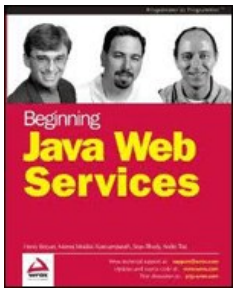


Beginning Java Web Services

by Henry Bequet

Chapter 8: Asynchronous Web Services

Overview

Up to this point, we have learned the basics of web services – how to create and consume them using Java. The examples we discussed so far have mostly relied on SOAP communication between the requester and the provider of a web service. This meant that the requester of the service sent a request envelope and the provider returned a response envelope.

In this chapter, we will focus on scenarios where the request is not followed by an immediate response. Instead, a request is sent out and the caller is not blocked until the response arrives. The caller can pick up the response, if there is one, at a later time. We call web services implementing such scenarios **asynchronous web services.** After we have looked at it from a programming model perspective, we can apply what we learned there to different web service scenarios like **message-based web services.**

The implementations of this type of web services are different from those of synchronous, RPC – style web services. They are also accessed in a different manner. Overall, there are many possibilities regarding the use of message-based web services. In this chapter we shall be discussing two of them. Namely:

- The **Java API for XML-based Messaging** or **JAXM,** which allows us to build and process generic XML messages with special support for SOAP messages.
- The **Java Message Service (JMS)** API. This is another standard Java interface for messaging. We can send SOAP messages over this layer, or we can send invocations for web services over this interface in some other format.

As we can see, there are many options for implementing and accessing web services in an asynchronous manner. By the time we complete this chapter, we will be in a position to understand what these options are.

Programming Models Revisited

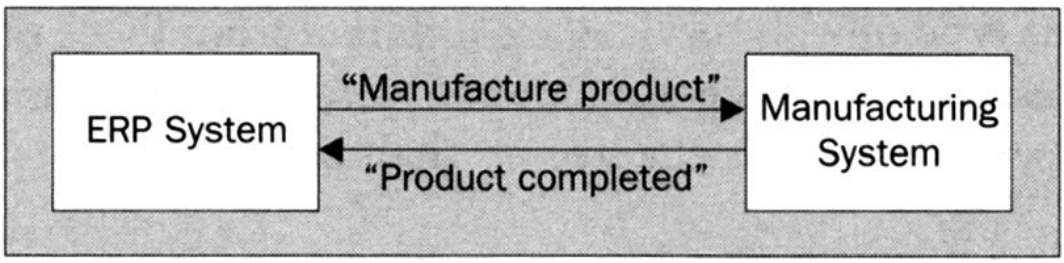
We cannot stress it enough: web services are all about allowing applications to interact with each other. So far, we have been focusing on synchronous communication. An application (the requester) communicates with another application (the provider) by invoking a function that the provider exposes in the form of a web service. After invoking this exposed function, the requester process blocks itself, until a response from the invoked service comes back.

In some cases, this blocking of the process is not acceptable; to say the least, this is not a productive use of our computing resources. The invoked web service may take a long time to complete its function. Instead of waiting for the reply, this waiting time of the requester can be used to perform some other task. Moreover, the called web service might not return any data, or the requester might not need the returned data immediately. In these types of scenarios, it is sufficient for us to trigger the execution of a remote function, without having to wait for its completion.

For example, let's assume that a business firm has two distinct software packages – one of them is an Enterprise Resource Planning (ERP) system, which apart from doing other functions is also responsible for handling the new orders placed by a customer. The other software package runs on the manufacturing floor and controls the manufacturing process for these orders.

These systems are loosely coupled. In other words, on getting an order, the ERP system will trigger some processes in the manufacturing system, so that the ordered products are actually build. At the same time, the manufacturing system will possibly send updates about the status of an order back to the ERP system.

Now, these two interactions can be asynchronous, as they do not require immediate feedback from the invoked service. Besides, it also possible that the triggered processes (for example, the manufacturing of a product) might take hours or sometimes even days to complete. In such a case, the ERP system cannot be kept waiting till the manufacturing system completes its work. A diagram showing these two systems is as follows:



In these types of daily settings, asynchronous message-based systems have been deployed for a long time. The advent of web services doesn't change this fundamental approach. It merely puts formalization around the model of system interaction, thereby making it easier for us to implement these systems. The implementation gets easier as the communication between existing subsystems is based on open standards, which work across heterogeneous platforms and different programming languages.

Message-based Web Services

The term **message-based web service** is somewhat redundant because a web service is always invoked through a request message by a requester. In the web services scenarios that we discussed in the previous chapters, we considered the request message to be an invocation of a function, which accepts parameters and returns a result. In this case, though, we talk about a generic message that contains a bunch of data, which a client passes to a server. The data that is sent by a requester to the provider is not meant to be interpreted as a collection of parameters to a function.

So, what does the programming model for these services look like? On a high level, the various systems that communicate in this environment are generally more event-driven than systems in other environments. A particular type of activity in one system (like the arrival of a customer order) leads to an event, which triggers other activities in another system (or systems). As a result, this can lead to environments where many processes can be taking place at the same time.

One crucial issue is how we map these characteristics in our web services. We have already seen that web services are described by using port types in WSDL. Each port type consists of one or more operations. There are four types of operations – request-response, one-way, solicit-response, and notification. What we have looked at so far were the request–response operations where the requester blocks after sending the request message until it gets the response message.

For message-based web services, which are using asynchronous communication to interact with their counterparts, one-way operations are more common. Note that WSDL does not mandate that request-response operations have to be handled synchronously. A requester could send a request message, continue to do other work, and receive the response to the request at a later time. However, in most cases this style of interaction would be implemented by using multiple one-way operations.

Keep in mind, though; that the WSDL specification does not mandate the use of synchronous or asynchronous communication; this is left to the underlying protocol to determine. WSDL maintains a strict separation between the interface of a service and the protocol that is used to invoke it. This means that we can have, for example, a request-response operation and still use an asynchronous protocol to handle the invocation of such an operation.

Speaking of protocols, is SOAP (the primary protocol for web services) a synchronous protocol or an asynchronous one? The answer is, that it is neither. The specification does not mandate the use of a certain network protocol to transmit a SOAP message, and at its very core, SOAP defines one-way messages.

This means that a SOAP request does not have to lead to a SOAP response. If we are using SOAP over HTTP, then we will make a synchronous request, because HTTP is a synchronous protocol. However, even in this case, the SOAP specification does not mandate that a SOAP response be returned. Thus, we could implement a compliant run time environment that supports asynchronous web service invocations, using SOAP over HTTP.

Besides, SOAP can also be used over other network protocols. For example, we could use SOAP over a messaging environment supporting the Java Message Service (JMS).

JMS defines an API that allows us to send and receive messages from a queue. Thus, this API supports asynchronous communication between programs' out of the box', which makes it easier to implement asynchronous web services. We will look at this case in more detail later on.

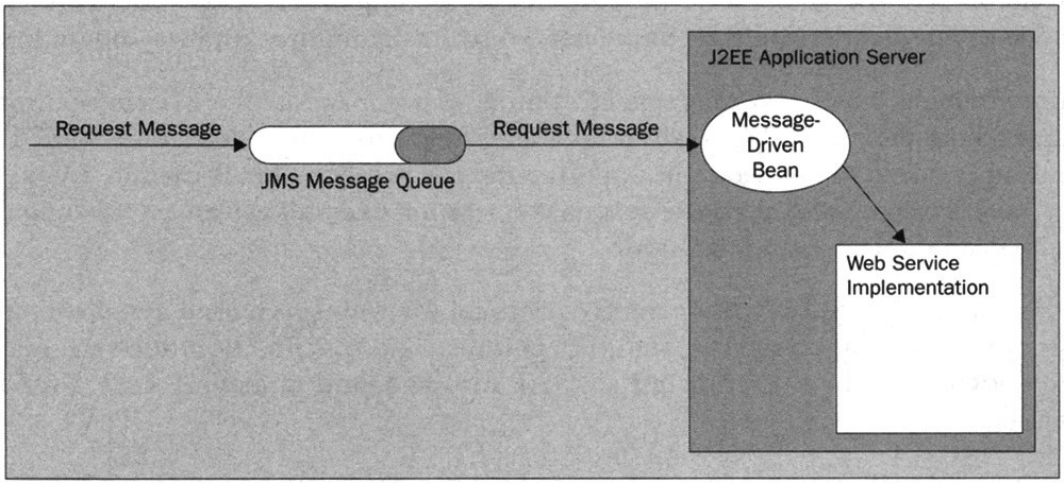
JMS also defines a way for broadcasting messages to multiple receivers by a **publish-subscribe mechanism.** In this case, clients subscribe to a certain 'topic'. Whenever a message is published on that topic, it is automatically forwarded to all subscribers, which could be used in web services scenarios, too. So, given that SOAP can be used for asynchronous, message-based web services, what other protocols are available? We have already mentioned JMS. In the previous chapter, we saw that in WSDL, we can describe the details of accessing a web service in the `<binding>` element. In this element, we can also define information that describes how to communicate with a web service over JMS. However, note that so far, this is not standardized.

