handler

A custom exception handler is a Java class that implements the `SOAPFaultListener` interface. Listing 4-11 provides the code for the custom handler.

Listing 4-11. *Custom Exception Handler* (`<working folder>\Ch04\Messaging\BrokerAppEx\BrokerFaultHandler.java`)

```
package StockBroker;

import org.apache.soap.*;
import org.apache.soap.rpc.SOAPContext;
import org.apache.soap.server.*;

/* The custom Fault Handler implements SOAPFaultListener interface
 */

public class BrokerFaultHandler implements SOAPFaultListener
  {

    /** Creates a new instance of BrokerFaultHandler */
    public BrokerFaultHandler() {
  }

    /* fault method receives SOAPFaultEvent object that may
     * be manipulated by the method
     */
  public void fault(SOAPFaultEvent evt) {
    Fault ft = evt.getFault();
    ft.setFaultString("Application Exception: Exceeded Credit Limit");
  }
}
```

As a part of the interface, the `BrokerFaultHandler` class needs to implement the sole method `fault`. The `fault` method receives an argument type `SOAPFaultEvent`. After the `BrokerFaultHandler` class is registered (this is shown later) with the SOAP server application, the server instantiates this class at the time of deployment. If there is an error, the `fault` method is called with the populated event object sent as a parameter to the `fault` method. From this event object, we retrieve the `Fault` object by calling its `getFault` method. You may now use the various methods of the `Fault` class to get and set its attributes. We use the `setFaultString` method to set the value of the fault string to a desired message.

Modifying the `StockOrderProcessor` Class

You will need to modify the stock order processor class to compute the purchase order value. If this value exceeds the predetermined limit, a custom exception is generated. Listing 4-12 shows the modification required in the `StockOrderProcessor` class.

■Note The modified code is available under the name `StockOrderProcessorEx` class in the downloaded files of this book's source code (`<working folder>\Ch04\Messaging\BrokerAppEx\StockOrderProcessorEx.java>`).

Listing 4-12. *Throwing a Custom Exception*

```
case Node.ELEMENT_NODE:
  if (n.getNodeName().equals("NYSE:scrip"))
    scripName = n.getTextContent();
  else if (n.getNodeName().equals("NYSE:quantity"))
    quantity = n.getTextContent();
  else if (n.getNodeName().equals("NYSE:price"))
    {
       price = n.getTextContent();
       float Amount = Float.parseFloat(price)* Integer.parseInt(quantity);
     if (Amount > 1000)
       throw new Exception ();
    }
  else
    throw new Exception("Unknown element: " + n.getNodeName());
  break;
```

If the node element equals `NYSE:price`, we retrieve the price and multiply it by the previously obtained `quantity`. We compare the product with a predetermined value, and if it exceeds the limit, we throw an exception. Note that the exception will be handled by the exception handler defined in the deployment descriptor.

Our next task is to register the custom exception handler.

Registering a Custom Exception Handler

The exception handlers are registered in the deployment descriptor. Listing 4-13 provides the deployment descriptor for the application.

■Note This deployment descriptor is available in the file `DeploymentDescriptorEx.xml` in the downloaded files of this book's source code (`<working folder>\Ch04\Messaging\BrokerAppEx\DeploymentDescriptor.xml>`.

Listing 4-13. *Deployment Descriptor (*`<working folder\Ch04\Messaging\BrokerAppEx\DeploymentDescriptor.xml`*)*

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:po-processorEx" type="message">
  <isd:provider type="java"
                scope="Application"
                methods="purchaseOrder">
    <isd:java class="StockBroker.StockOrderProcessorEx"
              static="false"/>
  </isd:provider>
  <isd:faultListener>
    StockBroker.BrokerFaultHandler
  </isd:faultListener>
</isd:service>
```

The value for the `faultListener` element specifies the class to be used as a fault handler. Note that you can list multiple `faultListeners` in the deployment descriptor. These listeners will be called in the order they are listed.

Deploying the Application

Compile the source by using the following commands:

```
C:\<working folder>\Ch04\Messaging\BrokerAppEx>javac -d . BrokerFaultHandler.java
C:\<working folder>\Ch04\Messaging\BrokerAppEx>javac -d . StockOrderProcessorEx.java
```

Copy the generated `.class` files to the folder `<Tomcat Installation Folder>\webapps\soap\WEB-INF\classes\StockBroker`.

To deploy the application on Tomcat, use the following command line:

```
C:\<working folder>\Ch04\Messaging\BrokerAppEx>java ➥
org.apache.soap.server.ServiceManagerClient ➥
http://localhost:8080/soap/servlet/rpcrouter deploy DeploymentDescriptorEx.xml
```

You can verify that the application is deployed by using the following command:

```
C:\<working folder>\Ch04\Messaging\BrokerAppEx>java ➥
org.apache.soap.server.ServiceManagerClient ➥
http://localhost:8080/soap/servlet/rpcrouter list
```

Be sure to use the appropriate port number for your installation.

Running the Application

You can run the application by using the following command line:

```
C:\<working folder>\Ch04\Messaging\BrokerAppEx>java StockClient.SendMessage ➥
http://localhost:8080/soap/servlet/messagerouter poEx.xml
```

The SendMessage client in the command line is the same as the one described in the earlier section. The poEx.xml (available in the downloaded source) that will cause an exception is given in Listing 4-14.

Listing 4-14. poEx.xml *That Causes an Exception During Processing (*<working folder>\Ch04\Messaging\BrokerAppEx\poEx.xml*)*

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
<s:Body >
<purchaseOrder xmlns="urn:po-processorEx"
               xmlns:NYSE="http://www.example.com/NYSE/Stocks/Tradeorder">
    <NYSE:scrip>
      MSFT
    </NYSE:scrip>
    <NYSE:quantity>
      1000
    </NYSE:quantity>
    <NYSE:price>
      25.25
    </NYSE:price>
</purchaseOrder>
</s:Body>
</s:Envelope>
```

During processing of the poEx.xml document, the net order is computed as 1,000 multiplied by 25.25. This exceeds the predefined value of 1,000 and thus causes an exception generation. The output produced by running the client application is given here:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:x
sd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENV="http://schemas.xmlsoap.org
/soap/envelope/">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode>SOAP-ENV:Server</faultcode>
<faultstring>Application Exception: Exceeded Credit Limit</faultstring>
<faultactor>/soap/servlet/messagerouter</faultactor>
</SOAP-ENV:Fault>

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

Note that the fault string element contains the application-specific message. To appreciate the importance of custom exception handling, replace the fault handler class in the deployment descriptor with the org.apache.soap.server.DOMFaultListener class and run the application one more time. This time you will see a message on the screen with a long stack trace describing the error.

Such types of error screens do the end user little good. Using the custom exception handlers, you will be able to generate messages that are application-domain specific and more meaningful to the end user.

Data Type Mappings

When you invoke a web service, you may occasionally want to send or receive user-defined data types in addition to the predefined data types. Note that any data type must be marshalled/unmarshalled to an XML data type. Apache SOAP defines mappings for several data types. In addition to primitive data types such as `int` and `float`, the list also contains data types such as `GregorianCalendar`, `Date`, `Vector`, `Hashtable`, `TreeMap`, and `List`. The list is quite exhaustive, and you can refer to Apache SOAP documentation for the complete list.[6]

Creating mappings for user-defined data types is not a complex process. Continuing with our example of a stock brokerage, say we want to offer the client a web service that returns the latest stock information for a desired stock code. The stock information consists of today's high price, today's low price, and the current bid and offer prices. The user sends the stock code to the service as a parameter and expects the four details in the response. One way to implement this would be to send the four values individually in the response. However, if your brokerage application has a built-in-class, say `StockInfo`, that contains all the required information, it would be easy to serialize an instance of this class in the response to the client.

Creating a User-Defined Data Type

You can write your `StockInfo` class representing the user-defined data type as shown in Listing 4-15.

