

Chapters *To Go*



Beginning Java Web Services

by Henry Bequet
Apress. (c) 2002. Copying Prohibited.

Reprinted for Venkata Kiran Polineni, Verizon

kiran2chotu@gmail.com

Reprinted with permission as a subscription benefit of **Skillport**,
<http://skillport.books24x7.com/>

All rights reserved. Reproduction and/or distribution in whole or in part in electronic, paper or other forms without written permission is prohibited.



Chapter 3: Creating Web Services with Java

Overview

Chapters 1 and 2 explained the historical and technical background of web services. We discussed the evolution of distributed and interoperable protocols along with technologies that are used in web services such as XML, XML Schema, and SOAP. It is now time to get practical and create our first SOAP-based web service.

Specifically, in this chapter, we will:

- Introduce Axis, which is a **SOAP engine** (or framework).
- Discuss Tomcat, which is a **servlet engine**. Using this selected framework we will be writing a simple web service called `HelloWorld`.
- Look at a web service that uses a JAR file and a **deployment descriptor**. We will see that this additional level of complexity comes with added flexibility.
- Discuss **serialization and de-serialization** – we will focus on the process of mapping Java data types to and from XML documents.

Of course, our first task is select a SOAP engine.

The SOAP Engine

We saw earlier that a major goal of web services is to provide a language-neutral and vendor-neutral platform for distributed applications. In this chapter, we will go from theory to practice by having a close look at a SOAP engine. First of all, let's try to answer the question, which is probably on your minds: what is a SOAP engine? A (Java) SOAP engine is a set of Java classes that facilitate the server-side and client-side development of SOAP applications. Practically speaking, the SOAP engine contains core classes to perform the following operations:

- Serialize method calls into SOAP packets
- Deserialize SOAP packets into Java calls
- Wrap XML documents into SOAP packets
- Unwrap XML documents from SOAP packets
- Submit SOAP requests and handle the responses
- Accept SOAP requests and return the responses

The list of operations is open ended. There is no bounded definition of what a SOAP application is and therefore there is no boundary to what one can expect from the SOAP engine. For instance, if our application supports **asynchronous** operations, should the SOAP engine support the asynchronous traffic or should it put that burden on another component? We will revisit the issues surrounding asynchronous processing in Chapter 8.

Note A list of the popular tools for SOAP development (Java and others) can be found at <http://www.soaprpc.com/software/>.

When it comes to selecting a SOAP engine, the Java developer has many options to choose from. At the time of writing, the most popular choices are:

- **Apache SOAP 2.2** (<http://xml.apache.org/soap/index.html>)
The Apache SOAP engine is probably the most mature tool available to Java developers. However, it does not support more recent technologies such as the Web Services Definition Language (WSDL), which we cover in this book.
- **Apache SOAP 3.0** (<http://xml.apache.org/axis/index.html>)
This Apache SOAP engine is also known as **Axis**. It is not really the next version of Apache SOAP, since it is a complete rewrite. According to the documentation, Axis stands for **Apache eXtensible Interaction System**.
- The **Web Services Developer Pack** (<http://java.sun.com/webservices/>)
This package from Sun Microsystems is in fact a bundle of several technologies. For instance, we can use the **Java API for XML-Based Remote Procedure Call (JAX-RPC)**, the **Java API for XML Processing (JAXP)** for document-style SOAP, and so on.
- The **IBM Web Services Toolkit (WSTK)** (<http://www.alphaworks.ibm.com/tech/webservicestoolkit>)
This download from IBM is similar to the Web Services Developer Pack in the sense that it also is a bundle of several technologies. It uses Axis as a SOAP engine and WebSphere as a web server. We discuss the WSTK in more detail in Chapter 7.
- The **IBM WebSphere SDK for Web Services (WSDK)** (<http://www-106.ibm.com/developerworks/webservices/wsdk/>)
This is a more 'official' web services pack from IBM (compared to the beta code of the WSTK above). It also comes with Axis, as well as lots more components such as Xerces, a private UDDI registry, a test databases, WebSphere web server, etc. We will be using the WSDK in Chapter 10.

Axis

We have decided to use Axis as the SOAP engine in this book, for the following reasons:

- Axis is a vendor-neutral offering.
- Axis has been adopted by several vendors. We mentioned IBM with WSTK, but other vendors such as Macromedia use the Axis implementation as part of their J2EE server (JRun).
- Axis provides an implementation of JAX-RPC. The goal is to provide an implementation that is compliant with JAX-RPC 1.0. At the time of writing, the current version of Axis is Beta 3.
- Axis provides an extensible implementation that allows for tremendous flexibility in customization.

Note Please note that, in the course of this book, not all the products we use are vendor neutral. For example, in Chapter 7, when we introduce the tools necessary for the publication and discovery of web services, we will rely on IBM's Web Services Developer Toolkit.

Before we can start working with Axis, we need to make sure that we have two other software components. They are the Java Development Kit (1.3), and a JAXP-1.1-compliant XML parser such as Xerces or Crimson. (We will be using Xerces).

Note The Java API for XML Processing (JAXP) supports the parsing of XML documents. More details about JAXP can be found in Chapter 2.

We will assume that you have the JDK already installed and that the environment variable `java_home` points to its installation directory, since we will be making use of this variable in our examples.

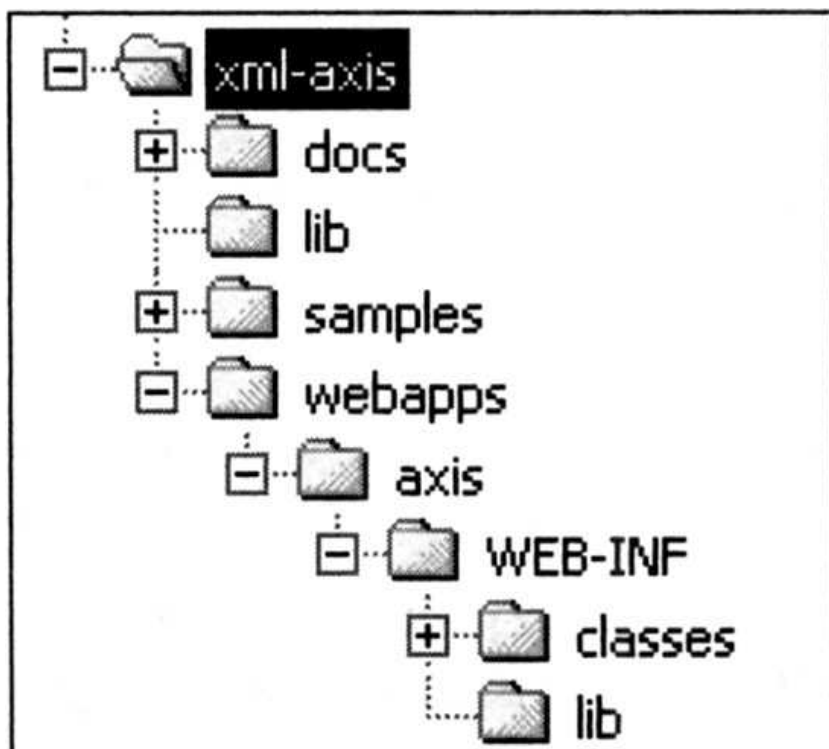
Try It Out: Download and Install Axis

The following instructions are for a computer running Windows 2000, but it is easy to adapt them for other platforms.

1. To download Axis, point your browser to the URL <http://xml.apache.org/axis>.
2. From the links given, select the Releases link presented under the Download section on the left hand side of the screen. At the time of writing, Axis is still in Beta so we downloaded `xml-axis-beta3.zip`.

Important Check the code download from <http://www.wrox.com> for any changes to these instructions on the release of later versions of Axis.

3. After the file is downloaded we decompress the files to `C:\` and rename the directory `C:\xml-axis-beta3` as `C:\xml-axis`. This will help us to keep the path names in our classpath to a reasonable length. At the end of this process, the Axis directory on our machine will look like this:



4. To simplify our life during development, we will use two environment variables (also known as shell variables):
 - `axisDirectory`:
The Axis installation directory (in our case: `C:\xml-axis`)
 - `jwsDirectory`:
The root directory where we will be putting the examples of this book (in our case `C:\Beginning_JWS_Examples`)

5. To run Axis we need only two files out of the Xerces package that we should have downloaded in the previous chapter:

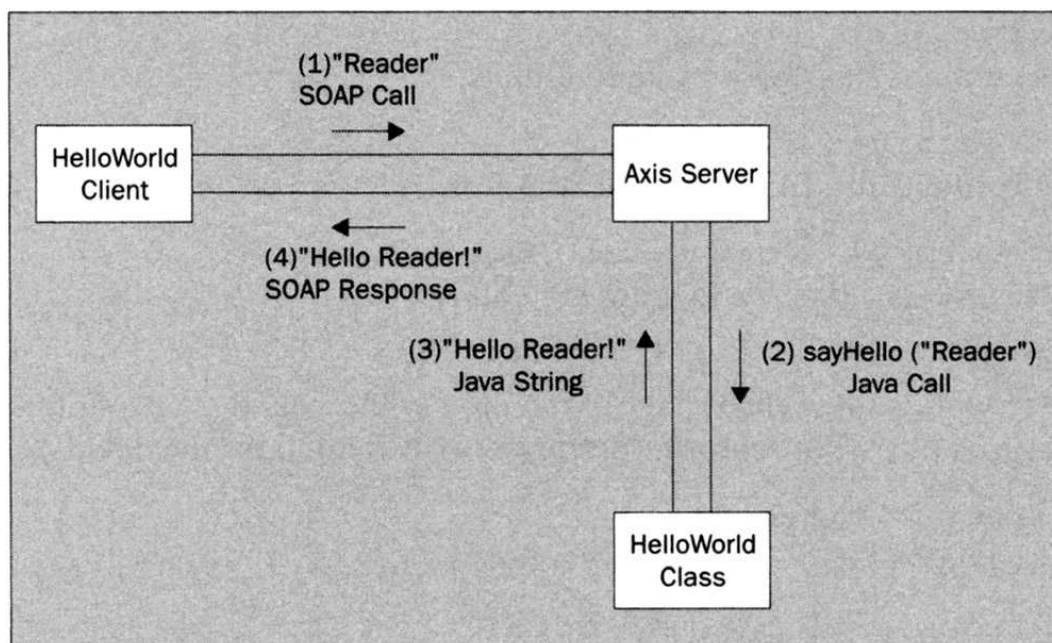
- `xercesImpl.jar`:
A Java Archive File (JAR) file containing all the parser class files, which implement the standard APIs supported by the parser
- `xmlParserAPIs.jar`:
A Java Archive File (JAR) file containing the definitions of the standard APIs implemented by the parser

We will copy these two files into our `%jwsDirectory%\lib` directory, but feel free to put them anywhere else on your machine. The important part is to include these two JAR files in our classpath, while running Axis.

We are now ready to try out the Axis SOAP engine with a simple web service. In the [next section](#) we will discuss one such simple web service called `HelloWorld`, and run it ourselves.

Writing a Web Service

Before getting into the details of creating a web service using Axis, let's have a look at the end product. The simplest possible service is a web service that returns fixed data such as an integer or a string. To convince ourselves that our web service responds to our input, we will write one that takes a string as input and returns a modified version of the input string. Specifically, if we send the string `Reader` to the service, it will respond with the string `"Hello Reader!"` The following figure depicts what we are trying to achieve:



As shown in the above diagram, the sequence of events is as follows:

1. The `HelloWorld` client submits a SOAP request to the SOAP engine, Axis.
2. The Axis server receives the request and forwards it as a Java call to the `HelloWorld` class. Notice that we said, "class", not "service". In other words, we do not have to do anything special to morph our class into a web service under this scenario: we only write a simple Java class and the SOAP engine does the rest. We will discuss the specifics shortly.
3. The `sayHello()` method of our class simply returns a Java string back to Axis.
4. Axis serializes this Java string into a SOAP response, which in turns gets processed by the client.

Before we can actually try out the `HelloWorld` client, we must briefly discuss the HTTP server.

SimpleAxisServer

To service remote calls over HTTP, we need a web server (it is also known as a HTTP Server). This server will be responsible for handling the

HTTP requests (GET and POST). The web server listens to a socket port, typically 80 in production and 8080 in development. When a request arrives at this port from the client, the web server forwards the request to the appropriate software component for processing. In the case of a SOAP request, the proper software component will be the SOAP engine (in this case, Axis).

When it comes to choosing a web server, there are several options available to us. We will start with the easiest, the web server that comes with Axis (`SimpleAxisServer`): `org.apache.axis.transport.Web.SimpleAxisServer`.