Message-Driven Bean (MDB)

In all cases where JMS is used as a transport layer for web services – be it transporting SOAP envelopes or data that was formatted in some other way, we need some process to receive the messages arriving for a service. The Java 2 Platform, Enterprise Edition (J2EE) specification defines a special kind of Enterprise JavaBean (EJB) for this purpose, it is known as **Message-Driven Bean (MDB).**

An MDB is configured to receive messages from a particular JMS queue or topic. Whenever a message arrives on a queue or topic that an MDB is configured for, the J2EE application server will automatically receive this message and forward it to this MDB.

We will look at this setup again in more detail later. For now, just keep in mind that from a programming model perspective, an MDB is the most common way to receive web services invocations for services using JMS as the transport layer. After receiving the invocations, the MDB will forward them to the actual web service:



If there is a response (depending on whether the operation was defined as a request-response, or a one-way operation), it is sent back to the requester in the same way. That is, the MDB will pick up the returned data, format it into a SOAP message, and send it to an output message queue. This queue is likely to be different from the receiver queue, and from there the requesting client can pick up the response.

If we wanted to build a solution that does not require a full J2EE application server environment with its support for MDBs, we would have to develop some code that can receive messages from a JMS queue and forward them to a web service. The advantage that the MDB and the J2EE environment brings us is that the reception of the message and the invocation of the service run in a secure and transactional environment. Moreover, if we are using an MDB, we don't have to write any JMS code, the J2EE application server will take care of that for us.

Now that we have looked at things from a programming model perspective, let's look at their implementation in Java.

The JAXM API

> **Note** The latest JAXM specification can be obtained from http://java.sun.com/xml/downloads/jaxm.html and the SAAJ specification from http://java.sun.com/xml/downloads/saaj.html.

As we mentioned earlier, the Java API for XML-based Messaging, or JAXM, provides us with a set of interfaces for building and processing generic XML messages, with special support for SOAP messages. These interfaces focus more on a messaging approach to web services, as opposed to the more procedural approach of JAX-RPC (discussed in the previous chapter).

The most recent version of JAXM specification is version 1.1. The JAXM 1.0 specification was split up into two new specifications, namely JAXM 1.1 and SOAP with Attachments API for Java (SAAJ) 1.1.

JAXM depends on SAAJ, so generally we get both of them in one set; however, they are available in separate packages as well. The reason for creating a separate package is to allow other packages (most notably JAX-RPC) to take advantage of the SOAP classes in SAAJ, without creating a dependency on all of JAXM. This split into two specifications is also reflected in the Java packages – the JAXM specification defines classes in the `javax.xml.messaging` package, whereas the SAAJ specification uses the `javax.xml.soap` package.

> **Note** Since both APIs are so closely related, for our discussions in this chapter we will only refer to JAXM, but please note that it will implicitly include SAAJ also.

JAXM defines a messaging interface for both synchronous and asynchronous communication. This communication can follow both a request-response and one-way protocol. For example, we could execute a request-response scenario in an asynchronous fashion. This means a client would send out a request to a service, then do some other work and come back later to pick up the response for the request.

> **Note** Note that in this case, the client needs to implement a mechanism to correlate a response message or any acknowledgement to a particular request message. JAXM does not provide any built-in support for this.

The messages that we send and receive in JAXM are SOAP messages. Even though it is possible, other message protocols are not defined at this time. The fact that SOAP messages are exchanged also implies that on one side of the equation, the sender or the receiver can have a non-JAXM implementation. A JAXM client can send a SOAP message to a web service that was implemented using a different interface, as long as both of them support the same SOAP format.

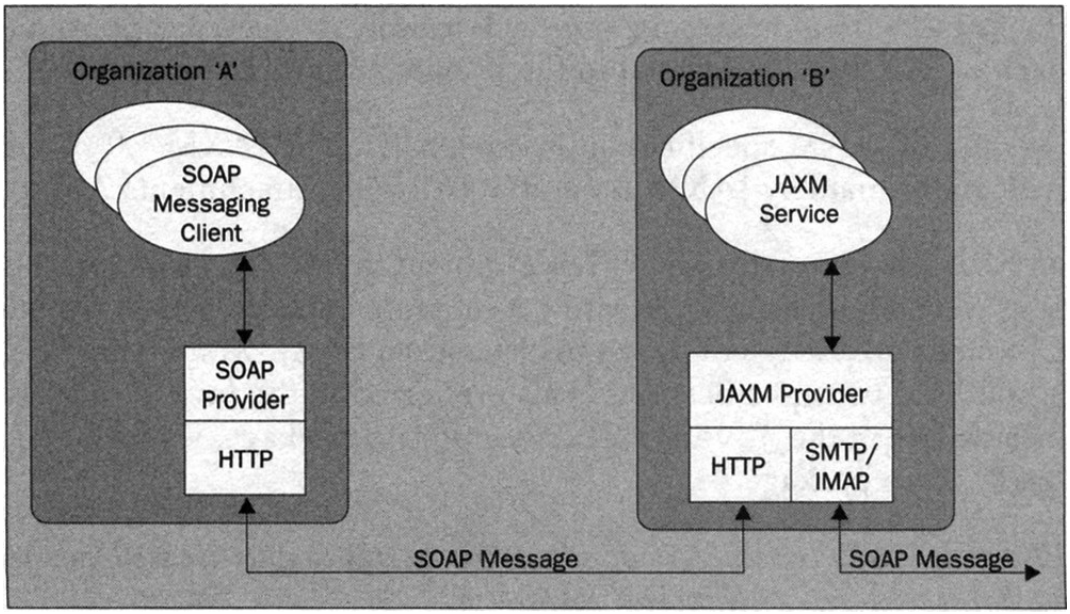
For example, we could take a WSDL definition of a web service and generate a JAXM client proxy class for it. However, if the web service is using 'RPC' style then using JAX-RPC may be the easier way.

The Provider

JAXM also supports two styles of sending a SOAP message. It can use a so-called JAXM **provider**, or it can send messages directly. If no provider is used, the message can only be sent synchronously through HTTP, to a concrete URL.

A provider is a piece of code that picks up a message from a JAXM sender, and forward it to the actual destination on the sender's behalf. Similarly, the provider receives a message on behalf of the JAXM receiver and forwards it to this receiver. This decouples the JAXM sender and receiver from the actual implementation of the message transport, since that is handled by the provider. However, note that to take advantage of a provider, a JAXM application needs to run in a J2EE container (for example, either run as part of a servlet application or as part of an EJB application). In this chapter we will be using Tomcat's web container.

The diagram below shows a typical scenario of sending a SOAP message over JAXM:



In this scenario, the client uses any SOAP-client implementation to build and send a SOAP message. This message goes over HTTP to a JAXM provider on the receiver side, which delivers the message to the service. This message could also be forwarded to another endpoint over another protocol such as SMTP.

As we have seen in the earlier chapters, SOAP does not define the content of any message. The specification only states that a SOAP message is contained within a root element called `<Envelope>`. This root element can have two children – namely `<Header>` and `<Body>`. The content of these elements is not defined. Higher-level standards can be used to define what this content looks like.

For example, the ebXML standard, which is an effort sponsored by OASIS and the United Nations, defines rules and data structures that can be used for electronic data interchange between companies. As part of this, it specifies message formats that are needed, in particular B2B scenarios. These formats are based on SOAP.JAXM provides the concept of a **Profile** to enable us to add definitions about these higher-level formats thereby helping us to build messages conforming to these formats. For example, there is an ebXML profile, which defines the additional fields available in the SOAP header. If a JAXM client uses a profile, both the sender and the receiver of the message have to agree on the same content.

Synchronous Communication

As we mentioned above, JAXM supports both synchronous and asynchronous communication. There are two different programming models to deal with either one. For the synchronous model, JAXM does not require the use of a provider; rather each sender communicates directly with the receiver of a message. The `javax.xml.soap.SOAPConnection` class contains a `call()` method that allows sending a SOAP request message and waiting for the response document to be returned.

Installing the Java XML Pack

The examples in this chapter will take advantage of the JAXM reference implementation in the **Java XML (JAX) Pack** (at the time of writing this was in its Summer 02 incarnation). We can download this package from http://java.sun.com/xml/downloads/javaxmlpack.html.

The package comes in a zip file, which we can extract into a new directory. For example, if we extract the contents of the zip file into our C: drive, we will find the package at `C:\java_xml_pack-summer-02`. It contains a number of subdirectories with implementations for the various XML APIs. The JAXM implementation will be in the `C:\java_xml_pack-summer-02\jaxm-1.1` directory.

> **Note** Hereafter we'll refer to the installation directory of the JAX Pack by the variable `%JAX_Pack%`.