Listing 4-15. `StockInfo` *Class (*`<working folder>\Ch04\StockInfoService\StockInfo.java`*)*

```
package stockinfoservice;

public class StockInfo {
    private float TodayHigh;
    private float TodayLow;
    private float CurrBid;
    private float CurrOffer;

    public StockInfo() {
    }

    /** Creates a new instance of StockInfo */
    public StockInfo(String Symbol) {
        if (Symbol.equals("IBM")) {
            setTodayHigh(25);
            setTodayLow(20);
```

6. Refer to "Creating Type Mappings" in the Apache SOAP v2.3.1 documentation that is the part of the SOAP installation.

```
            setCurrBid(22);
            setCurrOffer(23);
        } else if (Symbol.equals("MSFT")) {
            setTodayHigh(55);
            setTodayLow(50);
            setCurrBid(52);
            setCurrOffer(53);
        } else {
            setTodayHigh(15);
            setTodayLow(10);
            setCurrBid(12);
            setCurrOffer(13);
        }
    }
    public float getTodayHigh() {
        return TodayHigh;
    }
    public void setTodayHigh(float TodayHigh) {
        this.TodayHigh = TodayHigh;
    }
    public float getTodayLow() {
        return TodayLow;
    }
    public void setTodayLow(float TodayLow) {
        this.TodayLow = TodayLow;
    }
    public float getCurrBid() {
        return CurrBid;
    }
    public void setCurrBid(float CurrBid) {
        this.CurrBid = CurrBid;
    }
    public float getCurrOffer() {
        return CurrOffer;
    }
    public void setCurrOffer(float CurrOffer) {
        this.CurrOffer = CurrOffer;
    }
}
```

The `StockInfo` class follows the JavaBeans convention and defines its members as private and provides public getter/setter (accessor/mutator) methods. We define two constructors. The constructor that takes a string argument initializes the object state depending on the value of its argument. We have used hard-coded values for the stock. In real-life situations, you would pick up the values from a real-time database. You need to define a no-argument constructor because it is required by the web service that instantiates this class.

Web Service Implementation

Your web service implementation provides a method that can be invoked by a client to obtain the stock information provided by the `StockInfo` class in Listing 4-15. Listing 4-16 provides the server code that instantiates the `StockInfo` class.

Listing 4-16. *The Stock Info Server That Uses the* `StockInfo` *Class (*`<working folder>\Ch04\StockInfoService\StockInfoServer.java`*)*

```
package stockinfoservice;

public class StockInfoServer {

    /** Creates a new instance of StockInfoServer */
    public StockInfoServer() {
    }

    public StockInfo getStockInfo (String Symbol)
    {
        StockInfo info = new StockInfo(Symbol);
        return info;
    }
}
```

The `getStockInfo` method simply constructs a `StockInfo` object and returns it to the caller. The underlying runtime marshals this data to the appropriate XML data types.

Modifying the Deployment Descriptor

You will need to make a few modifications to your deployment descriptor before deploying the web service. Listing 4-17 shows the modified deployment descriptor.

Listing 4-17. *Deployment Descriptor for Mapping Java Objects (*`<working folder>\Ch04\StockInfoService\DeploymentDescriptor.xml`*)*

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:QuoteService">
  <isd:provider type="java"
                scope="Application"
                methods="getStockInfo">
    <isd:java class="stockinfoservice.StockInfoServer" static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
  <isd:mappings>
      <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:x="urn:xml-stockinfoserver-demo" qname="x:info"
          javaType="stockinfoservice.StockInfo"
          java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
          xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>

</isd:service>
```

You will need to add the mappings element as shown in Listing 4-17 to the deployment descriptor. The mappings element is a subelement of the service element. The mappings element may contain multiple mappings, each defined by using the map subelement.

The map element defines the encoding style by using its encodingStyle attribute. The qualified name for the XML data type is defined by using the qname attribute. The class to be marshalled/unmarshalled is defined by using the javaType attribute.

The map element defines how to serialize this class and uses two attributes that define the classes for serialization and deserialization. The java2XMLClassName attribute defines the class to be used for converting from Java to an XML data type. The BeanSerializer class provides this functionality. The xml2JavaClassName attribute provides the reverse mapping. Again, the Apache-provided BeanSerializer class provides this functionality. If you need custom functionality while serializing or deserializing, you can create your own classes and specify their names in these two attribute values.

Deploying the Service

Compile the two Java files in Listing 4-16 and Listing 4-17 by using the following commands:

```
C:\<working folder>\Ch04\StockInfoService\javac -d . StockInfo.java
C:\<working folder>\Ch04\StockInfoService\javac -d . StockInfoServer.java
```

Copy the generated .class files to the folder <Tomcat Installation Folder>\webapps\soap\WEB-INF\classes\stockinfoservice.

You are now ready to deploy the service. You can use the GUI tool or the following command line to deploy the service:

```
C:\<working folder>\Ch04\StockInfoService java ➥
org.apache.soap.server.ServiceManagerClient ➥
http://localhost:8080/soap/servlet/rpcrouter deploy DeploymentDescriptor.xml
```

This command is the same as the previous commands you have used while deploying web services.

After the service is deployed, our next task is to write the client that invokes this service.

Writing the Client

You will write a console-based Java application that invokes the web service. Listing 4-18 gives the source code for the client application.

Listing 4-18. *Client Application That Uses the* `StockInfo` *Service (*`C:\<working folder>\Ch04\StockInfoService\InvestorClient.java`*)*

```
package stockinfoservice;

// import required classes

public class InvestorClient {

  public static void main(String[] args) throws Exception {

    if (args.length != 2) {
      System.err.println("Usage:");
        System.err.println(
          " java stockinfoservice.InvestorClient SOAP-router-URL StockSymbol");
            System.exit(1);
    }

    String encodingStyleURI = Constants.NS_URI_SOAP_ENC;
    URL url = new URL(args[0]);
    String StockSymbol = args[1];
    SOAPMappingRegistry smr = new SOAPMappingRegistry();
    BeanSerializer beanSer = new BeanSerializer();

    // Map the types.
    smr.mapTypes(Constants.NS_URI_SOAP_ENC,
      new QName("urn:xml-stockinfoserver-demo", "info"),
      StockInfo.class, beanSer, beanSer);

    // Build the call.
    Call call = new Call();

    call.setSOAPMappingRegistry(smr);
    call.setTargetObjectURI("urn:QuoteService");
    call.setMethodName("getStockInfo");
    call.setEncodingStyleURI(encodingStyleURI);

    Vector params = new Vector();

    params.addElement(new Parameter("Symbol",
      String.class,
      StockSymbol, null));
    call.setParams(params);

    // Invoke the call.
    Response resp;
    try {
      resp = call.invoke(url, "");
    } catch (SOAPException e) {
      System.err.println("Caught SOAPException (" +
        e.getFaultCode() + "): " +
        e.getMessage());
      return;
      }

    // Check the response.
    if (!resp.generatedFault()) {
      Parameter ret = resp.getReturnValue();
      stockinfoservice.StockInfo info =
      (stockinfoservice.StockInfo) ret.getValue();
      System.out.println ("Stock Info for " + StockSymbol);
      System.out.println("Today High: " + info.getTodayHigh());
      System.out.println("Today Low: " + info.getTodayLow());
      System.out.println("Current Bid: " + info.getCurrBid());
      System.out.println("Current Offer: " + info.getCurrOffer());
    } else {
        Fault fault = resp.getFault();
      System.err.println("Generated fault: " + fault);
    }
  }
}
```

In the `main` method of the class, create instances of the `SOAPMappingRegistry` and `BeanSerializer` classes as shown here:

```
SOAPMappingRegistry smr = new SOAPMappingRegistry();
BeanSerializer beanSer = new BeanSerializer();
```

Next, you will need to map the data types. This is done by calling the `mapTypes` method of the `SOAPMappingRegistry` class:

```
// Map the types.
 smr.mapTypes(Constants.NS_URI_SOAP_ENC,
        new QName("urn:xml-stockinfoserver-demo", "info"),
        StockInfo.class, beanSer, beanSer);
```

The `mapTypes` method accepts the encoding style as its first parameter. The second parameter specifies the XML data type that is to be marshalled and unmarshalled. The third parameter specifies the Java class name to which the XML data type is marshalled and unmarshalled. The fourth parameter specifies the serialization class that implements this marshalling, and the fifth parameter specifies the class to be used for reverse mapping.

After you register the mappings in the mapping registry, your next task is to construct a call to the server:

```
Call call = new Call();
```

You need to specify the mapping registry that will be used by the `call`:

```
call.setSOAPMappingRegistry(smr);
```

On the `call` object, you set the URN of the web service, the method to be called, and the encoding style URI:

```
call.setTargetObjectURI("urn:StockInfoServer");
call.setMethodName("getStockInfo");
call.setEncodingStyleURI(encodingStyleURI);
```

Next, you need to construct and set the method parameters:

```
Vector params = new Vector();
params.addElement(new Parameter("Symbol",
  String.class,
  StockSymbol, null));
call.setParams(params);
```

In our case, we set the stock code as the only parameter to the method. Note that our web service implementation uses the hard-coded values for two stock symbols, IBM and MSFT. For other stock symbols, it returns another set of hard-coded values. The service implementation can be modified to retrieve the real-time information for the requested stock code and return its value to the client. However, to keep things simple enough so that we keep our focus on mapping user-defined data types, I have avoided this additional coding.

After constructing the `call` object, we call its `invoke` method by passing the URL of the web service as a parameter:

```
Response resp;
String url = "http://localhost:8080/soap/servlet/rpcrouter";

try
{
  resp = call.invoke(url, "");
}
catch (SOAPException e)
{
    System.err.println("Caught SOAPException (" +
      e.getFaultCode() + "): " +
      e.getMessage());
    return;
}
```

The `invoke` method returns a response to the client. Any exceptions are caught in the `SOAPException catch` block.

The program then checks whether the response contains a fault by calling its `generatedFault` method:

```
// Check the response.
if (!resp.generatedFault())
{
  Parameter ret = resp.getReturnValue();
  stockinfoservice.StockInfo info =
    (stockinfoservice.StockInfo)ret.getValue();
  System.out.println ("Stock Info for " + StockSymbol);
  System.out.println("Today High: " + info.getTodayHigh());
  System.out.println("Today Low: " + info.getTodayLow());
  System.out.println("Current Bid: " + info.getCurrBid());
  System.out.println("Current Offer: " + info.getCurrOffer());
}
else
{
  Fault fault = resp.getFault();
  System.err.println("Generated fault: " + fault);
}
```

If the program finds no fault, the program retrieves the response's returned value and prints it on the user console. Note how the return value is mapped to the `StockInfo` class. If the program finds a fault, the program prints the fault details on the console.

Examining the SOAP Request and Response

Use the SOAP message interceptor program described earlier to study the mappings between a Java class and XML in the SOAP request and response. You will need to change the service request port to 8079, where the interceptor is configured to listen to input requests.

The SOAP request is given in the following screen output:

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: localhost:8079
Content-Type: text/xml;charset=utf-8
Content-Length: 450
SOAPAction: ""
Accept-Encoding: gzip

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<ns1:getStockInfo xmlns:ns1="urn:QuoteService"
           SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<Symbol xsi:type="xsd:string">MSFT</Symbol>
</ns1:getStockInfo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Note how the `getStockInfo` method call is embedded in the SOAP body. Also, notice the presence of the `xsd:string` type parameter passed to the method.

When the server executes the request successfully, it returns a SOAP response to the client. The SOAP response is given here:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=46E561B2B240F52681DDB0C90D4D8481; Path=/soap
Accept-Encoding: gzip
Content-Type: text/xml;charset=utf-8
Content-Length: 692
Date: Sun, 08 Jan 2006 16:13:19 GMT
Connection: close

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<ns1:getStockInfoResponse xmlns:ns1="urn:QuoteService"
     SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xmlns:ns2="urn:xml-stockinfoserver-demo" xsi:type="ns2:info">
<currBid xsi:type="xsd:float">52.0</currBid>
<currOffer xsi:type="xsd:float">53.0</currOffer>
<todayHigh xsi:type="xsd:float">55.0</todayHigh>
<todayLow xsi:type="xsd:float">50.0</todayLow>
</return>
</ns1:getStockInfoResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Note the presence of several `xsd:float` elements that map to the individual data members of our `StockInfo` class.

The mappings for both the SOAP request and response are performed by the Apache-provided `BeanSerializer` class.

Deployment Descriptors

You have seen the use of deployment descriptors in the previous examples. You will now learn about their structure and purpose.

Purpose of the Deployment Descriptor

A *deployment descriptor* provides information on the runtime services required by the running application with the help of the `service` element. The `service` element describes the URN for the service, the method and the class name of the POJO class, and more. Note that the service can be provided not only by a POJO, but also by EJBs or by a service routine written in a scripting language. The scripting languages supported by the Bean Scripting Framework (BSF) are described later in this chapter. The contents of the deployment descriptor vary depending on the artifact that is exposed via SOAP.

Deployment Descriptor Structure

The deployment descriptor is basically an XML document. This XML document has a root element called `service`, which is defined in the `http://xml.apache.org/xml-soap/deployment` namespace:

```
  <isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
               id="urn:QuoteService">
      ...
  </isd:service>
```

The `service` element also contains an `id` attribute that defines the URN for the service. When the client invokes the service by using the specified URN, the SOAP server will redirect the call to the provider defined in the `service` element. How to define this provider is discussed next.

Depending on the artifact that is exposed via SOAP, we get three versions of the deployment descriptor:

- Standard Java class deployment descriptor
- EJB deployment descriptor
- BSF script deployment descriptor

Standard Java Class Deployment Descriptor

You have used the standard Java class deployment descriptors in the previous examples. To describe the structure, I have reproduced in Listing 4-19 a descriptor from Listing 4-3.

Listing 4-19. *Structure of a Deployment Descriptor for a POJO*

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:QuoteService">
  <isd:provider type="java"
                scope="Application"
                methods="getStockQuote">
    <isd:java
        class="StockBroker.StockQuoteService"
        static="false"/>
  </isd:provider>
  <isd:faultListener>
    org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
</isd:service>
```

The `provider` subelement provides provider details such as its type, scope, and methods. The code in bold shows the provider used by the service. The `type` attribute specifies that the provider is a Java class. The scope attribute specifies the lifetime of the Java object. This can be Request, Session, or Application. The scope of the object (lifespan) will be the scope as indicated by one of these values. The methods attribute defines the methods of the class that are exposed as SOAP-aware methods. If more than one method needs to be exposed, these will be separated by a space.

The java element defines the fully qualified path of the Java class that provides the service. The fully qualified name is specified as the value for the class attribute. The static attribute specifies whether the class methods are static.

The faultListener element specifies the class to be used for processing faults.

EJB Deployment Descriptor

Listing 4-20 provides an example of an EJB deployment descriptor.

Listing 4-20. *Deployment Descriptor for EJB Provider*

```
<?xml version="1.0"?>
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:testprovider">
  <isd:provider type="org.apache.soap.providers.StatelessEJBProvider"
                scope="Application"
                methods="create">
   <isd:java class="samples/MyHelloService"/>
   <isd:option key="FullHomeInterfaceName" value="samples.MyHelloServiceHome" />
   <isd:option key="ContextProviderURL" value="iiop://localhost:9000" />
   <isd:option key="FullContextFactoryName"
               value="com.ibm.ejs.ns.jndi.CNInitialContextFactory" />
  </isd:provider>
  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
</isd:service>
```

The id attribute of the service element defines a unique identifier that is used by the SOAP server to redirect the call to the appropriate service provider. The type of provider in Listing 4-20 is StatelessEJBProvider. This indicates that the service is provided by a stateless EJB. Note that an EJB can be a stateless bean, a stateful bean, or an entity bean. If the service is provided by a stateful EJB, you specify the type as StatefulEJBProvider. If the service is provided by an entity bean (both bean-managed and container-managed persistence beans), you specify the type as EntityEJBProvider.

Because the life cycle of an EJB is controlled by the EJB container, the only allowed value for the scope attribute is Application. The value for the methods attribute lists all the SOAP-aware bean methods.