In order to try out our `HelloWorld` example, we need to start up our server, but before starting the `SimpleAxisServer`, we need to make sure that the following JAR files are in the classpath (as mentioned, `%axisDirectory%` is the installation root for Axis, `%jwsDirectory%` is the installation root of our examples, and `%java_home%` is the JDK directory):

- `%axisDirectory%\lib\axis.jar`
This JAR contains the Axis implementation.
- `%axisDirectory%\lib\jaxrpc.jar`
This JAR contains the JAX-RPC declarations.
- `%axisDirectory%\lib\commons-logging.jar`
This JAR contains Apache's logging capability.
- `%axisDirectory%\lib\tt-bytecode.jar`
This JAR contains the Tech Trader bytecode toolkit, which is set of classes used to modify Java bytecodes (see <http://sourceforge.net/projects/tt-bytecode/> for details).
- `%axisDirectory%\lib\saaaj.jar`
This JAR contains the SOAP implementation.
- `%axisDirectory%\lib\log4j-core`
This JAR contains the Log4J classes for additional logging.
- `%jwsDirectory%\lib\xercesImpl.jar` and `%jwsDirectory%\lib\xmlParserAPIs.jar`
These JAR files contain the XML parser.
- `%java_home%\lib\tools.jar`
This JAR is required to compile the JWS file.

Note You may find it easiest to create a batch file such as the one included with the code download to set the classpath before you begin to compile or execute any of the example classes.

Once you have this set up correctly, we can begin with our example.

Try It Out: Hello World

Here are the steps to try out the `HelloWorld` service:

1. From within the `\Chp03\HelloWorld` directory, start the `SimpleAxisServer` by typing in the following command:
`java org.apache.axis.transport.http.SimpleAxisServer`

If all has gone well, you will see a message confirming that the server is starting on port 8080, like this:



Note By default, the `SimpleAxisServer` listens to port 8080, but you can change the port by passing the port number (for example 8081) as a command-line argument.

Once started, the `SimpleAxisServer` does not give the control back to the command prompt so we will have to open another one for running the client. At this point, we have a web server ready to process SOAP requests.

2. Now we need to install the `HelloWorld` web service. The code of the `HelloWorld` web service is listed below:

```
public class HelloWorld {
    /**
     * Returns "Hello <sender>!".
     */
    public String sayHello(String sender) {
```

```

    return "Hello " + sender + "!";
}
}

```

Installing a web service is usually referred to as **deployment**. For deploying the HelloWorld web service, we don't have to compile it as Axis will automatically do this for us. We just have to copy the HelloWorld.java (you can download the source code for this file from our web site <http://www.wrox.com>) file to the root directory of the SimpleAxisServer and rename it as HelloWorld.jws (JWS stands for **Java Web Service**):

Important The SimpleAxisServer looks for JWS file in the directory where it was started up, and not in the Axis installation directory. So, from our screenshot opposite, you can see that we should copy the HelloWorld.jws file to %jwsDirectory%\Chp03\HelloWorld\HelloWorld.jws

This simple operation completes the deployment of the HelloWorld web service.

3. The last step that we need to complete before seeing HelloWorld in action is to build the client, HelloWorldClient.java.

As we can see from the following snippet, the code for the client is slightly more complex than the HelloWorld web service, but we will review it in detail shortly:

```

// The Axis package is used to generate and handle the SOAP call
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;

// The rpc package is used to create the RPC call
import javax.xml.namespace.QName;
import javax.xml.rpc.NamespaceConstants;

// The java.net package gives us a URL class
import java.net.URL;

public class HelloWorldClient {
    /**
     * Test main.
     */
    public static void main(String args[]) {
        System.out.println("HelloWorld.main: Entering...");

        try {
            String url = "http://localhost:8080/HelloWorld.jws";
            String sender = "Reader";
            Service service = new Service();
            Call call = (Call) service.createCall();

            call.setTargetEndpointAddress(new URL(url));
            call.setSOAPActionURI("sayHello");
            call.setEncodingStyle(NamespaceConstants.NSURI_SOAP_ENCODING);
            call.setOperationName(new QName("urn:helloworld", "SayHello"));
            //call.setReturnType(XMLType.XSD_STRING);

            String hello = (String)call.invoke(new Object[] { sender } );

            System.out.println("The Web Service returned: " + hello);
            System.out.println("HelloWorld.main: All done!");

        } catch (Exception exception) {
            System.err.println("Caught an exception: " + exception);
        }
    }
}

```

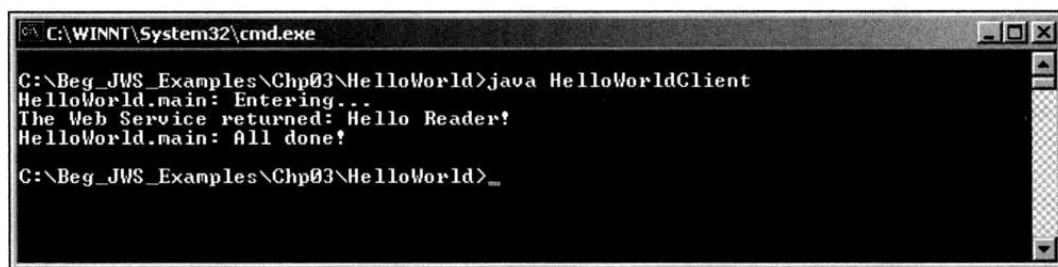
Compiling this code can easily be achieved by the following command. We have assumed that this is saved in the %jwsDirectory%\Chp03\HelloWorld directory (as you will see from the code download):

```
javac HelloWorldClient.java
```

Important Remember you need to have set your classpath to include the Axis and Xerces JAR files, as described earlier.

4. The command to run the HelloWorldClient is:

```
java HelloWorldClient
```



```
C:\WINNT\System32\cmd.exe
C:\Beg_JWS_Examples\Chp03\HelloWorld>java HelloWorldClient
HelloWorld.main: Entering...
The Web Service returned: Hello Reader!
HelloWorld.main: All done!
C:\Beg_JWS_Examples\Chp03\HelloWorld>
```

Congratulations! You have deployed and tested your first web service.

If everything works as described, read on. However, in case you experienced any problems the [next section](#) deals with some common problems you might encounter.

Troubleshooting

It is frustrating to try out something for the first time and have it fail repeatedly. Here are a few common exceptions that may get thrown up while trying out the HelloWorld example:

- Caught an exception: `java.io.FileNotFoundException: HelloWorld.jws` (The system cannot find the file specified)"

This error occurs when Axis is unable to find the JWS file. It is a very common problem the first time you work with a JWS file. To fix the problem, stop the SimpleAxisServer and double-check that the HelloWorld.jws file is present **in the directory where you started the SimpleAxisServer**. After verifying that the file is present, restart the SimpleAxisServer.

- Caught an exception: `java.lang.RuntimeException: Compiler not found in your classpath`. Make sure you added 'tools.jar'

This error occurs when Axis is unable to locate the tools.jar file. The reason is that without tools.jar, Axis cannot compile the JWS file. Please ensure that the tools.jar file is in your classpath.

- Caught an exception:

There can be several reasons for this error. In this case the window where the Axis server is running will probably contain a stack trace, from which we can usually find out the cause. As with Java development, the problem is likely to be a missing JAR file in the classpath. Make sure that the classpath (server and client) contains all the files that we listed above. Note that when we get an error like this one, the Axis server might crash. If this happens remember to restart it.

How It Works

The code of the HelloWorld web service is very simple. This web service is made up of just one Java method, `sayHello()`. Its implementation is rather trivial and does not contain any Axis-specific code.

Most of the work is actually in the client, so let's have a closer look at that code. The imported `org.apache.axis` package contains Axis's implementation of the JAX-RPC package. The imported `javax.xml` package contains the declarations (without the implementation) of the classes and interfaces defined by JAX-RPC. The client code is contained in `HelloWorld.main()`:

```
public static void main(String args[]) {
    System.out.println("HelloWorldClient.main: Entering...");

    try {
        String url = "http://localhost:8080/HelloWorld.jws";
        String sender = "Reader";
        Service service = new Service();
        Call call = (Call) service.createCall();
```

The URL of our web service is `localhost:8080`, port 8080 on the local machine. The `Service` class models a remote web service and the `Call` class models a remote procedure call to a (remote) web service.

Once we have a `Call` object, we must give it enough information to generate the SOAP packet, that is:

- The remote URL
- The SOAP action (`sayHello`)

- The encoding (SOAP)
- The remote method to call (`sayHello()`)
- The type of the object returned by the remote procedure (`String`)

The SOAP action is optional, but some servers use it as a hint of the method being called. Setting the return type is also discretionary when talking to Axis because it puts type information in the SOAP response. This is not necessarily the case with other SOAP engines, so to be safe you might want to specify the return type explicitly. We will examine the SOAP request and response shortly.

The code that performs all this is listed here:

```
call.setTargetEndpointAddress(new URL(url));
call.setSOAPActionURI("sayHello");
call.setEncodingStyle(SOAPConstants.URI_NS_SOAP_ENCODING);
call.setOperationName(new QName("urn:helloworld", "sayHello"));
//call.setReturnType(XMLType.XSD_STRING);
```

The actual call to the remote procedure is done through `call.invoke()` (notice the cast to a `String`):

```
String hello = (String)call.invoke(new Object[] { sender } );
```

The remainder of the code prints the string returned by the web service. It also contains the exception-handling code.

As we have said before, the bulk of the `HelloWorld` implementation is in the client. When we introduce WSDL in Chapter 5, we will see that it is possible to automatically generate a proxy for handling the details of setting up the `Service` and the `Call` objects.

Let's now have a look at the SOAP packets transmitted between the client and the server.

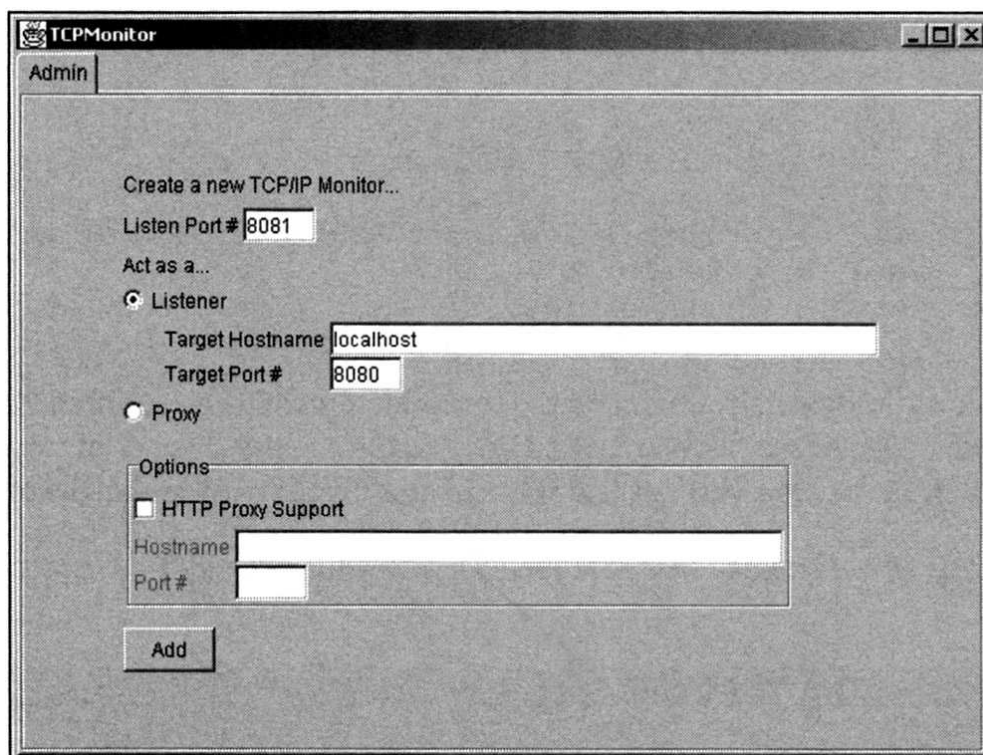
Try It Out: SOAP Debugging

Axis comes with a very useful class `tcpmon` (TCP monitor). It is a Swing-based utility used for setting up a TCP tunnel between our client (`HelloWorldClient`) and the server (the `HelloWorld` web service).