For compiling and running the examples given in this chapter, we will need the following JAR files in our classpath:

- `%JAX_Pack%\jaxm-1.1\lib\jaxm-api.jar`
- `%JAX_Pack%\jaxm-1.1\lib\saaj-api.jar`

- %JAX_Pack%\jaxm-1.1\lib\commons-logging.jar
- %JAX_Pack%\jaxm-1.1\lib\dom4j.jar
- %JAX_Pack%\jaxm-1.1\lib\mail.jar
- %JAX_Pack%\jaxm-1.1\lib\activation.jar
- %JAX_Pack%\jaxm-1.1\jaxm\jaxm-runtime.jar
- %JAX_Pack%\jaxm-1.1\jaxm\saaj-ri.jar
- %JAX_Pack%\jaxp-1.2\jaxp-api.jar
- %JAX_Pack%\jaxp-1.2\sax.jar
- %JAX_Pack%\jaxp-1.2\xercesImpl.jar
- %JAX_Pack%\jaxp-1.2\xalan.jar
- %JAX_Pack%\jaxp-1.2\dom.jar

Try It Out: Sending a Synchronous Request over JAXM

Let's now look at an example that shows us how to send a synchronous request using the `javax.xml.soap.SOAPConnection` class:

1. Save the following code as `TestJAXMSync.java`:

```
import javax.xml.messaging.*;
import javax.xml.soap.*;

import javax.xml.transform.*;
import javax.xml.transform.stream.*;

public class TestJAXMSync {

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

    //build a SOAP message
    MessageFactory mf = MessageFactory.newInstance();
    SOAPMessage message = mf.createMessage()
    SOAPPart part = message.getSOAPPart();
    SOAPEnvelope envelope = part.getEnvelope();
    SOAPBody body = envelope.getBody();

    // set up some namespace declarations in the envelope
    envelope.addNamespaceDeclaration("xsd",
      "http://www.w3.org/2001/XMLSchema");
    envelope.addNamespaceDeclaration("xsi",
      "http://www.w3.org/2001/XMLSchema-instance");
    envelope.addNamespaceDeclaration("SOAP-ENC",
      "http://schemas.xmlsoap.org/soap/encoding/");

    // build the element in the body: <getQuote>...</getQuote>
    Name name1 = envelope.createName("getQuote");
    SOAPBodyElement bodyElement = body.addBodyElement (name1);

    // build the parameter element:
    // <arg0 xsi:type="xsd:string">IBM</arg0>
    SOAPFactory soapFactory = SOAPFactory.newInstance();
    SOAPElement arg0 = soapFactory.creatElement ("arg0");
    Name name3 = soapFactory.createName ("type", "xsi",
      "http://www.w3.org/2001/XMLSchema-instance");
    arg0.addAttribute(name3, "xsd:string");
    arg0.addTextNode("IBM");

    //add the parameter element to the getQuote element
    bodyElement.addChildElement (arg0);

    // use a connection factory to create a new connection
    SOAPConnectionFactory factory =
      SOAPConnectionFactory.newInstance();
```

```
        SOAPConnection conn = factory.createConnection();

        // create an end point and make the call
        URLEndpoint endpoint =
          new URLEndpoint ("http://localhost:8080/axis/services/StockQuote");
        SOAPMessage response = conn.call(message, endpoint);

        // interpret response
        System.out.println ("Result:");
        TransformerFactory tFact = TransformerFactory.newInstance();
        Transformer transformer = tFact.newTransformer();
        Source src = response.getSOAPPart().getContent();
        StreamResult result=new StreamResult (System.out);
        transformer.transform(src, result);
        System.out.println();
      }
    }
```

2. Complile the file, (make sure that the JAR files that we listed earlier are in the classpath):

   `javac TestJAXMSync.java`

3. To run the program type:

   `java TestJAXMSync`

We will get the following output:

How It Works

Since this chapter is about asynchronous communication with a web service, we will not cover this example in great detail. There is a detailed explanation about SOAP message creation later in the chapter when we get to the asynchronous example. Here, let us just have look at the code that is specific to the synchronous case.

To send a synchronous request, we need to create a `SOAPConnectionFactory`. This factory object lets us create a `SOAPConnection`. Sending the request is done by using the `call()` method, passing in the actual SOAP message and an instance of `URLEndpoint`. The `URLEndpoint` object represents the address of the web service that we send the request to:

```
// use a connection factory to create a new connection
SOAPConnectionFactory factory =
  SOAPConnectionFactory.newInstance();
SOAPConnection conn = factory.createConnection();

//create an end point and make the call
URLEndpoint endpoint =
  new URLEndpoint ("http://localhost:8080/axis/services/StockQuote");
SOAPMessage response = conn.call (message, endpoint);
```

In this example, no provider is used and the call is synchronous. In other words, the calling process will block until the response is returned. As we mentioned before, the receiver of the message can be a JAXM application or any other application that can receive and process a SOAP message.

Asynchronous Communication

In this section we will be covering asynchronous communication in greater detail. We will also discuss senders and receivers of asynchronous

messages.

Sending Asynchronous Messages

There are five steps involved in sending a message asynchronously. They are:

1. Obtain an instance of `javax.xml.messaging.ProviderConnectionFactory`.
2. Create a `javax.xml.messaging.ProviderConnection` object.
3. Create a `javax.xml.soap.MessageFactory` object.
4. Create a `javax.xml.soap.SOAPMessage` object and populate it.
5. Send the message to its target.

We will walk through each of these steps here by looking at an example class that sends an asynchronous request to a web service via JAXM.

Try It Out: Building an Empty SOAP Message

Now let us look at an example which building a new SOAP message and print it out on the screen:

1. Save the code shown below in a file called as `TestJAXM1.java`:

```
import javax.xml.messaging.*;
import javax.xml.soap.*;

public class TestJAXM1 {

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

    MessageFactory msgFactory = MessageFactory.newInstance();
    SOAPMessage message = msgFactor.createMessage();
    message.writeTo (System.out);
    System.out.println ("/n");
  }
}
```

2. The command for compiling and running the example is:

```
javac TextJAXM1.java
java TextJAXM1
```

This will give us the following output:

How It Works

This code is pretty straightforward. In order to create a new `SOAPMessage` object, we create a `MessageFactory` object, which returns a new and empty SOAP message:

```
MessageFactory msgFactory = MessageFactory.newInstance();
SOAPMessage message = msgFactory.createMessage();
```

Finally, we take advantage of the `writeTo()` method in the `SOAPMessage` class to print the content of the message on the screen:

```
message.writeTo (System.out);
```

Providers and Profiles

Earlier we mentioned that a provider is needed for asynchronous communication. A JAXM application gets access to a provider by using a `javax.xml.messaging.ProviderConnectionFactory` object. The application will use this factory object to create new `javax.xml.messaging.ProviderConnection` objects, which are later used to send the actual message. We will get to that in a little while. Note that, by using the factory and the `javax.xml.messaging.ProviderConnection` instance, an application does not have any dependency on the provider, nor does it know which provider it is using.

So, how does the application get access to the correct `javax.xml.messaging.ProviderConnectionFactory` object? According to

the JAXM specifications, it is expected that a JAXM application using asynchronous communication will exist in a J2EE container. Let us discuss briefly what this means.

J2EE provides two types of container – the EJB container and the web container. In both the cases, the code that runs in the container is under control of the J2EE application server. This means, among many other things, that the code has access to information stored in a name server using the **Java Naming and Directory Interface (JNDI)**. While we can store all kinds of things in a JNDI name server, typically we use it to store references for commonly used objects.

In the case of JAXM, we store `javax.xml.messaging.ProviderConnectionFactory` object in the JNDI name server. Normally, the object is configured by the application server administrator and then created at run time whenever the application server is started. A reference to this object is then stored in the JNDI name server, from where the application can pick it up through a normal lookup.

Here is a code snippet that shows us how to use the JNDI name server to find a `javax.xml.messaging.ProviderConnectionFactory` instance:

```
Context context = new InitialContext();
ProviderConnectionFactory factory =
  context.lookup ("myProviderConnectionFactory");
```

However, the use of JNDI in J2EE containers is beyond the scope of this book, and we won't go into any more detail about it here. Luckily an alternative approach exists; the `javax.xml.messaging.ProviderConnectionFactory` class provides a static method called `newInstance()`, which returns a `javax.xml.messaging.ProviderConnectionFactory` for the default provider. This default provider depends upon the environment in which the application is running and also on the JAXM implementation

Once we have created a connection object over which we can send a message, we have to create the message. To do so, we use a `javax.xml.soap.MessageFactory` object. There can be different types of message factory objects, supporting different types of SOAP messages. Previously, we said that JAXM uses profiles to support different types of standard SOAP message formats; here is where these profiles come into play.

A `MessageFactory` object knows how to create a message. Thus, for different profiles, there are different `MessageFactory` objects. While creating a `MessageFactory` object from a `javax.xml.messaging.ProviderConnection` object, we have to specify the profile for the new message. For example, to build a message that follows the ebXML definitions, we would create the `MessageFactory` like this:

```
MessageFactory msgFactory = connection.createMessageFactory ("ebxml");
```

For our examples, we will use the profile that comes with the Java XML Pack, namely the `soaprp` profile. This profile acts as a simple example for what a profile can look like and allows for the exchange of basic SOAP messages.

Try It Out: Installing the JAXM Provider in Tomcat

Before we are able to use a JAXM Provider we need to set up the JAXM Reference Implementation in the form of a web application. Therefore, we will configure it to work without Tomcat installation. The process is not unlike installing Axis.