The java element defines the name of the bean class as a value of its class attribute. This specifies the local or the remote interface of your EJB.

The option element is specified more than once. The key attribute differentiates between multiple occurrences of the option element. The key value of FullHomeInterfaceName specifies the fully qualified name of your EJB home class. The ContextProviderURL key specifies the

Internet InterOperable Protocol (IIOP) listener on your EJB server. Note that the communication between the EJB client and the server is done using IIOP. You will specify the URL of your EJB's IIOP listener as the value of the `value` attribute. You will need to specify both the IP and the port number while specifying the IIOP listener. The key value of `FullContextFactoryName` specifies the name of the initial context factory class. This class is specific to the application server provider and thus varies depending on the application server you are using.

BSF Script Deployment Descriptor

Listing 4-21 shows the deployment descriptor for a web service provided by a method written in a scripting language.

Listing 4-21. *Deployment Descriptor for BSF Script Language Service*

```
<?xml version="1.0"?>

  <isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
               id="urn:soap-calculator">
  <isd:provider type="script"
                scope="Application"
                methods="ADD SUBTRACT MULTIPLY DIVIDE">
    <isd:script language="javascript">
      function ADD (x, y) {
        return x + y;
      }

      function SUBTRACT (x, y) {
        return x - y;
      }

      function MULTIPY (x, y) {
        return x * y;
      }

      function DIVIDE (x, y) {
        return x / y;
      }
    </isd:script>
  </isd:provider>
  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
</isd:service>
```

In Listing 4-21, the service provider type is defined as `script`. The scope can be `Request`, `Session`, or `Application` as in the case of the POJO class. The `methods` attribute lists all the exposed methods separated by a space character. The `script` element defines these various methods. The `language` attribute specifies the script language. Listing 4-21 defines four JavaScript methods.

Specifying Fault Listeners

You have learned how to create your own fault listeners for providing custom exception handling. Such fault listeners are registered in the deployment descriptor by using the `faultListener` element. In the `faultListener` element, you simply specify the fully qualified Java class name that receives fault events. You can list multiple fault listeners by using the `faultListener` element. Such fault listeners will be invoked in the order they are specified.

Specifying Type Mappings

If you want to map Java class types to XML types, you specify the type mappings in the deployment descriptor. The deployment descriptor provides an element called `mappings` for this purpose. Inside the mappings, you can include multiple occurrences of the `map` element. Each `map` element maps a Java class type to an XML data type. The `map` element defines attributes for specifying a Java class type, an XML data type, and classes used for serializing and deserializing data. You have studied how to specify such type mappings in the earlier section.

Summary

SOAP has been widely accepted as a standard of communication in distributed computing. Apache SOAP provides an important implementation of this SOAP standard. Web services use SOAP for communication between client and server. SOAP is XML based. Thus, a client requesting a service needs to create an XML-based SOAP request and receives an XML-based SOAP response from the server. On the server side, the server must be capable of interpreting a SOAP request and generating a SOAP response.

The Apache SOAP implementation provides the entire framework for generating SOAP requests and responses and parsing SOAP messages. This chapter discussed the architecture of the SOAP engine. The SOAP engine uses two servlets, one for RPC-type calls and another one for Document-type calls. Both servlets use configuration files by way of a deployment descriptor to redirect the calls to the appropriate server objects.

This chapter covered the development and deployment of both RPC-style and Document-style web services based on the Apache SOAP implementation. Both server and client applications were developed. The application may generate runtime errors. The chapter discussed how to handle the exceptions by using provided exception handlers or creating custom exception handlers.

When you invoke a web service, you may need to send a few parameters to the service. The mapping between the standard Java types and XML data types is desired. The Apache SOAP implementation provides such mappings for many standard data types and allows the developer to extend these mappings to other user-defined Java data types. The chapter covered how to map a user-created Java data type to an XML data type.

In a web service implementation, the service object can be a POJO, EJB, or a script code. You specify the type of service object in a deployment descriptor. The deployment descriptor defines the mapping between the service URN and the service object. The deployment descriptor also allows you to declare the custom exception handlers and type mappings. This chapter covered the structure and use of deployment descriptors.