1. The first step is to start the `tcpmon` utility (Remember to set the same classpath as in the `HelloWorld` client):

```
java org.apache.axis.utils.tcpmon
```

This command should display a Window as shown below into which we can enter the values shown here (the text fields are empty by default):

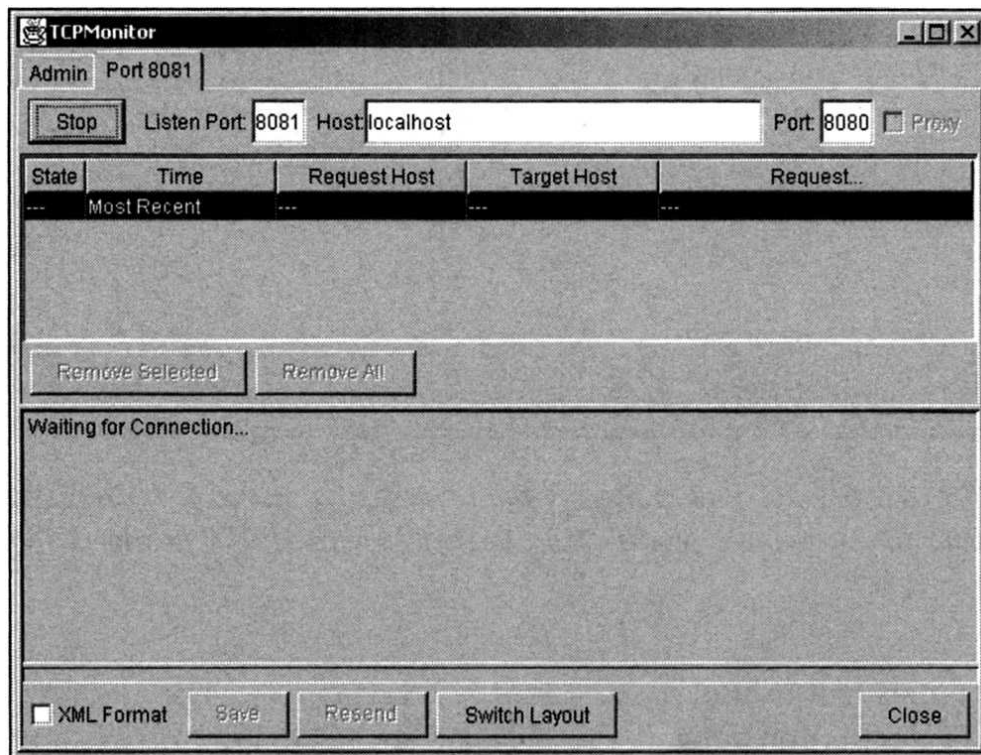


Important Do not use the same port values for the Listen Port and the Target Port or else we will get into an infinite loop.

2. Once we've entered the values as shown above screenshot, click on the `Add` button. This will create a new tab in the window labeled

Port 8081.

- Click on the Port 8081 tab, to show its contents. The display should now look like the following:

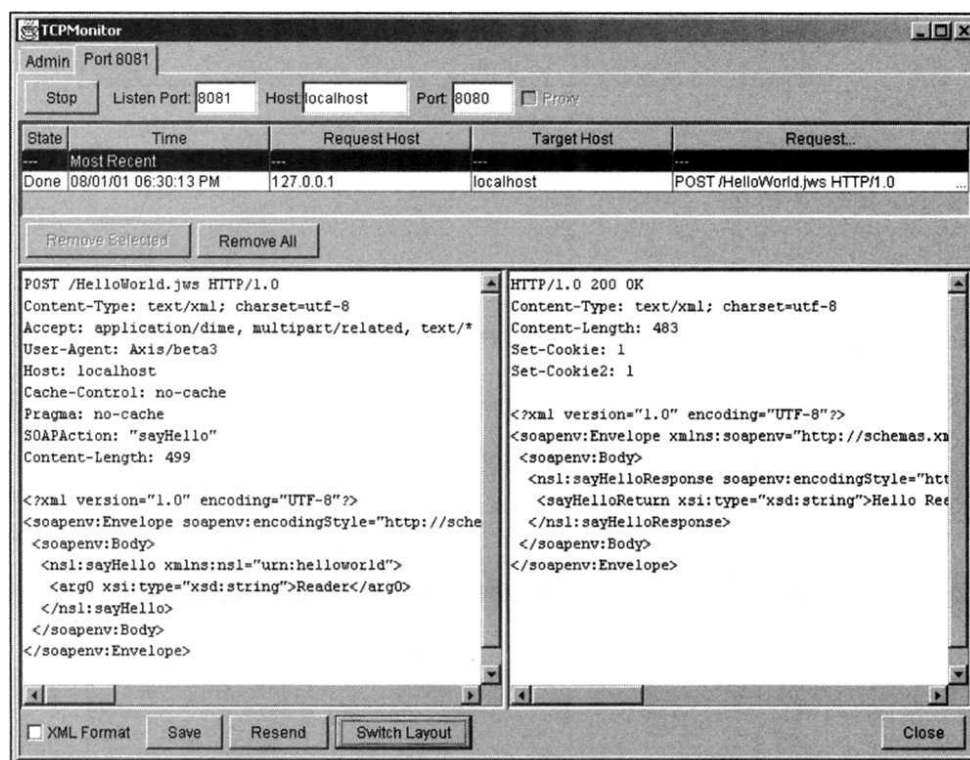


- Since we have set up the `tcpmon` utility to listen to port 8081, we need to modify our client to submit its requests to port 8081. This is relatively simple; we just have to change one line in our code for `HelloWorld`. For this we will open the file `HelloWorldClient.java` in a text editor and change the value as shown below:

```
public static void main(String args[]) {
    System.out.println("HelloWorldClient.main: Entering...");
    try {
        String url = "http://localhost:8081/HelloWorld.jws";
        String sender = "Reader";
        Service service = new Service();
```

- The JWS file should remain unchanged. Compile and run the modified `HelloWorldClient`. After the execution of `HelloWorldClient.main()`, the display of `tcpmon` should look like this (we can switch between a vertical and a horizontal display using the `Switch Layout` button):

Note When running this example make sure that the `SimpleAxisServer` is listening to port 8080, its default value.



Note Note that it is theoretically possible to modify the request and resend it to the server. If you want to do so, keep in mind that you must modify the Content Length HTTP header to reflect the change.

How It Works

Let's start with the request submitted by `HelloWorld.main()`. The request starts with an HTTP header that contains information such as the HTTP verb (POST), the version of the protocol, the length and type of the content, and the optional `SOAPAction` header that we discussed previously:

```
POST /axis/HelloWorld.jws HTTP/1.0
Content-Length: 505
Host: localhost
Content-Type: text/xml; charset=utf-8
SOAPAction: "sayHello"
```

Note that the HTTP header is terminated by an empty line and followed by the content of the document. The SOAP request is an RPC call to the `sayHello()` method of the web service qualified with the `urn:helloworld` namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <ns1:sayHello xmlns:ns1="urn:helloworld">
      <arg0 xsi:type="xsd:string">Reader</arg0>
    </ns1:sayHello>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The response is a SOAP response to the RPC and it is a return value. Once again the HTTP header is terminated by an empty line and followed by the SOAP envelope:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 489
Set-Cookie: 1
Set-Cookie2: 1