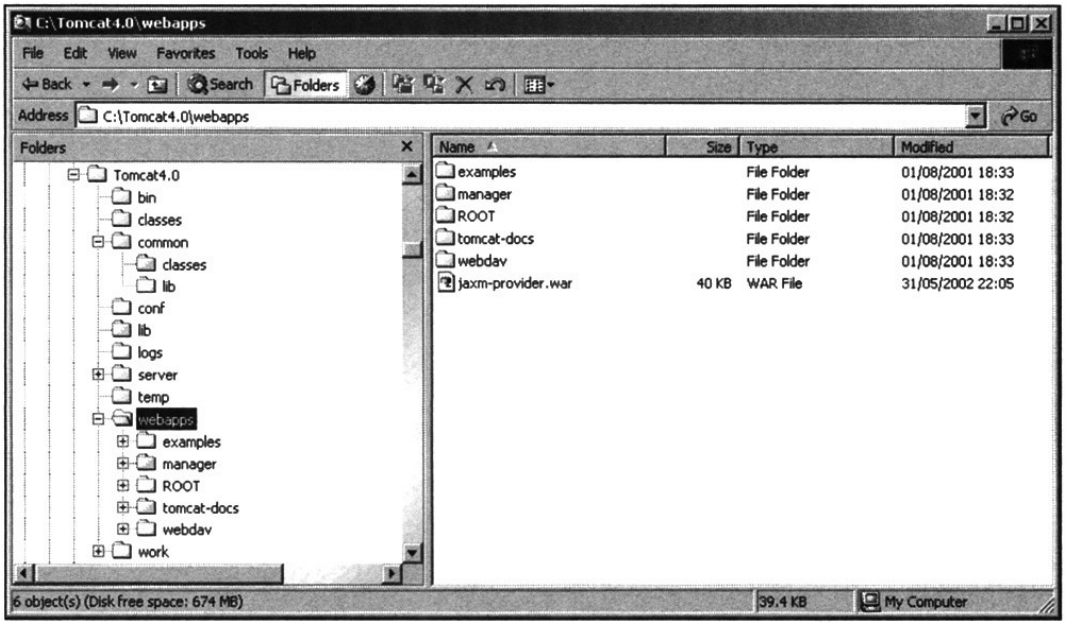
1. We need to make sure the JAXM JAR files are loaded into Tomcat's classpath. So copy all the JAR files from the `%JAX_HOME%/jaxm-1.1/lib` and `%JAX_HOME%/jaxm-1.1/jaxm` directories into `%CATALINA_HOME%/common/lib`.

2. Next we also need some of the JAXP JARs, so copy `jaxp-api.jar` from the `%JAX_HOME%/jaxp-1.1` directory into `%CATALINA_HOME%/common/lib`.

3. Unfortunately the SOAP JAR that we installed with Axis conflicts with the JAXM installation so we need to remove the classpath reference to Axis' `saaj.jar` file in Tomcat's `setclasspath.bat` file (which we set in Chapter 3):

   ```
   ...
   set CLASSPATH=%classpath%;%axisDirectory%/lib/log4j-1.2.4.jar
   set CLASSPATH=%classpath%;%axisDirectory%/lib/tt-bytecode.jar
   rem set CLASSPATH=%classpath%;%axisDirectory%/lib/saaj.jar
   ...
   ```

4. To install the provider itself, all we need to do is copy the `jaxm-provider.war` file (from the `%JAX_HOME%\jaxm-1.1\jaxm` directory) to the `%CATALINA_HOME%\webapps` directory:



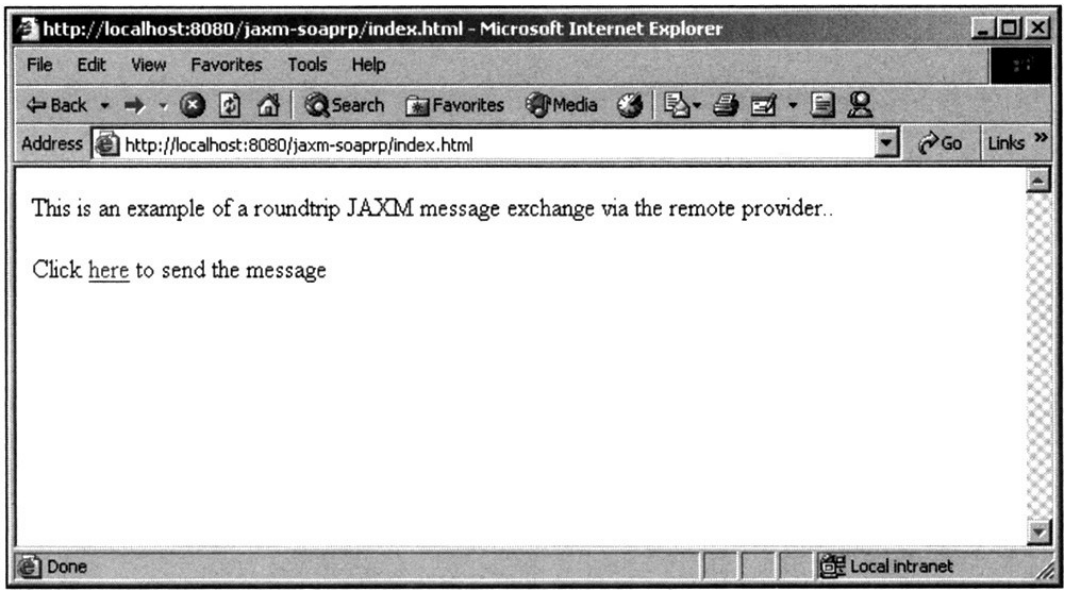
The next time we start Tomcat, it will automatically deploy this web application for us.

5. In order to test that our JAXM provider installation was successful, we'll also install one of the applications that comes with JAXM.

   Copy the `jaxm-soaprp.war` file from the `%JAX_HOME%\jaxm-1.1\samples` directory to the `%CATALINA_HOME%\webapps` directory.

6. Start up Tomcat.

7. Browse to `http://localhost:8080/jaxm-soaprp/` to test the JAXM installation:



Now we're read to code.

Try It Out – Building an Empty SOAP Message for a Profile

Let's build a SOAP message that can be used with a provider and which follows a certain profile.

1. Here is the complete listing of the example application, entitled `TestJAXM2.java`:

```
import javax.xml.messaging.*;
import javax.xml.soap.*;

import com.sun.xml.messaging.jaxm.soaprp.*;
```

```
public class TestJAXM2 {

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

    ProviderConnectionFactory factory =
      ProviderConnectionFactory.newInstance();
    ProviderConnection connection = factory.createConnection();

    ProviderMetaData metaData = connection.getMetaData();
    String [] supportedProfiles = metaData.getSupportedProfiles();
    String profile = null;

    for (int i = 0; i < supportedProfiles.length; i++) {
      if (supportedProfiles [i]. equals ("soaprp")) {
        profile = supportedProfiles [i];
        break;
      }
    }

    MessageFactory msgFactory = connection.createMessageFactory (profile);
    SOAPMessage message = msgFactory.createMessage();

    message.writeTo (System.out);
    System.out.println ("\n");

  }
}
```

2. Compile the code:

   `javac TestJAXM2.java`

3. The `javax.xml.messaging.ProviderConnection` implementation class uses a file called `client .xml` to find the right provider. The client will exchange some initialization message with the provider to learn about available profiles, and so on. Here is what the `client .xml` file for our example should look like:

```
<?xml version="1. 0" encoding="ISO-8859-1"?>

<!DOCTYPE ClientConfig
  PUBLIC "-//Sun Microsystems, Inc. //DTD JAXM Client //EN"
  "http: // java.sun.com/xml/dtds/jaxm_client_1_0.dtd">

<ClientConfig>
  <Endpoint>
    The JAXM client
  </Endpoint>

  <CallbackURL>
    http://localhost:8080/dummy
  </CallbackURL>

  <Provider>
    <URI>http://java.sun.com/xml/jaxm/provider</URI>
    <URL>http://localhost:8080/jaxm-provider/sender</URL>
  </Provider>

</ClientConfig>
```

   This file must be available on the classpath of the client. If we have the current directory (".") in our classpath, we can simply create the file in the current directory.

4. Now we can run the class:

   `java TestJAXM2`

   It creates the following output:

How It Works

There are a number of important steps that are executed here. Let us now look at each of these steps in detail.

As mentioned earlier, for our examples, we are using the Java XML Pack, Summer 02. The JAXM implementation in this package comes with a default `javax.xml.messaging.ProviderConnectionFactory` object that can be obtained through the `newInstance()` method, like this:

```
ProviderConnectionFactory factory = ProviderConnectionFactory.newInstance();
```

The factory object can now be used to create a `javax.xml.messaging.ProviderConnection` object. This is a straightforward operation:

```
ProviderConnection connection = factory.createProviderConnection():
```

A regular SOAP message does not contain any information about the receiver of the message. Thus, we have to add additional information here. The `javax.xml.messaging.ProviderConnection` implementation class uses the `client.xml` file to find the right provider:

```
<?xml version="1. 0" encoding="ISO-8859-1"?>

<!DOCTYPE ClientConfig
  PUBLIC "- //Sun Microsystems, Inc. //DTD JAXM Client //EN"
  "http: //java.sun.com/xml/dtds/jaxm_client_1_0.dtd">

<ClientConfig>
  <Endpoint>
    The JAXM client
  </Endpoint>

  <CallbackURL>
    http://localhost:8080/dummy
  </CallbackURL>

  <Provider>
    <URI>http://java.sun.com/xml/jaxm/provider</URI>
    <URL>http://localhost:8080/jaxm-provider/sender</URL>
  </Provider>

</ClientConfig>
```

The `<CallbackURL>` element describes the address to which the provider can send back messages destined for this client. In our case, however, the client is a command-line application that cannot receive any messages. We will be covering the acceptance of asynchronous JAXM messages later in the chapter. So, for now, we can simply put a dummy address here.