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

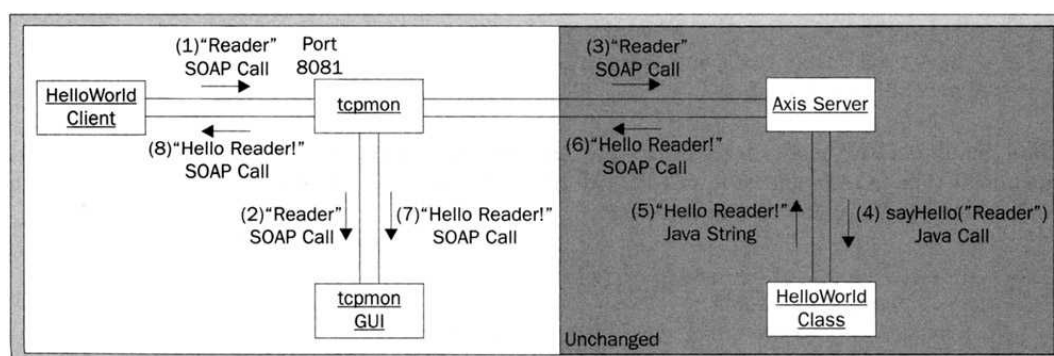
```

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <ns1:sayHelloResponse SOAP
      ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" .
      xmlns:ns1="urn:helloworld">
      <sayHelloReturn xsi:type="xsd:string">Hello Reader!</sayHelloReturn>
    </ns1:sayHelloResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

As we can see from this example, when it comes to debugging SOAP applications, the `tcpmon` utility can be a priceless asset. Now let's spend some time in discussing the working of the TCP Monitor.

The following diagram shows how the message flow is modified for `HelloWorld`:



When we set the listening port to 8081, we instructed `tcpmon` to listen to TCP/IP (and therefore HTTP) connections on port 8081. We also modified the client to connect to 8081 rather than 8080. When the request comes in to port 8081 (label 1) in the figure above, `tcpmon` does two things. Firstly, it displays the request in its GUI (label 2) and secondly, it forwards the same request to the target host (label 3).

In our case, the target host is port 8080 where the `SimpleAxisServer` is waiting for the request. From the point of view of the server (in our case the `SimpleAxisServer`), nothing has changed (labels 4 and 5). When the response is ready, it does not come back to our client, instead it first comes back to `tcpmon` (label 6). When `tcpmon` receives the response it updates its GUI (label 7) and simultaneously forwards the response to the client (label 8), the `HelloWorld.main()` that submitted the request to port 8081.

Note that the use of `tcpmon` is not restricted to SOAP traffic alone; we can use it for our HTML or JSP development.

Let's now shift our attention to a production-grade servlet engine, Tomcat.

Using Tomcat as the HTTP Server

We had mentioned in passing that the `SimpleAxisServer` was not production-worthy. It has several shortcomings, such as:

- **It is not robust**
As we said earlier, the server will crash and consequently stop serving all HTTP requests. This is simply not acceptable in a production environment.
- **It is not secure**
It provides no security mechanism whatsoever. A robust security mechanism is one of the prerequisites for a production-grade server.
- **It is not scalable**
The `SimpleAxisServer` can only serve one request at a time because it is single threaded. As a result, it cannot handle high workloads in a production environment.

So, is it fair to say that the `SimpleAxisServer` is useless? Absolutely not! Its main advantages are its simplicity and lightweight nature. These advantages are very handy during web service development. As `SimpleAxisServer` is single threaded, we can be sure that the server is performing no other work while we are working on our code. This is a major asset when it comes to debugging or profiling. Also, being lightweight, the `SimpleAxisServer` will start quickly. So make sure that you do not forget the appropriate uses of the `SimpleAxisServer` once you have learned how to set up Axis with a servlet engine.

In this book we will be using Tomcat 4, which is the most current version at the time of writing (4.0.4 to be precise). Tomcat is an attractive solution as it is an open source (vendor-neutral) servlet engine. It is also the servlet reference implementation selected by Sun. Other servlet engine implementations provided by WebSphere or WebLogic application servers are also available. We will be talking about application servers in Chapter 10.

Tomcat addresses the main concerns raised about the `SimpleAxisServer`:

- **Robustness**

The crash of a servlet will not bring down the server.

- **Security**

It implements the J2EE security specifications. See <http://java.sun.com/j2ee> for a short introduction to J2EE.

- **Scalability**

It is a multi-threaded server that can, under some circumstances, be replicated across several machines.

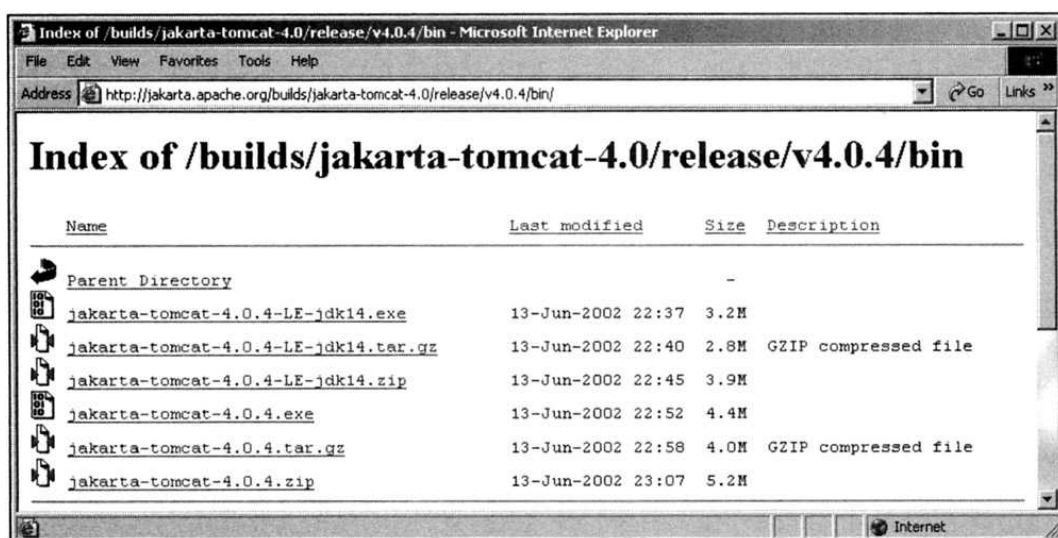
We have touched on the fact that these desired features came at the expense of simplicity. The first manifestation of this added complexity is the download and installation of Tomcat. But don't worry; with the instructions provided in the coming pages, setting up Tomcat with Axis will be a breeze.

Try It Out: Installing Tomcat

Let's go through the steps of installing Tomcat.

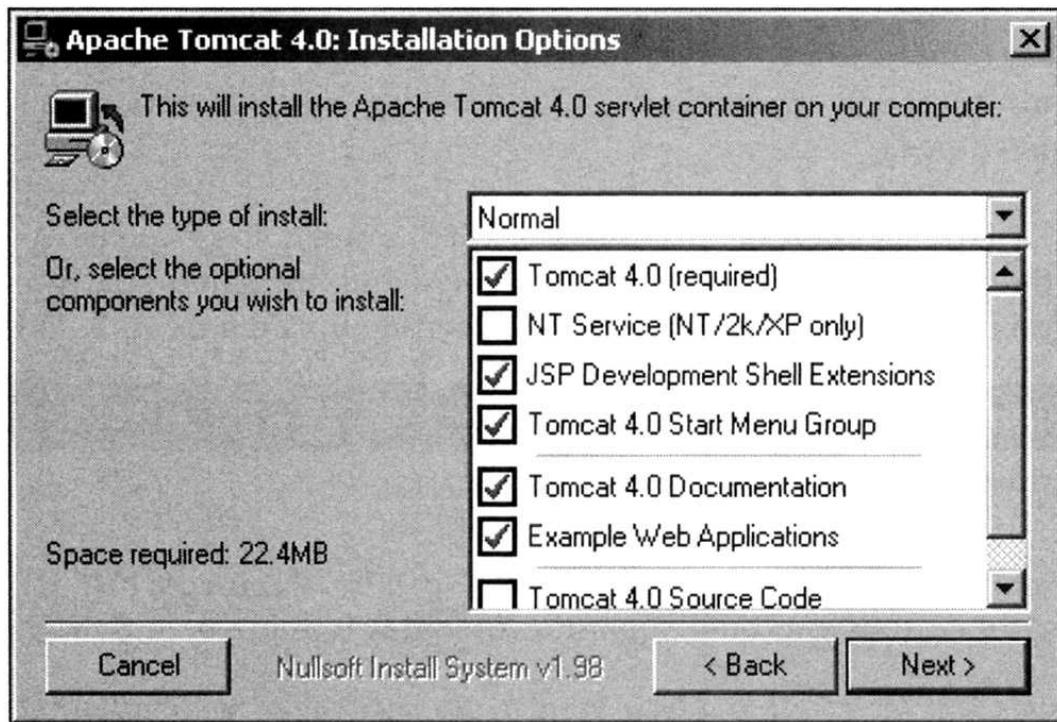
1. The recent release of Tomcat (Tomcat 4.0.4) can be obtained from <http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/v4.0.4/bin/>:

Note It's quite possible that there will be a later build by the time you're reading this; however, the instructions should be identical; only the version number will be different.

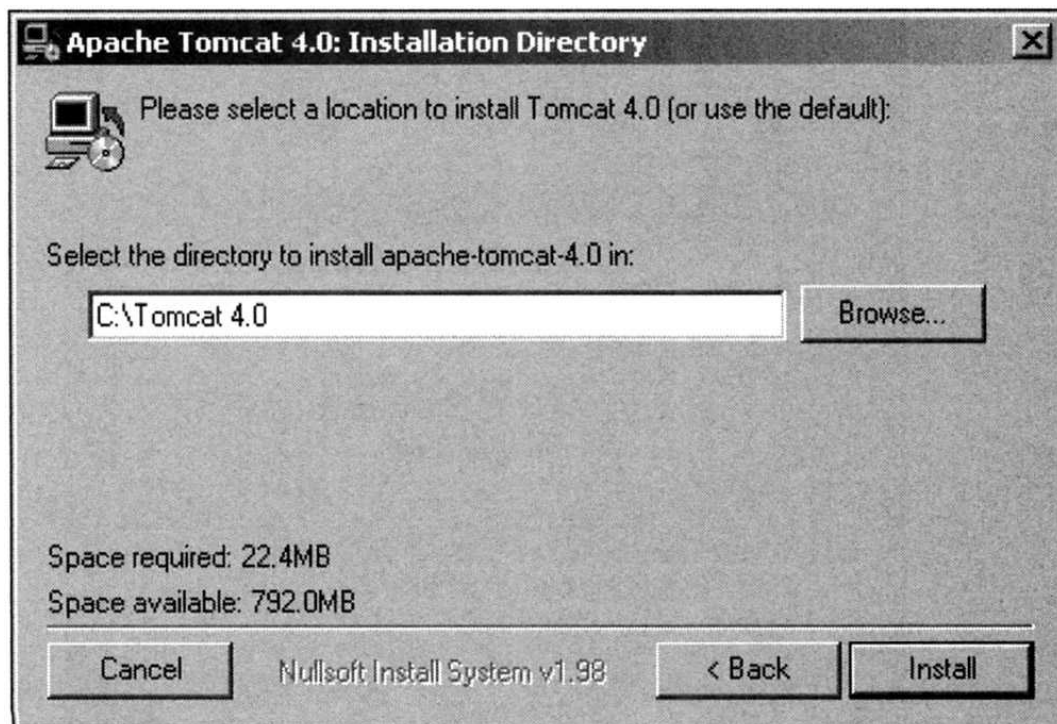


For Windows platform, the easiest method is to download and run the `jakarta-tomcat-4.0.4.exe` file.

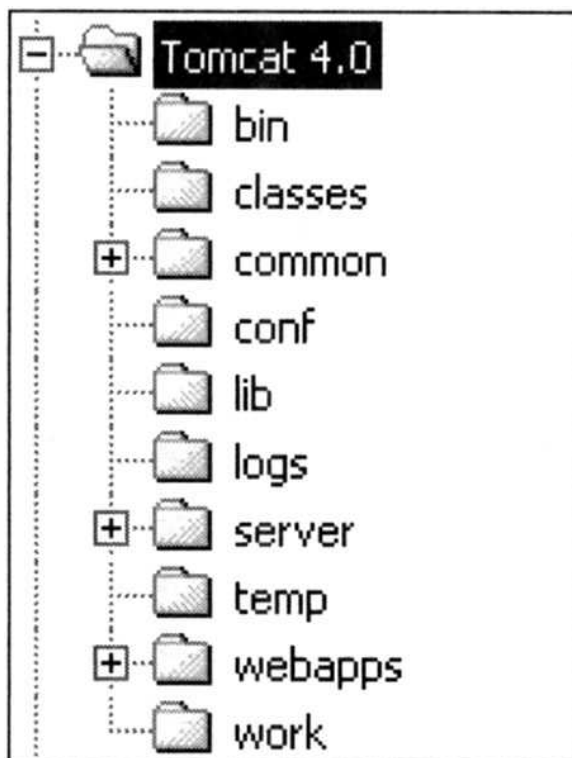
2. Run the executable. After we've agreed to the license, the `jakarta-tomcat-4.0.4` displays the following dialog box:



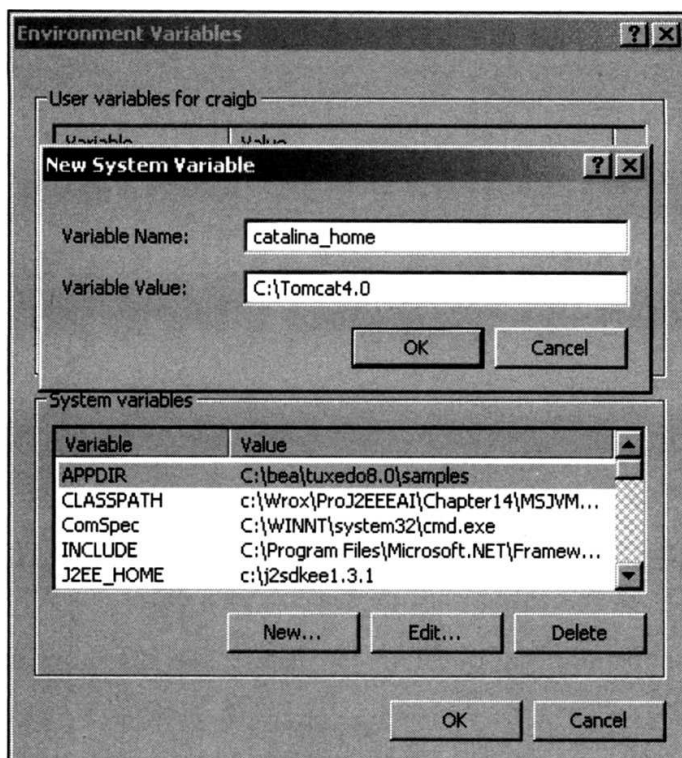
3. Select the optional components that you might need. Note that in this book we will not use the source code or the NT service.
4. The next screen prompts for the installation directory. We will use C:\Tomcat 4.0 but any other directory will do just fine:



5. At the end of the installation, our Tomcat installation directory should look like this:

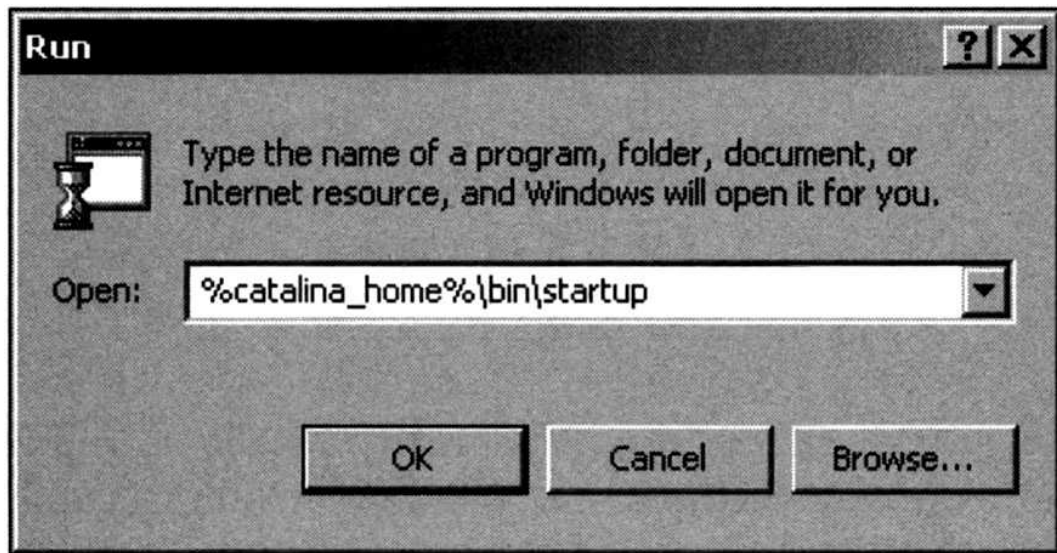


6. We will define a `catalina_home` environment variable to represent the Tomcat installation directory (in our case `C:\Tomcat4`). In Windows 2000 this can be done through the `Environmental Variables` tab in the `System control panel` applet:

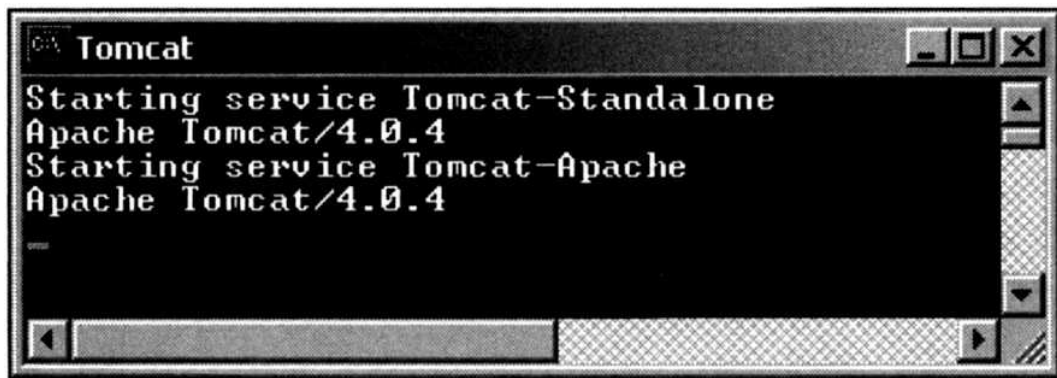


Note Catalina was the code name for the Tomcat 4.0 development. It is still used in several places, including the `.sh` and `.bat` files that come with the installation.

7. To test the installation, simply run `%catalina_home%\bin\startup:`

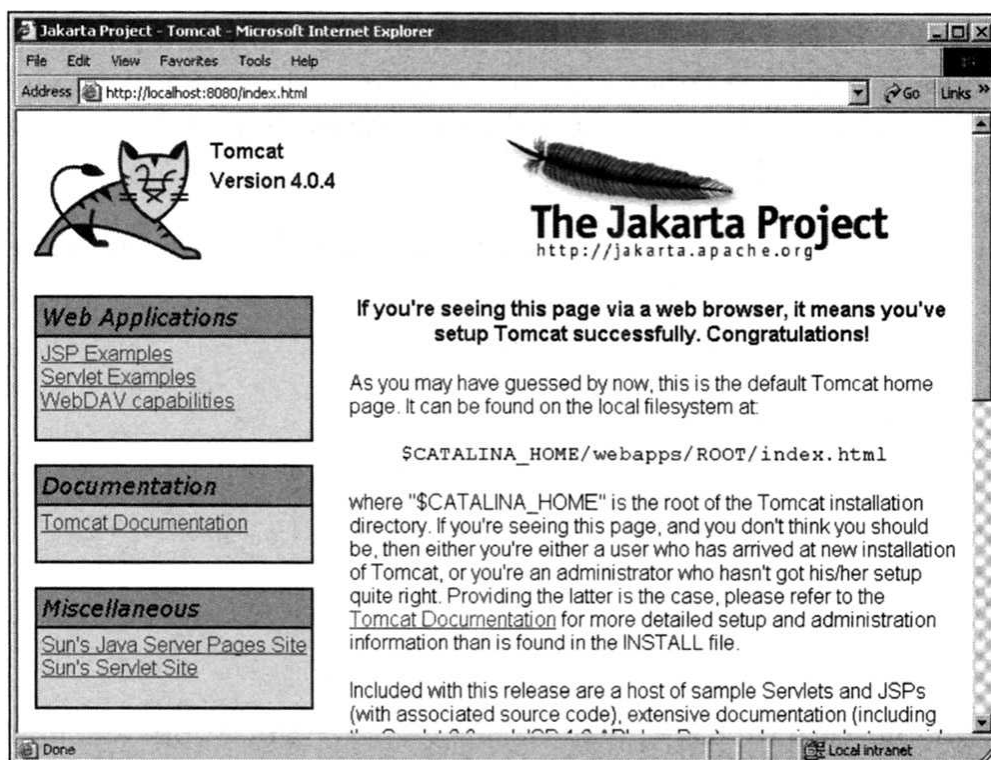


This launches the Tomcat server in a new window that looks like the following:



Note If you would rather run Tomcat from within the same window (perhaps to see any exceptions thrown) then the command is %
catalina_home%\bin\catalina run.

8. At this point, our Tomcat server is listening to port 8080 for HTTP requests (providing you have stopped our SimpleAxisServer listening on 8080). By navigating to the URL <http://localhost:8080>, we can view its home page:



Now that we have Tomcat running correctly, we can focus our attention back to SOAP development and modify our Tomcat installation to run with Axis.

Try It Out: Modify Tomcat to Work with Axis

We need to modify Tomcat's default installation to make it suitable for running Axis inside Tomcat. We must:

- Alter the classpath to include the Axis JAR files we mentioned previously
 - Install the Axis servlet
1. There are several ways to modify the classpath used by Tomcat. A straightforward way is to edit the %catalina_home%\bin\setclasspath.bat file using a text editor, like so:

```
rem Set standard CLASSPATH
rem Note that there are no quotes as we do not want to introduce random
rem quotes into the CLASSPATH
set CLASSPATH=%JAVA_HOME%\lib\tools.jar
set CLASSPATH=%classpath%;%axisDirectory%\lib\axis.jar
set CLASSPATH=%classpath%;%axisDirectory%\lib\commons-logging.jar
set CLASSPATH=%classpath%;%axisDirectory%\lib\jaxrpc.jar
set CLASSPATH=%classpath%;%axisDirectory%\lib\log4j-core.jar
set CLASSPATH=%classpath%;%axisDirectory%\lib\tt-bytecode.jar
set CLASSPATH=%classpath%;%axisDirectory%\lib\saaaj.jar

set CLASSPATH=%classpath%;%jwsDirectory%\lib\xercesImpl.jar
set CLASSPATH=%classpath%;%jwsDirectory%\lib\xmlParserAPIs.jar

rem Set standard command for invoking Java.
```

2. The easiest way to install the Axis servlet in Tomcat is to copy the %axisDirectory%\webapps\axis directory into the %catalina_home%\webapps\ROOT directory.

Another solution that has the advantage of avoiding duplicating the webapps\axis directory is to edit the %catalina_home%\conf\server.xml and add a **context** to Tomcat for Axis. A context is essentially the definition of a web application in Tomcat. In our case, we will be following the second approach.

Here is the modification needed in the server.xml file. Edit the file using any text editor:

```
<name>mail.smtp.host</name>
<value>localhost</value>
</parameter>
</ResourceParams>
```



```

        </Context>
        <Context docBase="c:\xml-axis\webapps\axis" path="/axis"/>
    </Host>
</Engine>
</Service>

```

Note that in our case we have given `c:\xml-axis\webapps\axis` as the `docBase`, since this is the path for our `%axisDirectory%\webapps\axis` directory (recall we installed Axis in the directory `C:\xml-axis`).

3. Stop Tomcat with the following command:

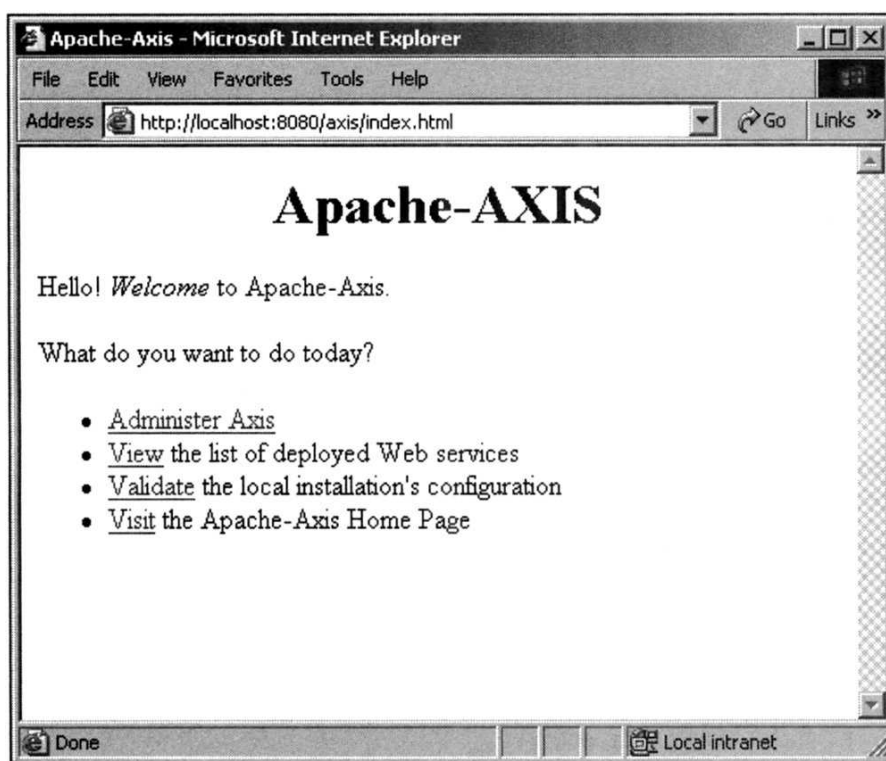
```
%catalina_home%\bin\shutdown
```

4. After Tomcat has stopped, restart it by typing this command in order to load the Axis web application:

```
%catalina_home%\bin\startup
```

Important If you get classpath errors now when you try to start Tomcat, then copy the `axis.jar` file into Tomcat's `\common\lib` directory and then remove the relevant `set classpath` line in the batch file.

5. Once Tomcat restarts, we should be able to navigate to the Axis home page; the URL is `http://localhost:8080/axis/index.html`. If the modifications were carried out correctly, our browser will display the Axis home page:



The `Administer Axis` link provides a way to start and stop Axis. When the Axis server is stopped, any request returns a SOAP fault. The `View` link provides a list of the registered web services (more on this later).

Note Note that when it comes to the installation and configuration of Tomcat, we have barely scratched the surface. Among other things, we have not described how to set up Tomcat with a production web server like Apache or IIS, but the instructions that we have reviewed in this section are enough to start our development efforts. For detailed documentation on how to set up Tomcat in a production environment, you can visit the <http://jakarta.apache.org/tomcat/index.html>.

We are now ready to deploy the `HelloWorld` service inside Tomcat.

Try It Out: HelloWorld (Reprise)

The method of using a Java Web Service (JWS) file still works with Axis running as a servlet hosted by Tomcat. We will, however, need to make a small change in the `HelloWorldClient.java` file.

1. First we need to modify the `HelloWorldClient.java` file. Since Axis is now running as the Axis web application inside Tomcat, the `HelloWorld` service changes to `http://localhost:8080/axis/HelloWorld.jws` (the earlier URL was `http://localhost:8080/HelloWorld.jws`). So open the `HelloWorldClient.java` file in a text editor and make the following change to it:

```
public static void main (String args[]) {
```

```

System.out.println("HelloWorldClient.main: Entering...");
try {
    String url = "http://localhost:8080/axis/HelloWorld.jws";
    String sender = "Reader";
    Service service = new Service();
    Call call = (Call) service.createCall();
    ...
}

```

The `HelloWorld.jws` file remains unchanged.

- Assuming that `HelloWorld` is the current directory, the following command will recompile the client:

```
javac HelloWorldClient.java
```

- Copy the `HelloWorld.jws` file to the `%axisDirectory%\webapps\axis` directory.

- The command to run the client is identical to the `SimpleAxisServer` example:

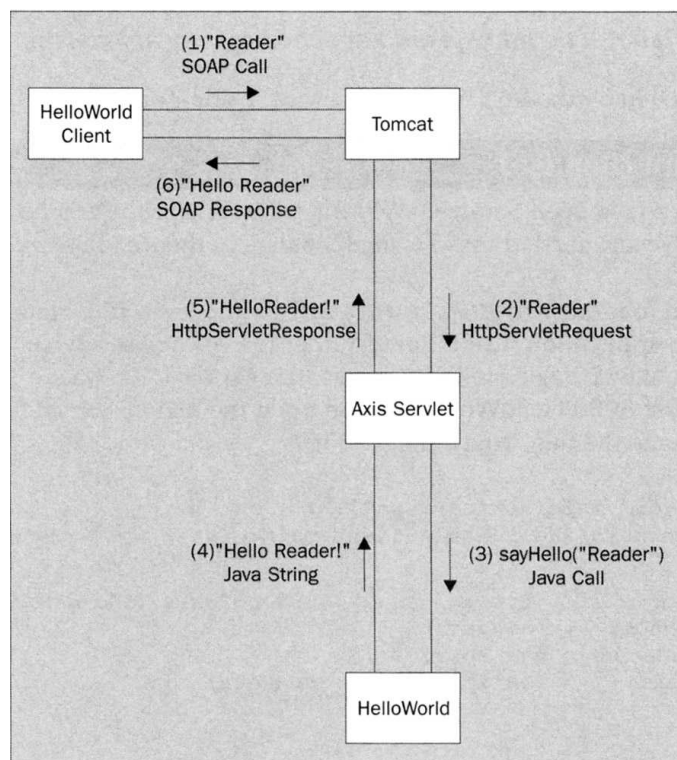
```
java HelloWorldClient
```

We should get exactly the same exciting result as before.

Before trying out this example make sure that Tomcat is running and that the `SimpleAxisServer` is stopped. Note that the exceptions thrown by the (Tomcat) server are logged in the file `localhost.log.date`, where `date` represents the date on which the log file was created. This file is in the `%catalina_home%\logs\` directory.

How It Works

The main difference between the `SimpleAxisServer` and running Axis inside Tomcat is that Tomcat handles the HTTP requests and passes them as servlet requests to Axis. For more information on the servlet methodology, please refer to <http://java.sun.com/products/servlet>. This situation is depicted in the high-level diagram below:



As we can see above, the request from the client goes to Tomcat on port 8080. Tomcat then creates an instance of the `AxisServlet` class, and calls it with the SOAP call repackaged in a servlet-specific data structure: `HttpServletRequest`. The `AxisServlet` parses the requests and transforms it into a Java call for our web service: `HelloWorld`. On the return side, the `AxisServlet` generates a SOAP response from the Java return value and puts it inside another servlet-specific data structure, an `HttpServletResponse`. Finally, Tomcat uses that data to create the HTTP response sent back to the client.

Using JWS files might not be the best deployment strategy for a production web site. Some of the arguments against using JWS are:

- The source code of our web service is available to everybody

- The entire source must be in the JWS files, or the JAR files must be manually copied to the `lib` directory
- The amount of configuration required is limited (see below for more details)

However, like the `SimpleAxisServer`, the use of JWS files is quick and easy.

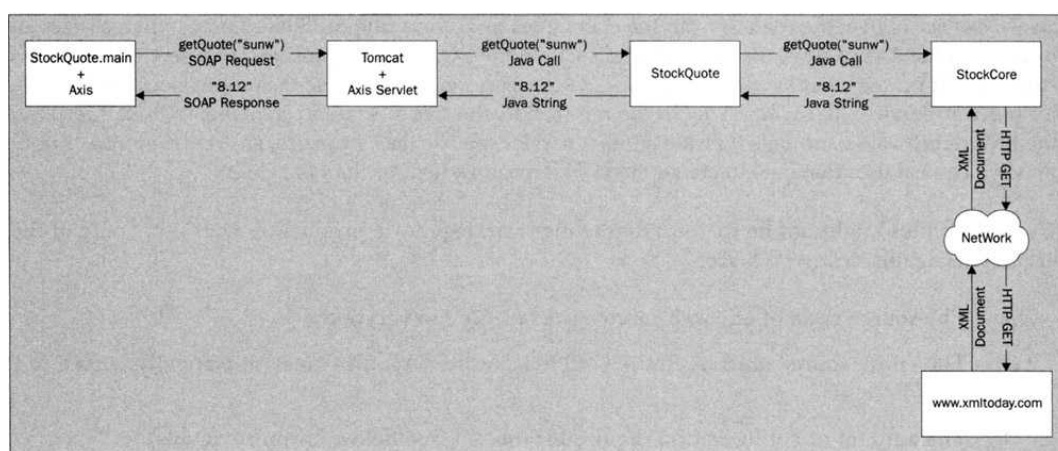
There is an alternative to JWS-based web services. Instead of using a JWS file we can use **compiled Java classes** and **deployment descriptors**. This form of deployment is also called **manual or custom deployment**.

Manual Deployment – StockQuote

To explain the process of manual deployment better, we need to take a more complex example than `HelloWorld`. We will work with a stock quote service that provides web service-based stock quotes over the Internet. To show a realistic scenario that involves the use of class libraries, we will divide the web service into two classes:

- The `StockCore` class, which deals with the details of getting the stock quote. This was developed in Chapter 2.
- The `StockQuote` class, which exposes the functionality as a web service.

The `StockQuote` class contains one method that defines the interface of the web service and one `main()` method that acts as the client to the web service. The following diagram shows the organization of the application:



The figure above shows that a request to `StockQuote` is made of the following steps:

1. The client code builds a SOAP request with the help of the Axis runtime similarly to how we did with the `HelloWorld.main()`.
2. The Axis runtime sends the SOAP request over HTTP to Tomcat, which then invokes the Axis servlet. The Axis servlet then **de-serializes** the SOAP request, in other words, the Axis servlet builds a Java call for the `StockQuote` class (`StockQuote.getQuote("SUNW")`). We will revisit the issue of **serialization** (Java to XML) and **de-serialization** (XML to Java) at the end of this chapter.
3. The `StockQuote.getQuote()` method gets the actual value of the stock using the `StockCore` class that we build in Chapter 2.
4. Upon the return of the method, Axis serializes the stock quote into a SOAP response.
5. On the client side, Axis **de-serializes** the SOAP response into a Java string that can easily be used by the client code.

We will now compile, deploy, and test the `StockQuote` web service. But first, we'll need to use the functionality we developed in the `StockCore` example in the last chapter as a JAR file. We'll begin our next example by turning `StockCore.class` into `stockcore.jar` in order to use it as a library for the rest of the example.

Try It Out: StockQuote

In order to build our stock quote service, the first steps will involve making our `stockquote.jar` file available to the stockquote service. To manually build the `stockcore.jar` file, follow these simple steps:

1. From the `\Chp02\StockCore\classes\` directory run the following command (assuming you have a compiled version of `StockCore.class` in the `com\wrox\jws\stockcore` directory):

```
jar -cf stockcore.jar com\wrox\jws\stockcore\StockCore.class
```
2. Place `stockcore.jar` in the `%axisDirectory%\webapps\axis\WEB-INF\lib\` path so that we have it in our classpath.
3. Here is the code for `StockQuote.java` (to be placed in the `\Chp03\StockQuote\src` directory):

```
package com.wrox.jws.stockquote;
```

```

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;

import javax.xml.rpc.ParameterMode;
import javax.xml.namespace.QName;

import com.wrox.jws.stockcore.StockCore;

public class StockQuote {

    public String getQuote(String ticker) throws Exception {
        StockCore stockcore = new StockCore();
        return stockcore.getQuote(ticker);
    }
}

```

The `StockQuote.main()` method is used as a client for testing:

```

public static void main (String [] args) {
    final String methodName = "StockQuote.main";

    try {
        if(args.length != 1) {
            System.err.println("StockQuote Client");
            System.err.println(
                "Usage: java com.wrox.jws.stockquote.StockQuote"+
                "<ticker-symbol>");
            System.err.println(
                "Example: java com.wrox.jws.stockquote.StockQuote sunw");
            System.exit(1) ;
        }

        // Replace the following URL with what is suitable for
        // your environment
        String endpointURL = "http://localhost:8080/axis/servlet/AxisServlet";
        Service service    = new Service();
        Call    call       = (Call) service.createCall();

        call.setTargetEndpointAddress(new java.net.URL(endpointURL));
        call.setOperationName(new QName("StockQuote", "getQuote"));
        call.addParameter("ticker", XMLType.XSD_STRING, ParameterMode.IN);
        call.setReturnType(org.apache.axis.encoding.XMLType.XSD_STRING);
        String ret = (String) call.invoke(new Object[] { args[0] });

        System.out.println("The value of" + args[0] + "is:" + ret);

    } catch(Exception exception) {
        System.err.println(methodName + ": " + exception.toString());
        exception.printStackTrace();
    }
}
}

```

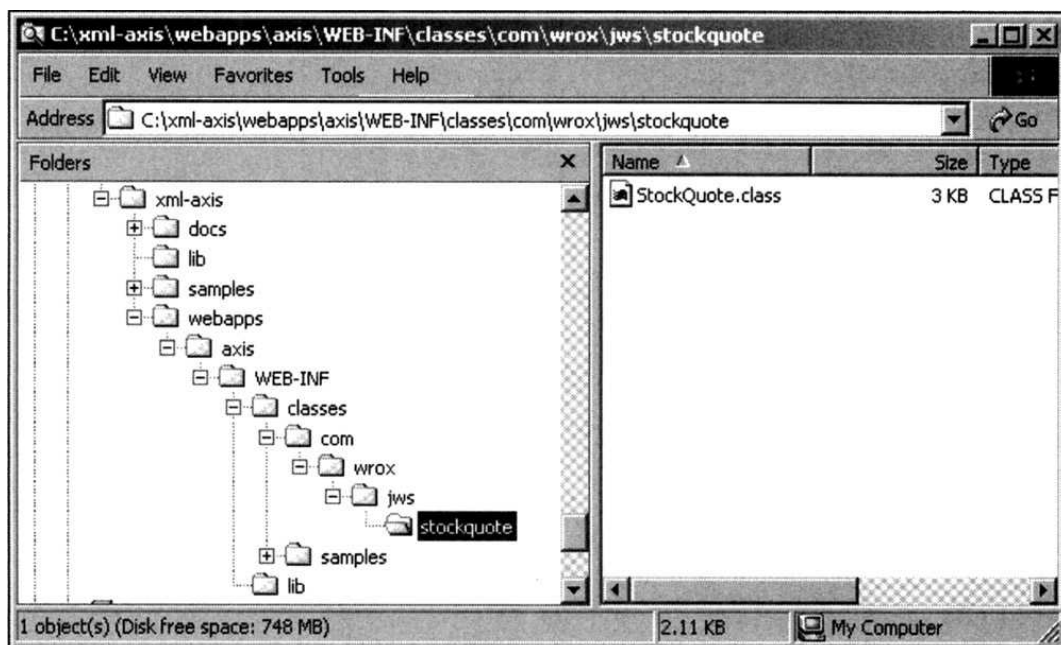
4. To compile `StockQuote` class, type in the following commands (we need the `StockCore` class in our classpath so we reference the copy we stored in Axis):

```

javac -classpath %classpath;%axisDirectory%\webapps\axis\
      WEB-INF\lib\stockcore.jar
      -d ..\classes StockQuote.java

```

5. Once the class file is compiled, we need to copy it into the `\classes` directory of the Axis web application. Depending which methodology you elected to use to modify Tomcat to run with Axis (see *Try It Out - Modify Tomcat to Work with Axis* earlier in this chapter), you will either copy the compiled file to the `\webapps` directory inside Axis or to the `\axis` directory inside Tomcat. We mapped the Axis application to our Axis installation directory so, here is what our `\axis` directory should look like:



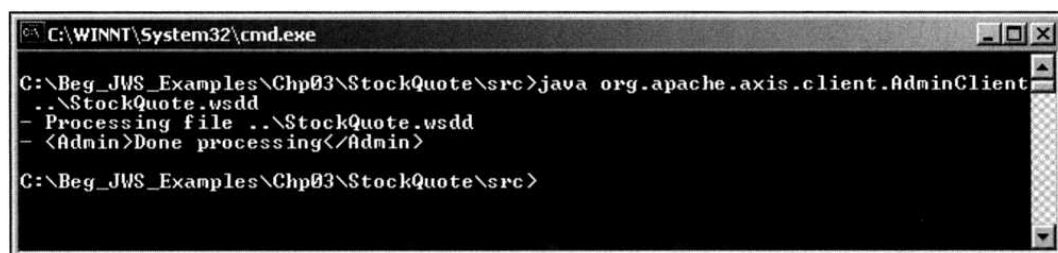
6. Now we need to tell Axis that `StockQuote` is a valid web service. This is done with the help of a deployment descriptor and the Axis `AdminClient` (make sure that Tomcat is running before launching these commands).

Save this deployment descriptor into a file called `StockQuote.wsdd`:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java=http://xml.apache.org/axis/wsdd/providers/java">
<service name="StockQuote" provider="java:RPC">
  <parameter name="className" value="com.wrox.jws.stockquote.StockQuote"/>
  <parameter name="allowedMethods" value="getQuote"/>
</service>
</deployment>
```

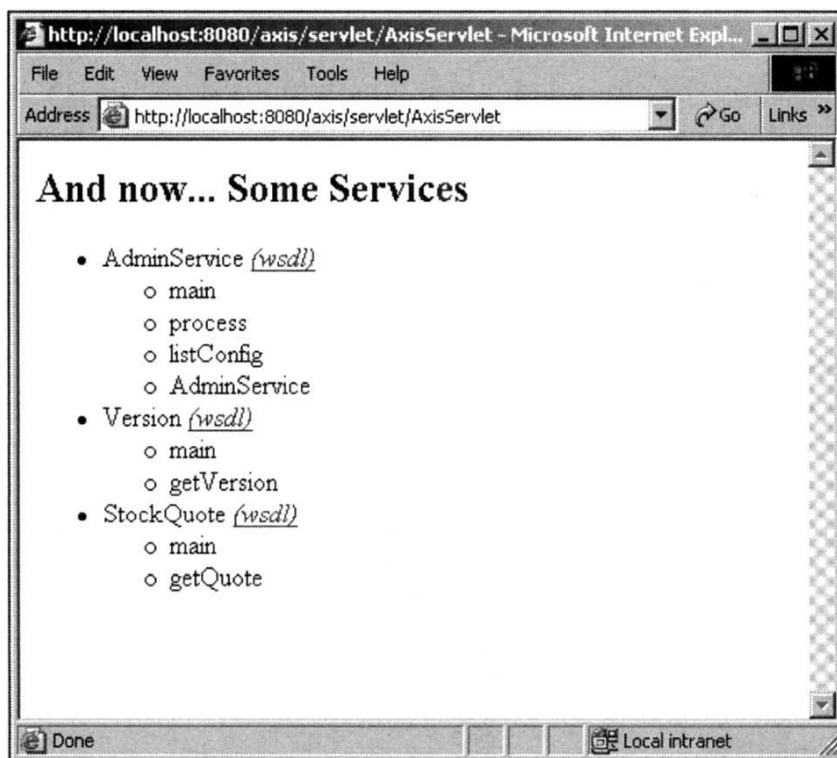
7. The following command sends the deployment descriptor to Axis for processing (assuming that `\src` is still our current directory):
- ```
java org.apache.axis.client.AdminClient ..\StockQuote.wsdd
```

Here is how it look on our client machine:



**Note** Bear in mind that Tomcat should not be running when we are building the example. It might give some errors as the `Stockcore.jar` and `StockQuote.class` files are likely to be locked by Tomcat. However, when trying out the example, make sure that Tomcat is running.