The `<Provider>` element describes the address of the provider. In our case, we installed the JAXM SOAPRP provider from the Java XML Pack into Tomcat 4 so the URL will be at http://localhost:8080/jaxm-provider/.

Part of using a profile is always to make sure that both sides agree on the structure of message. In fact, that is what profiles are all about – making sure that both parties can build and interpret the messages they exchange. The Java XML Pack comes with two predefined profiles – one for ebXML and one for SOAPRP. For example, one of them is the `com.sun.xml.messaging.jaxm.soaprp.SOAPRPMessageImpl` class, which adds endpoint definitions to the SOAP message to tell the provider where a message should go.

We can check the `javax.xml.messaging.ProviderConnection` object, to make sure it supports the profile we want to use (in this case, the `soaprp` profile), and then create a `MessageFactory` instance:

```
ProviderMetaData metaData = connection.getMetaData();
String[] supportedProfiles = metaData.getSupportedProfiles();
String profile = null;
```

```
for (int i =0 ; i < supportedProfiles.length; i++) {
  if (supportedProfiles[i].equals("soaprp")) {
    profile = supportedProfiles[i];
    break;
  }
}
```

Once we have obtained the `MessageFactory` object, we can use it to build the actual SOAP message and print it out on the screen:

```
MessageFactory msgFactory = connection.createMessageFactory(profile);
SOAPMessage message = msgFactory.createMessage();
message.writeTo(System.out);
```

Populating SOAP Messages

The `javax.xml.soap.MessageFactory` object allows the creation of message objects, specifically, SOAP message objects. The `javax.xml.soap.SOAPMessage` class represents SOAP messages. Each `javax.xml.soap.SOAPMessage` object, regardless of the profile that it is created for, contains at least four other objects in it. They are:

- A `javax.xml.soap.SOAPPart` object
  As its name indicates, the SAAJ specification supports SOAP attachments. If a SOAP message, with attachment, is sent, it contains multiple parts – the SOAP part and one or more other parts (the actual attachments). As SAAJ considers each `SOAPMessage` object as one that potentially has an attachment, it stores the actual SOAP message in the `SOAPPart` attribute.
- A `javax.xml.soap.SOAPEnvelope` object
  In the `SOAPPart` object, we find a `SOAPEnvelope` object. This object represents the content of the actual `<Envelope>` element and all of its children.
- A `javax.xml.soap.SOAPHeader` object
- A `javax.xml.soap.SOAPBody` object
  Obviously, once we have access to the envelope object, we can retrieve the values of the `<Header>` and `<Body>` elements from within that envelope.

Here is an example that shows how to parse an existing `javax.xml.soap.SOAPMessage` object and retrieve its content:

```
// receive the SOAP message from somewhere, we will cover this later
SOAPMessage message = ...
SOAPPart part = message.getSOAPPart();
SOAPEnvelope envelope = part.getEnvelope();
SOAPHeader header = envelope.getHeader();
SOAPBody body = envelope.getBody();
```

Before we can start adding new content to the message, we have to briefly discuss another topic – **names.** Elements in an XML document have a name that is defined within a namespace. Which means that each name has two parts to it – the actual name, and the name of the namespace in which it is defined. When we add new elements to our SOAP message, we need to give them fully qualified names. Note that the names do not *have* to be fully qualified, but it is recommended that we always create new elements that way. JAXM provides the `javax.xml.soap.Name` interface to describe the name of an element.

The steps for adding a new element to a SOAP envelope are:

- Obtain the `javax.xml.soap.SOAPEnvelope` object
- Get the `javax.xml.soap.SOAPHeader` or `javax.xml.soap.SOAPBody` object from it, as appropriate
- Use the `javax.xml.soap.SOAPEnvelope.createName()` method to create a name for the new element
- Add a new element using either the `javax.xml.soap.SOAPHeader.addHeaderElement()` method or the `javax.xml.soap.SOAPBody.addBodyElement()` method
- Add content to the new element as needed

Let us build an example for this. Here, we will assume that we want to create a request message for the stock quote example that we have worked with in the earlier chapters.

Note The stock quote service is a synchronous service, thus it is not a perfect example here. However, since we are familiar with its interface, we will reuse it here to make it easier to understand the code. More specifically, we will send an asynchronous message to our service and simply ignore the (synchronous) response.

Try It Out: Populating the SOAP Message

This example populates a SOAP message, which can then be sent to the stock quote web service we used before. The creation of that message is similar to what we have shown in the previous example above:

1. Save this code in a file called as `TestJAXM3.java`:

```
import javax.xml.messaging.*;
import javax.xml.soap.*;

import com.sun.xml.messaging.jaxm.soaprp.*;

public class TestJAXM3 {

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

    ProviderConnectionFactory factory =
      ProviderConnectionFactory.newInstance();
    ProviderConnection connection = factory.createConnection();

    ProviderMetaData metaData = connection.getMetaData();
    String[] supportedProfiles = metaData.getSupportedProfiles();
    String profile = null;

    for(int i = 0; i < supportedProfiles.length;i++) {
      if(supportedProfiles[i].equals("soaprp")) {
        profile = supportedProfiles[i];
        break;
      }
    }

    MessageFactory msgFactory = connection.createMessageFactory(profile);
    SOAPMessage message = msgFactory.createMessage();

    SOAPPart part = message.getSOAPPart();
    SOAPEnvelope envelope = part.getEnvelope();
    SOAPBody body = envelope.getBody();

    // set up some namespace declarations in the envelope
    envelope.addNamespaceDeclaration("xsd",
      "http://www.w3.org/2001/XMLSchema");
    envelope.addNamespaceDeclaration("xsi",
      "http://www.w3.org/2001/XMLSchema-instance");
    envelope.addNamespaceDeclaration("SOAP-ENC",
      "http://schemas.xmlsoap.org/soap/encoding/");

    // build the element in the body: <getQuote>...</getQuote>
    Name name1 = envelope.createName("getQuote");
    SOAPBodyElement bodyElement = body.addBodyElement (name1);

    // build the parameter element: <arg0 xsi:type="xsd:string">IBM</arg0>
    SOAPFactory soapFactory = SOAPFactory.newInstance();
    SOAPElement arg0 = soapFactory.createElement("arg0");
    Name name3 = soapFactory.createName("type", "xsi",
      "http://www.w3.org/2001/XMLSchema-instance");
    arg0.addAttribute(name3, "xsd:string");
    arg0.addTextNode("IBM");

    //add the parameter element to the getQuote element
    bodyElement.addChildElement(arg0);

    message.writeTo(System.out);
  }
}
```

2. Compile and run the code:

```
javac TestJAXM3.java
java TestJAXM3
```

Here is the output from the example:

How It Works

In this example, we create a new `SOAPMessage` object as we have done before. This object will already have a `SOAPPart` and a `SOAPEnvelope` with empty body and header elements.

From the `SOAPMessage` object, we retrieve the `javax.xml.soap.SOAPPart` and `javax.xml.soap.SOAPEnvelope` objects. The `javax.xml.soap.SOAPEnvelope` allows us to obtain the `javax.xml.soap.SOAPBody` element, like so:

```
SOAPPart part = message.getSOAPPart();
SOAPEnvelope envelope = part.getEnvelope();
SOAPBody body = envelope.getBody();
```

Next, we define the namespace declarations for the envelope. These are standard definitions that we can always add to our envelope:

```
envelope.addNamespaceDeclaration("xsd",
  "http://www.w3.org/2001/XMLSchema");
envelope.addNamespaceDeclaration("xsi",
  "http://www.w3.org/2001/XMLSchema-instance");
envelope.addNamespaceDeclaration("SOAP-ENC",
  "http://schemas.xmlsoap.org/soap/encoding/");
```

The next two lines add the `<getQuote>` element to the body. This is a simple element with only a local (not fully qualified) name. The `addBodyElement()` method returns a reference to the new element, which will use later:

```
Name name1 = envelope.createName("getQuote");
SOAPBodyElement bodyElement = body.addBodyElement(name1);
```

To create a new element within the `<getQuote>` element, we need an instance of `javax.xml.soap.SOAPFactory`. This factory object provides methods to create new XML artifacts needed within a body or header:

```
SOAPFactory soapFactory = SOAPFactory.newInstance();
```

The `<arg0>` element is again built with a local name only:

```
SOAPElement arg0 = soapFactory.createElement("arg0");
```

This new element, `<arg0>` has one attribute and one text node inside of it. The attribute is fully namespace qualified, which means that we have to create another `javax.xml.soap.Name` object. Adding the attribute and the text node is straightforward:

```
Name name3 = soapFactory.createName("type", "xsi",
  "http://www.w3.org/2001/XMLSchema-instance");
arg0.addAttribute(name3, "xsd:string");
arg0.addTextNode("IBM");
```

What happens here is that the SOAP factory creates a `Name` object first, before adding the new attribute to the element. The `type` parameter represents the name of the parameter that is set to the web service. The `xsi` parameter serves at the prefix for all entries down that are scoped to the same namespace. Finally, http://www.w3.org/2001/XMLSchema-instance is the URI (note that this is not necessarily a valid URL) that is used to uniquely identify elements that are part of the XML Schema data-types namespace.

Sending the Message

Now we have a complete SOAP message that we can send to the target. In the asynchronous case the message is not directly sent to the target. Instead, it gets forwarded to the provider, which then delivers it to the final destination. How this is done depends on the selected profile.

Try It Out: Sending the Message

Here is the complete source for the test application that sends the message.

1. Save the listing given below in a file called `TestJAXMAsync.java`:

```
import javax.xml.messaging.*;
import javax.xml.soap.*;
```

```
import com.sun.xml.messaging.jaxm.soaprp.*;

public class TestJAXMAsync {

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

    ProviderConnectionFactory factory =
      ProviderConnectionFactory.newInstance();
    ProviderConnection connection = factory.createConnection();

    ProviderMetaData metaData = connection.getMetaData();
    String[] supportedProfiles = metaData.getSupportedProfiles();
    String profile = null;

    for (int i = 0; i < supportedProfiles.length; i++) {
      if (supportedProfiles[i].equals ("soaprp")) {
        profile = supportedProfiles[i];
        break;
      }
    }
    MessageFactory msgFactory = connection.createMessageFactory(profile);
    SOAPRPMessageImpl message =
      (SOAPRPMessageImpl)msgFactory.createMessage();

    message.setFrom(
      new Endpoint ("http://localhost:8080/someclient"));
    message.setTo(new Endpoint ("http://localhost:8080/myservice"));

    SOAPPart part = message.getSOAPPart();
    SOAPEnvelope envelope = part.getEnvelope();
    SOAPBody body = envelope.getBody();

    //set up some namespace declarations in the envelope
    envelope.addNamespaceDeclaration ("xsd",
      "http://www.w3.org/2001/XMLSchema");
    evelope.addNamespaceDeclaration ("xsi",
      "http://www.w3.org/2001/XMLSchema-instance");
    evelope.addNamespaceDeclaration("SOAP-ENC",
      "http://schemas.xmlsoap.org/soap/encoding/");

    //build the element in the body: <getQuote>...</getQuote>
    Name name1 = envelope.createName ("getQuote");
    SOAPBodyElement bodyElement = body.addBodyElement(name1);

    //build the parameter element: <arg0 xsi:type="xsd:string">IBM</arg0>
    SOAPFactory soapFactory = SOAPFactory.newInstance();
    SOAPElement arg0 = soapFactory.createElement ("arg0");
    Name name3 = soapFactory.createName ("type", "xsi",
      "http://www.w3.org/2001/XMLSchema-instance");
    arg0.addAttribute (name3, "xsd:string");
    arg0.addTextNode ("IBM");

    //add the parameter element to the getQuote element
    bodyElement.addChildElement (arg0);

    message.writeTo(System.out);
    connection.send(message);
  }
}
```

2. The provider must be configured properly to understand and process messages that are sent to it. The provider is configured through a file called `provider.xml`. It contains entries that define the protocol used for communication (that is, HTTP), plus additional entries for each target that we will send a message to.

   The file can be found at `%CATLINA_HOME%\webapps\jaxm-provider\WEB-INF`. We need to configure a new endpoint to match that which we defined in the class above (`("http://localhost:8080/myservice")`). To do this add a new `<Endpoint/>` element to the `provider.xml` file, like so:

   ```
   <?xml version="1.0" encoding="ISO-8859-1"?>
   ```

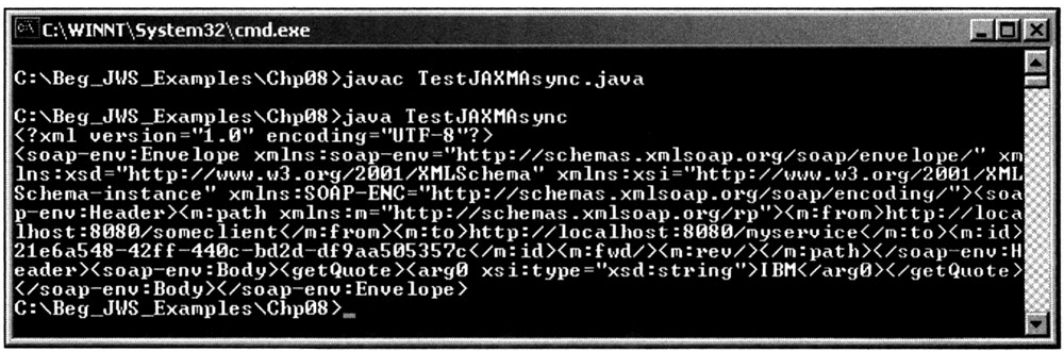
```
<!DOCTYPE ProviderConfig
  PUBLIC "-//Sun Microsystems, Inc.//DTD JAXM Provider//EN"
  "http://java.sun.com/xml/dtds/jaxm_provider_1_0_.dtd">

<ProviderConfig>
<!-profile definition for 'ebxml' left out here -->
  <Profile profileId="soaprp">
    <Transport>
    <Protocol>
      http
    </protocol>

    <Endpoint>
      <URI>
        http://www.wombats.com/soaprp/sender
      </URI>
      <URL>
        http://127.0.0.1:8080/jaxm-provider/receiver/soaprp
      </URL>
    </Endpoint>

    <Endpoint>
      <URI>
        http://localhost:8080/myservice
      </URI>
      <URL>
        http://localhost:8080/axis/services/StockQuote
      </URL>
    </Endpoint>

    <ErrorHandling>
       <Retry>
          <MaxRetries>
           3
          </MaxRetries>
          <RetryInterval>
           2000
          </RetryInterval>
       </Retry>
    </ErrorHandling>
...
</ProviderConfig>
```

3. Stop and restart Tomcat.

4. Now compile and run the code:

```
javac TextJAXMAsync.java
java TestJAXMAsync
```

   Here is what the output on the screen will look like:



   In the Tomcat window you should see the message being processed:

How It Works

Most of the code for this example is exactly as we have described before. What is new here is that we define the two endpoints for our communication with the stock quote service, namely the originator and the destination. As shown in the code extract below, the `com.sun.xml.messaging.jaxm.soaprp.SOAPRPMessageImpl` class defines two attributes for this:

```
MessageFactory msgFactory = connection.createMessageFactory(profile);
SOAPRPMessageImpl message = (SOAPRPMessageImpl)msgFactory.createMessage();

message.setFrom(new Endpoint ("http://localhost:8080/someclient"));
message.setTo(new Endpoint("http://localhost:8080/myservice"));
```

Note that, in this case we use a dummy originator address, because we cannot receive any response messages anyway. We had already defined this dummy receiver in the `client.xml` file when we configured the client. The code also shows that the target address, `http://localhost:8080/myservice` is `http://localhost:8080/axis/services/StockQuote`.

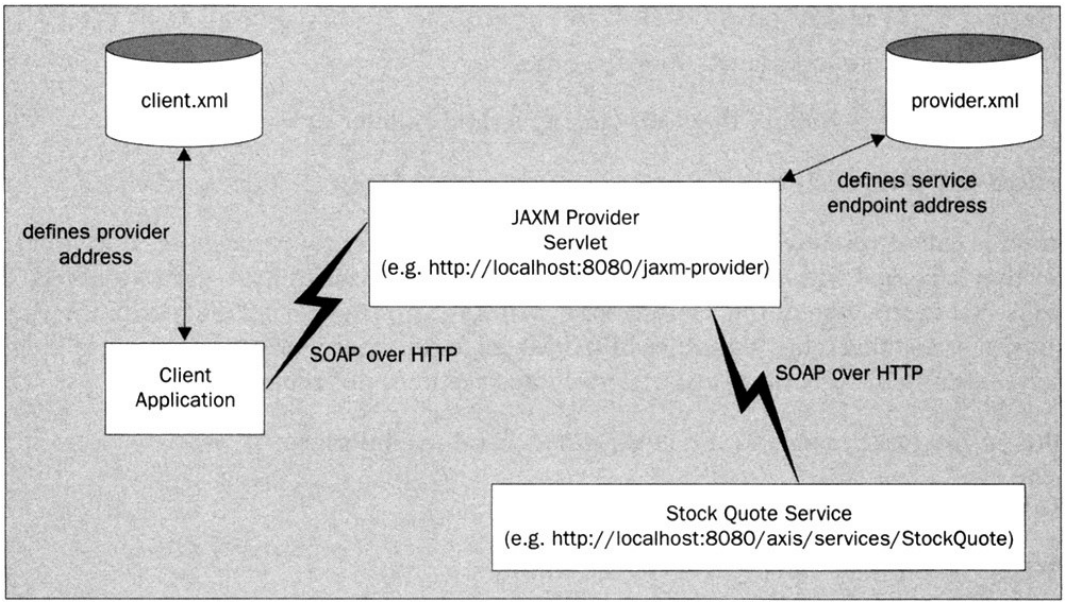
The `provider.xml` file configures a profile called `soaprp` for this provider. Within the profile definition, it specifies that an endpoint URI named `http://localhost:8080/myservice` is to be translated into a concrete URL, namely `http://localhost:8080/axis/services/StockQuote`. This allows for redirecting messages to different endpoint URLs without having to change any code.

Another interesting setting here is the `<ErrorHandling>` element:

```
<ErrorHandling>
   <Retry>
      <MaxRetries>
       3
      </MaxRetries>
      <RetryInterval>
       2000
      </RetryInterval>
   </Retry>
</ErrorHandling>
```

It specifies that the provider should try to send a message three times if necessary to the target. Note that the client will not be blocked for the duration of this; after all, we are making an asynchronous call.