8. We will examine the deployment descriptor in detail shortly. For now, we will check whether the `StockQuote` service has been successfully deployed. For this we will visit the URL `http://localhost:8080/axis/servlet/AxisServlet`:

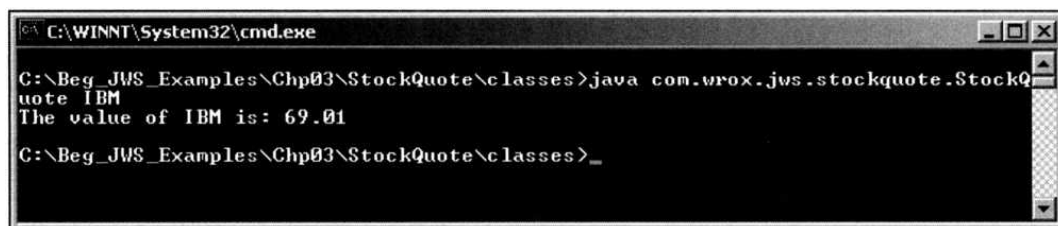


You can see that there is now a web service called `StockQuote` with two methods: `main` and `getQuote` that correspond to our class.

9. Copy the `stock_quote.xml` file to the `%axisDirectory%\webapps\axis\WEB-INF\classes` directory; otherwise we'll get an exception when `StockCore` can't load the source **XML** document.
10. Now we can test our `StockQuote` class. As you will recall we built a test harness into the `StockQuote` class itself in the `main()` method. Therefore, we can run it either on the class we copied to Axis (from the `%axisDirectory%\webapps\axis\WEB-INF\classes` directory) or our original compiled class (from the `\Chp03\StockQuote\classes` directory):

```
java com.wrox.jws.stockquote.StockQuote IBM
```

All being well, this should return the following result:



## How It Works

Before jumping into the code for `StockQuote`, it is worth spending a few moments to review what we achieved and compare it to the **JWS**-based implementation.

Our first step was to compile the `StockQuote.java` file. This step solved the first problem we mentioned with **JWS**; our source code is no longer accessible to everybody.

The second step was to copy the compiled class along with its dependency (`stockcore.jar`). In fact, nothing prevents us from installing a complete web application with libraries and resource files. This brings up an important point: our application does not have to be contained inside the Axis web application. We are free to define our own web application completely separated from Axis. The security implications of this simple fact are significant since this allows our application to run completely isolated from other web services deployed on the same machine.

The third and last step before running the client was the actual deployment. The `AdminClient` can easily deploy or un-deploy a web service. All this sounds a little too easy, but there is a catch. What is there to prevent a malicious client from un-deploying all our web services from a server?

The Axis designers are aware of this potential security breach, so by default **remote administration** is not allowed on Axis. In other words, we

are only permitted to deploy and un-deploy web services from the local machine. However it is possible to allow remote administration by modifying the `\axis\WEB-INF\server-config.wsdd` file setting the `enableRemoteAdmin` attribute to `true` instead of `false`:

```
...
<handler name="MsgDispatcher">
 type="java:org.apache.axis.providers.java.MsgProvider"/>
 <service name="AdminService" provider="java:MSG">
 <parameter name="allowedMethods" value="AdminService"/>
 <parameter name="enableRemoteAdmin" value="true"/>
 <parameter name="className" value="org.apache.axis.utils.Admin"/>
 <namespace>http://xml.apache.org/axis/wsdd/</namespace>
 </service>
</service name="Version" provider="java:RPC">
...

```

If you plan on using remote administration in your deployed web services, be sure to secure access to the web site.

Let's now focus our attention to the `StockQuote.java` file (the full source code can be downloaded from our web site). The imports of the `org.apache.axis` and `javax.xml` packages are identical to what we saw in `HelloWorld`. Once again they are only required for the client portion of this sample (contained in the `main()` method). The import of `com.wrox.jws.stockcore.StockCore` gives us access to the `StockCore` class.

Most of the server-side work is done in the `getQuote()` method:

```
public class StockQuote {

 public String getQuote(String ticker) throws Exception {
 StockCore stockcore = new StockCore();
 return stockcore.getQuote(ticker);
 }
}

```

As we can see from the above code snippet, the method `getQuote()` takes a string as input (the ticker symbol of the stock we are interested in) and returns the current stock value as a string.

The class also contains a `main()` method to test the service; this is very similar to the `HelloWorld.main()` method, which we reviewed previously. A notable difference is in the invocation of the `StockQuote` service, which is shown below:

```
...
call.setReturnType(org.apache.axis.encoding.XMLType.XSD_STRING);

String ret = (String)call.invoke(new Object[] { args [0] });

System.out.println("The(delayed) value of " + args[0]
+ " is: " + ret);
...

```

Armed with the implementation of our web service, we can now focus on the deployment descriptor: `StockQuote.wsdd`. The outermost element, `<deployment/>`, tells us that the XML document is an Axis deployment descriptor. The deployment descriptor is Axis specific since the namespace is `"http://xml.apache.org/axis/wsdd/"`. The Java namespace is used for Java providers, or in other words, web services implemented as a Java class.

For each web service that we deploy, we will have one `<service/>` element in the deployment descriptor. In our example, we deploy the `StockQuote` service as a Java class and we use the Remote Procedure Call (RPC) methodology. The meanings of the `classname` and `allowedMethods` attributes are self-explanatory. Instead of listing methods (separated by commas or spaces), we can specify `"*"` to allow all the methods to be invoked.

It is worth mentioning that Axis instantiates the class that we specify using the default constructor, hence the requirement that the default constructor (for example, `StockQuote()`) must be defined and public. Without further instructions in the deployment descriptor Axis instantiates a new object with every request. Since creating an object with every request can be wasteful at times, Axis supports three modes of object creation, also referred as **scoping**:

- **Request**  
This is the default: a new object is created with every request to the server
- **Session**  
The object will exist for the duration of the session
- **Application**  
One object will be created for the entire application

To specify different scoping, simply set the `scoping` attribute to the desired value in the `<service/>` element:



```

...
<parameter name="allowedMethods" value="getQuote"/>
<parameter name="scope" value="session"/>
</service>
</deployment>

```

When deciding on the scope of a web service it is important to keep two things in mind:

#### ■ Threading

When using the request scope, our web service object will be created and used in the same thread. It is not necessarily the case with the session and application scopes; the `SimpleAxisServer` is single-threaded, but Tomcat and other production-grade servers are multi-threaded.

#### ■ Security

When using the application scope, all clients will use the same object and any residual state from one request might affect another request. In addition to the added complexity of developing thread-safe web services, this is clearly a security risk, which we must consider during our design and balance it against the benefits of sharing object instances across client requests.

This concludes our description of Tomcat as a container for Axis. Before concluding this chapter, we need to discuss an important topic: **serialization**.

## Market Example

**Note** The code for the Market class can be found in the `\Chp03\Market` directory.

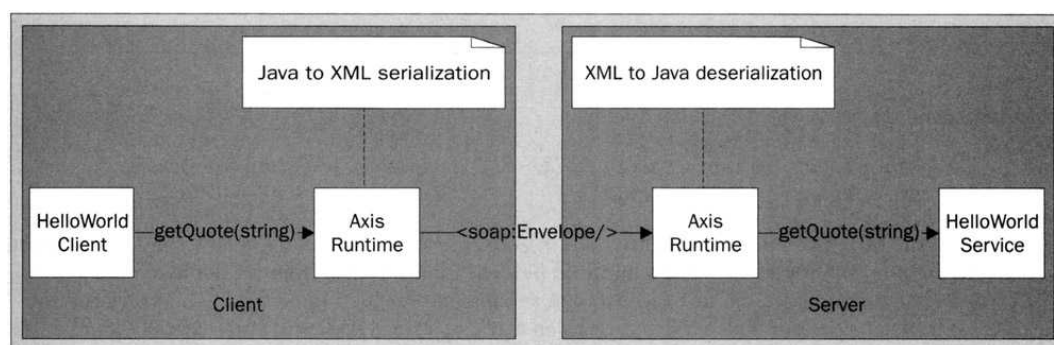
We will use the `Market` class to show how data types are serialized and deserialized by the Axis framework. The `Market` class is a modified version of the `StockQuote` class. It returns more information through the `MarketData` class. The `MarketData` class contains the following instance data:

- The ticker symbol
- The ticker symbol value as a string
- The ticker symbol value as a double
- The values of the three major US stock indices (DOW, NASDAQ and S&P 500)

Before looking at the `Market` class in action, we need to discuss some background information on the mappings between Java and XML.

## Java/XML Mappings

We have seen that `HelloWorld.main()` uses Axis to serialize Java data types into XML data types and that the Axis server deserializes the XML data types into Java data types. That serialization/deserialization process for a request is briefly summarized in the following diagram:



The only data type exchanged between the client and the server in the case of `HelloWorld` is a string. On the calling side, the string argument of `HelloWorld.getQuote()` is serialized as follows:

```
<arg0 xis:type="xsd:string">Reader</arg0>
```

Note that the return string value is serialized in the same way.

The following table describes how Axis implements the XML to Java mapping for simple data types:

| Simple Type | Java Type        |
|-------------|------------------|
| xsd:string  | java.lang.String |

|                  |                      |
|------------------|----------------------|
| xsd:integer      | java.math.BigInteger |
| xsd:int          | int                  |
| xsd:long         | long                 |
| xsd:short        | short                |
| xsd:decimal      | java.math.BigDecimal |
| xsd:float        | float                |
| xsd:double       | double               |
| xsd:Boolean      | boolean              |
| xsd:byte         | byte                 |
| xsd:dateTime     | java.util.Calendar   |
| xsd:base64Binary | byte[]               |
| xsd:hexBinary    | byte[]               |

We can combine these simple types to form arrays and complex data structures (we will see an example shortly).

**Note** The specific mapping rules are defined in the JAX-RPC specification, which can be downloaded from <http://java.sun.com/xml/jaxrpc/>.

Thanks to the serialization infrastructure built in Axis, most times we don't need to pay much attention to these mappings. Simple types are a hundred percent transparent. When we deployed the `StockQuote` example we did not specify what serialization needed to be used. The same goes for arrays; for instance, if our web service returns an array of strings Axis will automatically handle the serialization and deserialization without further instructions from the web service developer or consumer. The mapping for complex types is almost handled automatically. We must specify how to handle the serialization but the default mechanism provided by Axis will take care of the details for us. The serialization information is part of the deployment descriptor.

Let's see how this works in practice with the `Market` class.

### Try It Out: The Market and MarketData Classes

The market example contains two classes: the `Market` class is the web service, and the `MarketData` class gets the data returned by `Market.getQuote()`. Our first step is to compile these two classes.