The following diagram describes how a client finds a provider and sends a message to it:



The provider then delivers the message to the destination based on the information found in the `provider.xml` file.

We are finally ready to send the message! The following line of code shows that we can use the `send()` method on the `javax.xml.messaging.ProviderConnection` object to do that:

```
connection.send(message);
```

Note Note that, the SOAP `<Header>` element contains additional fields, which will tell the provider where to send the message.

Now that we know how to send a message asynchronously using a provider, we can look at setting up an application to receive the message.

Receiving Asynchronous Messages

The only protocol for asynchronous communication that is supported in the Java XML Pack, Summer 02, is HTTP. In other words, we can only send and receive messages over HTTP. The typical receiver of an HTTP request is a servlet. As a result, our receiver application will be wrapped in a servlet.

In a more realistic example, with a more sophisticated messaging provider, we would most likely use JMS as the communication vehicle for the message exchange. In the next section, we will be talking about the use of JMS in such a scenario. For now, let us just say that a service, which is based on JMS and running in a J2EE application server, would have an MDB as its access point.

In this case, however, we are using HTTP. Even though HTTP is a synchronous protocol, the use of the provider decouples the receiving and processing of a message from the sending of that message. We have seen in the previous section that the message is handed off to the provider without waiting for any response to come back. So how does the provider forward the message to its final destination? We already mentioned that a servlet would receive the message. So, the provider will invoke another HTTP request to the destination servlet's URL.

JAXM defines two listener interfaces to standardize the way messages are received:

- `javax.xml.messaging.ReqRespListener`
- `javax.xml.messaging.OnewayListener`

They are pretty similar – both of them provide a method called `onMessage()`.

Note A method with this name also exists on every message-driven bean.

This method is called whenever a message arrives at the receiver for processing. In both cases, the parameter that is passed with this message is a `javax.xml.soap.SOAPMessage` object. The only difference is that in the case of the `javax.xml.messaging.ReqRespListener` interface, the `onMessage()` method returns another SOAPMessage (in other words, the response), whereas the `javax.xml.messaging.OnewayListener` interface does not return anything.

Here is the `javax.xml.messaging.ReqRespListener` interface:

```
package javax.xml.messaging;

import javax.xml.soap.SOAPMessage;

public interface ReqRespListener {
  public SOAPMessage onMessage (SOAPMessage message);
}
```

and here is the `javax.xml.messaging.OnewayListener` interface:

```
package javax.xml.messaging;

import javax.xml.soap.SOAPMessage;

public interface OnewayListener {
  public void onMessage (SOAPMessage message);
}
```

Note that both of the interfaces are there for reasons of convenience; we are not required to use them. However, our code will be more portable across different JAXM implementations if we take advantage of them. Plus, if we don't use these interfaces, we have to define our own.

Another convenience class that JAXM provides for receiving JAXM messages over HTTP, is the `javax.xml.messaging.JAXMServlet` class. This class provides a default implementation for the `doPut()` method that will extract a `javax.xml.soap.SOAPMessage` from the incoming HTTP request and forward it to the `onMessage()` method. The `javax.xml.soap.SOAPMessage` object is built using a `javax.xml.soap.MessageFactory` object, just like we used before when we sent a message.

So, how do we use this servlet class for implementing our own receiver? We can create a new class that inherits from `javax.xml.messaging.JAXMServlet` and implement either the `javax.xml.messaging.ReqRespListener` or the `javax.xml.messaging.OnewayListener` interface, depending on whether a response will be provided or not. In our example, we will create a servlet that will not return any response to the caller, thus it will implement the `javax.xml.messaging.OnewayListener` interface, like so:

```
import javax.servlet.*;
import javax.servlet.http.*;

import javax.xml.messaging.*;
import javax.xml.soap.*;

import com.sun.xml.messaging.jaxm.soaprp.*;

public class MyReceiverServlet extends JAXMServlet implements OnewayListener
```

For a closer look at what goes into the servlet class, let's walk through an example.

Try It Out: Receiving Asynchronous Messages

Here is an example of a servlet that can receive asynchronous JAXM messages:

1. Save the listing shown below in a file called `MyReceiverServlet.java`:

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.xml.messaging.*;
import javax.xml.soap.*;
import com.sun.xml.messaging.jaxm.soaprp.*;

public class MyReceiverServlet extends JAXMServlet
    implements OnewayListener {

  private ProviderConnectionFactory pcf;
  private ProviderConnection pc;

  public void init (ServletConfig servletConfig) throws ServletException {
    super.init (servletConfig);

    try {
      pcf = ProviderConnectionFactory.newInstance();
      pc = pcf.createConnection();
      setMessageFactory (new SOAPRPMessageFactoryImpl());
    } catch (JAXMException e) {
      throw new ServletException (e.getMessage());
    }
  }

  public void onMessage (SOAPMessage message) {
    try {
      message.writeTo (System.out);
    } catch (Exception e) {
```

```
            System.out.println ("Exception occurred! " + e.getMessage());
        }
      }
    }
```

2. In order to compile this class we will need to add to our classpath the `javax.servlet` packages that can be found within Tomcat:

```
javac -classpath %classpath%;%CATALINA_HOME%\common\lib\servlet.jar
      MyReceiverServlet.java
```

3. We will also need to deploy this servlet into Tomcat in order for it to run, so we'll create a simple web application for this purpose. Create a `WEB-INF` directory and place the following `web.xml` file within it:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <servlet>
        <servlet-name>
            MyReceiverServlet
        </servlet-name>
        <servlet-class>
            MyReceiverServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>
            MyReceiverServlet
        </servlet-name>
        <url-pattern>
            /receiver
        </url-pattern>
    </servlet-mapping>
</web-app>
```

4. Copy the compiled `MyReceiverServlet` class into a `\classes` subdirectory of `WEB-INF` so we have the following structure:

```
\WEB-INF
        \web.xml
        \classes
                \MyReceiverServlet.class
```

5. Now use the `jar` command to create a WAR file from the top-level directory containing the WEB-INF folder:

```
jar -cf Ch08.war WEB-INF
```

6. Then copy the `Ch08.war` file into Tomcat's `\webapps` directory.

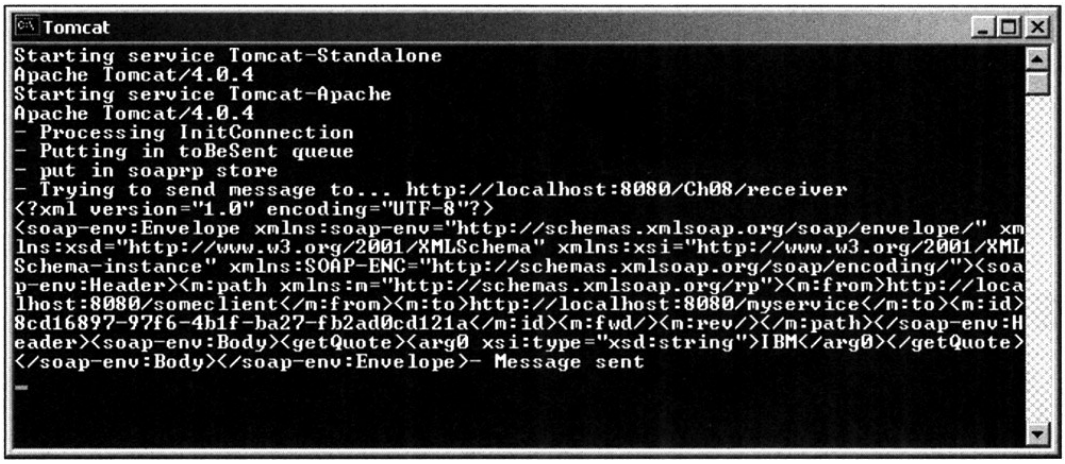
7. Before we can restart Tomcat we also need to configure the correct destination URL for the receiver in the `provider.xml` file. This will make sure that the JAXM provider will deliver the message to the correct URL. Modify the `<Endpoint/>` element we added before as follows:

```
        <Endpoint>
          <URI>
            http://localhost:8080/myservice
          </URI>
          <URL>
            http://localhost:8080/Ch08/receiver
          </URL>
        </Endpoint>
```

8. Restart Tomcat and then rerun our `TestJAXMAsync` client from before:

```
java TestJAXMAsync
```

We'll get the same result in the client window, but now in Tomcat's window we will see:



How It Works

For the servlet to be able to receive messages and build `javax.xml.soap.SOAPMessage` objects from it, it must create a `javax.xml.messaging.ProviderConnection` object and derive a `javax.xml.soap.MessageFactory` object from it. We only need to do this once for the lifetime of the servlet, so we will use the servlet's `init()` method to create these objects.

```
private ProviderConnectionFactory pcf;
private ProviderConnection pc;

public void init (ServletConfig servletConfig) throws ServletException {
  super.init (servletConfig);
  try {
    pcf = ProviderConnectionFactory.newInstance();
    pc = pcf.createConnection();
    setMessageFactory (new SOAPRPMessageFactoryImp1());
  } catch (JAXMException e) {
    throw new ServletException (e.getMessage());
  }
}
```

One interesting statement to note here is:

```
setMessageFactory(new SOAPRPMessageFactoryImpl());
```

As we mentioned earlier, the `javax.xml.messaging.JAXMServlet` convenience class creates a `javax.xml.soap.SOAPMessage` from the incoming HTTP request. It needs a `javax.xml.soap.MessageFactory` object to do that. Our servlet class uses the `setMessageFactory()` method to set this factory object.