1. The `MarketData` class contains a stock quote with the market indices. Save this code in a file called `MarketData.java` in the `\Chp03\Market\src` directory):

```
package com.wrox.jws.stockquote;

import com.wrox.jws.stockcore.StockCore;
public final class MarketData {

 public transient String DOW = "^DJI";
 public transient String NASDAQ = "^IXIC";
 public transient String SP500 = "^GSPC";

 String ticker; // The ticker symbol.
 String stringValue; // The value of the stock as a string
 double doubleValue; // The value of the stock as a double
 double indices []; // The DOW, the NASDAQ, and the S&P 500 values

 /**
 * Empty Market constructor. For the bean serializer.
 */
 public MarketData() {}

 /**
 * Market constructor. Simply pass in the ticker symbol and the
 * constructor will do the rest.
 *
 * @param ticker The ticker symbols to get a quote for (e.g. sunw)
 * @throws exception If anything goes wrong
 */
 public MarketData(String ticker) throws Exception {
 this.ticker = ticker;

 StockCore stockcore = new StockCore();

 stringValue = stockcore.getQuote(ticker);
```

```

 doubleValue = Double.parseDouble(stringValue);
 indices = new double [3];
 indices [0] = Double.parseDouble(stockcore.getQuote(DOW));
 indices [1] = Double.parseDouble(stockcore.getQuote(NASDAQ));
 indices [2] = Double.parseDouble(stockcore.getQuote(SP500));
 }

 /**
 *Public get and set methods. Used by the bean serializer
 */
 public double [] getIndices() {
 return indices;
 }
 public void setIndices(double indices []) {
 this.indices = indices;
 }
 public String getTicker() {
 return ticker;
 }
 public void setTicker(String ticker) {
 this.ticker = ticker;
 }
 public double getDoubleValue() {
 return doubleValue;
 }
 public void setDoubleValue(double doubleValue) {
 this.doubleValue = doubleValue;
 }
}

```

2. The Market class contains the web service and a test client in the Market.main(). Save this code in a file called Market.java in a \Chp03\Market\src directory):

```

package com.wrox.jws.stockquote;

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;
import org.apache.axis.encoding.ser.BeanSerializerFactory;
import org.apache.axis.encoding.ser.BeanDeserializerFactory;

import javax.xml.rpc.ParameterMode;
import javax.xml.namespace.QName;

public class Market {

 /**
 * Returns a delayed quote for the ticker symbol passed in. This one line
 * method is the only method of the service thanks to the simplicity of
 * Axis and the use of our StockCore class that we developed in Chapter 2.
 *
 * @param ticker The ticker symbol of the the stock
 */
 public MarketData getQuote(String ticker) throws Exception {
 return new MarketData(ticker);
 }
}

```

The main() method is used as a client of the StockQuote SOAP service. This method is used for testing purposes and as such is not part of the StockQuote web service. In a production mode, it is usually preferable to isolate test methods like this one in a separate test class that is not included in the web service distribution:

```

/**
 * @param args The ticker symbols to get a quote for (e.g. sunw)
 */
public static void main (String [] args) {
 final String methodName ="Market.main";

 try {
 if(args.length != 1) {
 System.err.println("Market Client");

```

```

 System.err.println(
 "Usage: java com.wrox.jws.stockquote.Market <ticker-symbol>");
 System.err.println(
 "Example: java com.wrox.jws.stockquote.Market sunw");
 System.exit(1);
 }

 // Replace the following URL with what is suitable for your
 // environment
 String endpointURL =
 "http://localhost:8080/axis/servlet/AxisServlet";
 Service service = new Service();
 Call call = (Call) service.createCall();

 // Setup input arguments
 call.setTargetEndpointAddress(new java.net.URL(endpointURL));
 call.setOperationName(new QName("Market", "getQuote"));
 call.addParameter("ticker", XMLType.XSD_STRING, ParameterMode.IN);

 // register the MarketData class
 QName qName = new QName("http://stockquote.jws.wrox.com",
 "MarketData");
 call.registerTypeMapping(MarketData.class, qName,
 BeanSerializerFactory.class, BeanDeserializerFactory.class);

 call.setReturnType(new QName("marketData"));

 MarketData data = (MarketData) call.invoke(new Object [] { args [0] });

 System.out.println("The(delayed) value of " + args [0]
 + "is: " + data.doubleValue + "(NASDAQ is "
 + data.indices [1] + ")");
} catch(Exception exception) {
 System.err.println(methodName + ": " + exception.toString());
 exception.printStackTrace();
}
}
}

```

3. Compiling Market and MarketData is similar to the StockQuote example. Assuming that the current directory is src, the following instructions will compile and run the example on the Windows platform. Be sure to include all the JAR files that we used earlier (including StockQuote.jar) in the classpath:

```

javac -d ..\classes com\wrox\jws\stockquote\MarketData.java
javac -d ..\classes com\wrox\jws\stockquote\Market.java

```

4. Once we have compiled these two classes, we will need to copy them into the Axis web application directory (we used Tomcat 4, but you could also use the SimpleAxisServer).
5. To deploy the Market web service manually we need the following MarketDeploy.wsdd file:

```

<deployment xmlns="http://xml.apache.org/axis/wsdd/"
 xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
 <service name="Market" provider="java:RPC">
 <parameter name="className" value="com.wrox.jws.stockquote.Market"/>
 <parameter name="allowedMethods" value="getQuote"/>
 </service>
 <beanMapping qname="ns:MarketData"
 xmlns:ns="http://stockquote.jws.wrox.com"
 languageSpecificType="java:com.wrox.jws.stockquote.MarketData"/>
</deployment>

```

6. Make sure that Tomcat (or the SimpleAxisServer) is running and listening on port 8080, and run the AdminClient to deploy the Market service as shown below (assuming that src is still your current directory):

```

java org.apache.axis.client.AdminClient ..\MarketDeploy.wsdd

```

Here is how it should look:

```

C:\WINNT\System32\cmd.exe

C:\Beg_JWS_Examples\Chp03\Market\src>java org.apache.axis.client.AdminClient ..\
MarketDeploy.wsdd
- Processing file ..\MarketDeploy.wsdd
- <Admin>Done processing</Admin>

C:\Beg_JWS_Examples\Chp03\Market\src>_

```

7. Our last step is to run the Market client with the following command (from either the Chp03\Market\classes directory or the %axisDirectory%\webapps\axis\WEBINF\classes directory):

```
java com.wrox.jws.stockquote.Market SUNW
```

8. The output should be similar to the following screenshot:

```

C:\WINNT\System32\cmd.exe

C:\Beg_JWS_Examples\Chp03\Market\classes>java com.wrox.jws.stockquote.Market SUNW
W
The value of SUNW is: 5.93 <NASDAQ is 1360.62>

C:\Beg_JWS_Examples\Chp03\Market\classes>_

```

## How It Works

Let's have a closer look at the response as it contains the serialization of the MarketData class:

```

(HTTP Header omitted from this listing)
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
 <ns1:getQuoteResponse
 SOAP-ENV:encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:ns1="Market">
 <getQuoteReturn href="#id0"/>
 </ns1:getQuoteResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

The first part of the listing shows the SOAP header for an RPC return value; it is similar to what we saw in `StockQuote`. The return value is actually a reference to another element in the file, the `href="#id0"` attribute indicates that the element to look for is identified with the `id="id0"` attribute (see below):

```

</ns1:getQuoteResponse>
<multiRef id="id0"
 SOAP-ENC:root="0"
 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xsi:type="ns3:MarketData"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:ns2=
 "http://schemas.xmlsoap.org/soap/envelope/:encodingStyle"
 xmlns:ns3="http://stockquote.jws.wrox.com">

```

The `<multiRef/>` element is a placeholder for data that might be referenced multiple times. In this case, however, it is only referenced once in `<getQuoteReturn/>`.

Inside the `<multiRef/>` element, we will find the actual data of the `MarketData` object. The first instance data to get serialized is the array of index values (DOW, NASDAQ, S&P500). Notice that the type of each element is double and that the array encoding is defined as SOAP encoding ("`http://schemas.xmlsoap.org/soap/encoding/`"). Also, the name of the element is the name of our instance data (`indices`):

```

<indices xsi:type="SOAP-ENC:Array"
 SOAP-ENC:arrayType="xsd:double[3]">
 <item>9925.25</item>
 <item>1615.73</item>
 <item>1067.14</item>
</indices>

```



The remaining instance data items are the ticker symbol (string), its value as a double, and the three constants for US market indices. If we want to avoid the serialization of these constants by Axis, we need to remove them from the class definition (marking them as `transient` does not work as of Axis Beta 2):

```
<ticker xsi:type="xsd:string">sunw</ticker>
<doubleValue xsi:type="xsd:double">6.89</doubleValue>
<DOW xsi:type="xsd:string">^DJI</DOW>
<NASDAQ xsi:type="xsd:string">^IXIC</NASDAQ>
<SP500 xsi:type="xsd:string">^GSPC</SP500>
</multiRef>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Let's now review the code of `Market`, `MarketData`, and their deployment descriptor.

We will start with the `Market` class. A quick glance at the code below shows us that the `getQuote()` method is modified to return an instance of `MarketData` rather than a string as we did in `StockQuote`:

```
public MarketData getQuote(String ticker) throws Exception {
 return new MarketData(ticker);
}
```

The input argument is still the same: the ticker symbol of the stock we are interested in. As we mentioned previously, the `MarketData` class contains the ticker symbol, its value, and the values of the major US stock indices:

```
public final class MarketData {

 public String DOW = "^DJI";
 public String NASDAQ = "^IXIC";
 public String SP500 = "^GSPC";

 String ticker; // The ticker symbol.
 String stringValue; // The value of the stock as a string
 double doubleValue; // The value of the stock as a double
 double indices[]; // The DOW, the NASDAQ, and the S&P 500 values

 ...
 public MarketData(String ticker) throws Exception {
 this.ticker = ticker;

 StockCore stockcore = new StockCore();

 stringValue = stockcore.getQuote(ticker);
 doubleValue = Double.parseDouble(stringValue);
 indices = new double [3];
 indices[0] = Double.parseDouble(stockcore.getQuote(DOW));
 indices[1] = Double.parseDouble(stockcore.getQuote(NASDAQ));
 indices[2] = Double.parseDouble(stockcore.getQuote(SP500));
 }
}
```

The development of the `MarketData` class is straightforward thanks to the `StockCore` class that we built earlier. The remainder of the `MarketData` code contains the set and get methods, which follow the JavaBean pattern.

Prior to running the client, we deployed the `Market` web service with the following descriptor. This deployment descriptor is similar to `StockQuote.wsdd`. The key difference is the serialization information contained in the `<beanMapping/>` element:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
 xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
 <service name="Market" provider="java:RPC">
 <parameter name="className"
 value="com.wrox.jws.stockquote.Market"/>
 <parameter name="allowedMethods" value="getQuote"/>
 </service>
 <beanMapping qname="ns:MarketData"
 xmlns:ns="http://stockquote.jws.wrox.com"
 languageSpecificType=
 "java.com.wrox.jws.stockquote.MarketData"/>
</deployment>
```

The `<beanMapping/>` element contains the necessary information for the **bean serializer** to perform the Java to XML serialization and XML to Java deserialization. It is a helper class provided by Axis to serialize and deserialize instances of classes that follow the JavaBean pattern. For the purpose of this discussion, a JavaBean is a Java class with a public default constructor and (public) set/get methods to set

and get the values of instance data. Note that the JavaBean specification also requires the implementation of the `Serializable` interface, but we will not use that interface in this discussion.

That wraps up our final example, but before we finish off the chapter, let's quickly discuss the `<beanMapping/>` element in more detail.

### beanMapping Element

`<beanMapping/>` specifies the following information:

- **A QNAME (and its namespace)**

An XML namespace-qualified name (QName) that identifies the element to be serialized and deserialized. In the `Market` web service example the element is <http://stockquote.jws.wrox.com.MarketData>.

- **A language-specific (for example Java or C#) type**

The data type that will be used to represent the element in a specific language, in our case Java.

When the bean serializer needs to generate XML from a Java data structure (serialize), it goes through the following steps:

- **Create an XML element for the data structure**

This element is either a place holder (`<multiRef/>`) or bears the same name as the Java data structure (for example `<indices/>`).

- **Serialize each data element that supports a get method**

For instance `stringValue`, `doubleValue` in the case of `MarketData`.

When the bean serializer needs to generate a Java data structure from an XML document (deserialize), it goes through the following steps:

- **Create a Java class for the XML element**

This is done using the public default constructor.

- **Deserialize each data element that supports a set method**

The bean serializer reads the data from the XML document and calls the set method corresponding to the data. For instance, from our previous example, when encountering the `<doubleValue/>` element, it calls the `MarketData.setDoubleValue` to set the value of `MarketData.doubleValue`.

As long as the data being exchanged in SOAP packets can be represented by JavaBeans, they can be serialized and deserialized using the bean serializer. This is the case with most types of data, but sometimes, a bean will not do the job. For instance, if the object that needs to be serialized contains runtime-only data structures like references that would be too expensive to exchange with every call then we might have to write our own serializer. A potential use of **custom serializers** is the optimization of the data being transferred, for instance, we could write a custom serializer for strings that compresses the data when it reaches a certain size.

Before considering the use of a custom serializer, take note of the fact that they have the potential of limiting interoperability. For more detail about custom serializers, please refer to the user's guide that comes with Axis (`%axisDirectory%\docs\users-guide.html`).

## Summary

We have covered a lot of topics in this chapter. After installing Axis, we created a very simple web service called `HelloWorld`. The `HelloWorld` service defines one method, `getQuote()`, which takes a string as its input argument and modifies the input before returning it to the client. We also saw that thanks to the Axis framework and the Java Web Service (JWS) concept, the creation of a web service based on a Java class was a trivial exercise. After deploying and testing the `HelloWorld` service, we used `tcpmon` to look at the SOAP packets as they traveled between the client and the server.

We then downloaded and configured Tomcat to run with Axis as a production-ready replacement for the `SimpleAxisServer`, and tested Tomcat and Axis using `HelloWorld`. We then developed the `StockQuote` web service using a packaged library, namely `StockCore`. This more realistic web service allowed us to introduce Axis deployment descriptors and the flexibility that they provide.

We concluded the chapter with the `Market` example, which returns a structure and an array that can be serialized and de-serialized by the bean serializer provided by Axis.

This chapter has given us enough information to create and deploy our own web services. In the next chapter, we will discuss techniques for describing the features of web services in a platform- and language-independent manner.