The only other thing we need is the `onMessage()` method implementation. Here is where we process the incoming message:

```
public void onMessage (SOAPMessage message) {
  try {
    message.writeTo(System.out);
  } catch (Exception e) {
    System.out.println("Exception occurred! " + e.getMessage());
  }
}
```

Web Services over JMS

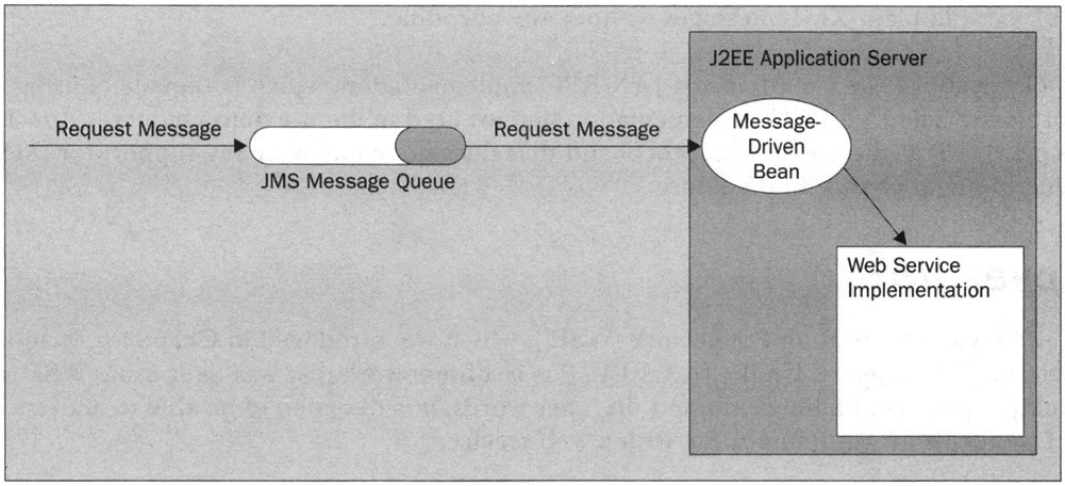
So far, we have covered the asynchronous innovation of web services. More specifically, we have looked at how to asynchronously send a SOAP-formatted message and how to receive one.

This sounds exactly like the description of a scenario that uses the Java Message Service (JMS). JMS provides interfaces for sending and receiving messages, both synchronously and asynchronously. JMS is just an interface (we can think of it as an API on top of messaging middleware), and it is independent of any communication protocol that is used to implement it. J2EE application servers typically come with an embedded JMS provider, that is, code that implements the JMS interface.

Thus, for asynchronous web services, using JMS as the underlying messaging infrastructure makes perfect sense; even more so, since the J2EE Enterprise JavaBean specification describes so called Message-Driven Bean. Explaining exactly what an MDB is and how it works goes well beyond the scope of this book; however, we can think of it as a message receiver that sits there and waits for messages to arrive on a queue or a topic. As soon as a message arrives, it will be forwarded to the MDB, which can then process it in a transactional manner.

Since we need to discuss it again, the following diagram shows again how this works (recall we looked at this picture at the beginning of this

chapter):



JAXM over JMS

Now that we've developed our own sender and receiver applications using JAXM, the previous diagram should start making a lot of sense to us. Instead of creating a servlet that accepts HTTP requests with the SOAP message as its content, we deploy an MDB into the J2EE container.

While the deployment of these solutions is different, the concept is the same. We can easily imagine how we could develop a provider for JAXM that takes messages from a client and sends them to a JMS queue or topic. The destination would then be this queue's name instead of a URL. From the queue, an MDB reads the request and forwards it to the actual receiver.

Thus, the MDB would play exactly the same role that our `javax.xml.messaging.JAXMServlet` played in the previous section. Neither the client sending the request, nor the web service that receives the message would look any different, since all of this handling is hidden in the provider.

However, the reference implementation for JAXM does not come with a JMS provider. The most obvious starting point to look for a JAXM over JMS provider is for us to check with the JMS implementation and J2EE application server vendors. JAXM is relatively new, so we can expect to see more products coming out that support it in the near future.

JAX-RPC over JMS

Another option to take advantage of the asynchronous capabilities of JMS would be to use the JAX-RPC interfaces, which we have described in Chapter 6. JAX-RPC supports one-way style operations – for example, the `javax.xml.rpc.Call` class offers a method called `invokeOneWay()`, which takes a number of input parameters and returns no response. It also takes an arbitrary string as the target endpoint address (through the `setTargetEndPointAddress()` method), so that we can pass a JMS queue name instead of a URL there.

However, note that in the case of JAX-RPC, it is assumed that we are invoking a remote function (as opposed to sending a message). This means the request message is always assumed to contain a number of (potentially encoded) parameters for the remote function invocation. JAXM, on the other hand, allows us to build plain XML messages without any encoding.

On top of the above, we would need a JAX-RPC implementation, which is capable of using JMS as a transport layer. The JAX-RPC implementation that we used in the previous chapter – Apache Axis – focuses on HTTP as the transport protocol and thus does not come with any support for JMS. However, this scenario might change in the future.

WSIF over JMS

The Web Services Invocation Framework (WSIF), which we introduced in Chapter 6, is another candidate for JMS support. Unlike JAX-RPC, this is a framework that was built using WSDL documents with multiple protocol bindings in mind. In other words, it is designed to be able to address different kinds of protocols for communicating with a web service.

Presently, WSIF does not define protocol bindings for JMS; however, this can be expected to change soon. WSIF is in the process of being turned into an open source project by Apache, and support for JMS is already in the pipeline. As in the case of JAX-RPC, this is unlikely to result in a client API change. What will change is the runtime implementation, which will process and interpret JMS based bindings.

This can happen in two ways – through SOAP over JMS or native JMS.

Soap over JMS

This style of JMS-based web service invocation will use SOAP envelopes as the message format between a requester and the provider of a service. In this respect, it will be very similar to the way web services invocations are done today – a SOAP message is built, with all of the

header and body elements that are needed, but instead of being sent to a destination via HTTP, it is sent to a JMS queue.

Native JMS

In this case, there is no SOAP. Instead, all the message elements are serialized into a binary format and sent to the JMS queue, without ever being turned into an XML-based message. This style is likely be a lot faster than using SOAP, since no XML parsing is necessary. A binary object representation will almost always be faster than the readable, tagged XML format.

However, this also requires that both the requester and the provider of a service agree on how the message is serialized. If we assume that the requester is a Java application, it will create serialized Java objects. This normally means that we also need a Java program to deserialize these objects and turn them into something the web service can understand.

In other words, this style of JMS-based web service implementation, while being faster, requires Java code on both sides of the equation.

WSIF plans to support both styles in the near future, so check with the WSIF site at http://cvs.apache.org/viewcvs.cgi/xml-axis-wsif/for more details on this.

J2EE Web Services over JMS

Both JAX–RPC and WSIF focus on the requester of a web service. WSIF exclusively talks about the requester, without providing any support for hosting or provisioning of a service. JAX–RPC defines how a web service is mapped into a Java interface, but it does not define how a service is deployed into a run time environment. This is left to the implementation.

At the time of this writing, work is under way to define how a J2EE application container handles web services. This work is happening as part of the Java Specification Request (JSR) number 109. For the latest draft of the JSR 109 specification, go to http://jcp.org/jsr/detail/109.jsp. In its first version, this JSR will not address how to host JMS-based web services, but this will change over time. However, for the time being all J2EE application server vendors will probably come up with their own way of supporting JMS for asynchronous web services.

Summary

In this chapter, we have seen how web services can be invoked asynchronously. This is very useful in scenarios where the requester does not depend on a response being returned right away (if at all), and/or when the service execution takes a very long time. Allowing the requester to do other work in the meantime provides for better optimization of computing resources.

This leads to a programming model that is different from the one that we had discussed so far. Here, systems are more loosely coupled than in the synchronous case, namely a ‘message-based’ web services programming model that takes the interaction between applications as an exchange of messages rather than as an invocation of a remote function.

The Java API for XML-based Messaging (JAXM) is an effort to standardize this programming model for the Java world. It supports both a synchronous and an asynchronous messaging model. The asynchronous model requires the use of a so-called provider, which is responsible for the delivery of a message to its final destination. JAXM assumes the use of SOAP-formatted messages. The SOAP-dependent part of the API was split off into a separate specification, called the SOAP with Attachment API for Java (SAAJ). Both JAX-RPC and JAXM depend on SAAJ-JAX-RPC for synchronous, RPC-style interaction and JAXM for asynchronous, message-style interactions.

We have seen an example of a client sending a SOAP message via the JAXM API, as well as an example of receiving a JAXM message over HTTP.

Finally, we discussed how the Java Message Service (JMS) could be used as the transport layer for message-based asynchronous web service communication. This can happen on various levels and in various ways, be it as an additional provider underneath JAXM, or be it as another binding element in WSDL, which can then be interpreted by a WSIF client. JMS can be used both with and without formatting a message in SOAP. While SOAP is more flexible because of its standardized, self-describing nature, a binary (or native) format will most likely be faster and more efficient.