Covers Java Web Services Developer Pack

JAVA™ WEB SERVICES IN A NUTSHELL

A Desktop Quick Reference

O'REILLY®

Kim Topley

JAVA™ WEB SERVICES

IN A NUTSHELL

Other Java™ resources from O'Reilly

Related titles

Enterprise JavaBeans™
Java™ & XML
Java™ Cookbook
Java™ Enterprise in a Nutshell
Java™ I/O
Java™ in a Nutshell
Java™ Performance Tuning
Java™ Programming with Oracle SQLJ
Java™ Security
JavaServer™ Pages
Java™ Swing
Learning Java™

Java Books Resource Center

java.oreilly.com is a complete catalog of O'Reilly's books on Java and related technologies, including sample chapters and code examples.

OnJava.com is a one-stop resource for enterprise Java developers, featuring news, code recipes, interviews, weblogs, and more.

Conferences

O'Reilly & Associates bring diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

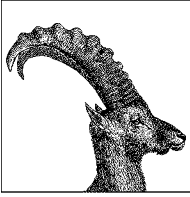
JAVA™ WEB SERVICES

IN A NUTSHELL

Kim Topley

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo



3
SAAJ

In the last chapter, you saw how to use JAX-RPC to create web services based around the use of remote procedure calls. Even though JAX-RPC uses an XML-based protocol, there was hardly any XML in evidence in Chapter 2. For the most part, JAX-RPC does a very good job of hiding the details of messages that are exchanged between a web service and its client during a remote procedure call. However, most of the more advanced JAX-RPC features require an understanding of SOAP, the XML-based protocol on which JAX-RPC in particular, and web services in general, are based. This chapter paves the way for a more in-depth examination of JAX-RPC by introducing SOAP, together with SAAJ, which is a Java API for creating, sending, and receiving SOAP messages.

Introduction to SAAJ

SAAJ, which stands for *SOAP with Attachments API for Java*, provides a convenient API for constructing SOAP messages without having to directly create the XML yourself. SAAJ was originally part of the *Java API for XML Messaging* (JAXM), which was developed by the JSR 67 expert group. The final release of this specification (JAXM Version 1.0) provided two different but related facilities:

- Core functionality concerned with manipulating SOAP messages in a generic way, together with the ability to send a SOAP message from one entity to another.
- A higher-level messaging facility that included reliable delivery of messages and support for *messaging profiles*, which require SOAP messages to be constructed in specific ways.

During the maintenance cycle for JSR 67, it was decided to unbundle the low-level SOAP message creation features into a separate specification, thus creating SAAJ 1.1, leaving the higher-level features to form JAXM Version 1.1. At the same time, minor modifications were made to the API to remove a dependency that would

otherwise have had the undesirable effect of making SAAJ 1.1 dependent on JAXM 1.1. The result is that it is possible to use SAAJ as a lightweight library for building and exchanging SOAP messages, without requiring the inclusion of JAXM 1.1, which provides facilities that go beyond the requirements of many web service clients. JAX-RPC, in particular, uses SAAJ to construct and decode SOAP messages, but it does not require reliable messaging and therefore is not dependent on the presence of a JAXM implementation.

The formal specifications for both SAAJ 1.1 and JAXM 1.1 can be downloaded from *http://jcp.org/jsr/detail/67.jsp*. This chapter looks only at SAAJ and uses it as a convenient means of introducing SOAP messages; the more advanced messaging facilities provided by JAXM are covered in Chapter 4.

SAAJ Programming

Whereas JAX-RPC is concerned with allowing a client program to make a remote procedure call to a web service without exposing the underlying XML-based protocols that are used, SAAJ is a much lower-level API that is entirely concerned with the messages exchanged between the web service and its clients. Furthermore, while JAX-RPC applications look, for the most part, like ordinary Java applications making local method calls, an application that uses SAAJ needs to construct SOAP messages piece by piece and extract information from response messages. Using SAAJ requires much more work on the part of the developer than JAX-RPC, so why would you bother to use it? Here are some of the circumstances in which you might want to use SAAJ and its close relative JAXM instead of JAX-RPC:

- JAX-RPC is convenient for accessing web services that present an RPC-like interface. However, the RPC model is not suitable for all services. In many cases, it is more convenient to simply send an XML message to a service, which the service interprets and then generates an XML response. The most commonly quoted example of this is a business-to-business service where the client sends a purchase order in the form of an XML document to which the service responds with a confirmation and an invoice, also encoded as an XML document. A simple service like this does not require method calls or arguments—all that is necessary is to exchange fragments of XML. SAAJ represents a convenient way to encode and decode the SOAP messages that will carry these XML documents.
- Most services accessed using JAX-RPC are likely to be synchronous in nature, so that the service immediately processes the request and returns a reply to the client, which is blocked until the call completes. However, it is not always convenient or appropriate for the service to handle the request and reply immediately. Using the purchase order example again, the business that receives the purchase order may not be able to respond with a confirmation or an invoice immediately—perhaps not until the goods are in stock, or until it is verified that the initiator of the request has an account against which the goods can be ordered and that sufficient credit is available. For this type of business model, which is likely to be very common in the real world, it is more appropriate to think of the whole process as two separate operations that are not tightly coupled to each other and that the responding business

might take anything from several seconds to several days to reply. To implement this type of loosely coupled messaging, it is appropriate to use JAXM, which is built on top of SAAJ and provides the ability to send and receive messages asynchronously.

- JAX-RPC works only when the client and the service are active at the same time and also assumes that there is an available network path that directly connects them. If the service is not available when the client initiates the request, or there are good reasons (perhaps security-related) why the client cannot be directly connected to the server, then you can't use JAX-RPC. JAXM, on the other hand, provides a reliable delivery service without requiring the client application to be involved in how the reliability is provided and can also support routing of SOAP messages between hosts that cannot be directly connected.

SAAJ and JAXM provide a complete solution for XML-based messaging. The major differences between SAAJ and JAXM are as follows:

- SAAJ provides the API for the generic handling of SOAP messages; JAXM builds on this by adding the capability to create SOAP messages with preset content as required by messaging standards such as SOAP-RP and ebXML-TRP.
- SAAJ can be used to create a freestanding Java client that communicates directly with a web service. JAXM adds the concept of a messaging provider, which acts as an intermediary between the client and the eventual recipient of the message. The message provider can provide a reliable delivery service and can route messages to other intermediaries without the involvement of the client. JAXM clients that use a messaging provider must be hosted in a web container or an application server.
- SAAJ clients (and JAXM clients that choose not to use a messaging provider) can only engage in synchronous request/response message exchanges. However, JAXM clients using a messaging provider have access to additional message exchange modes, including asynchronous delivery and receipt of messages. See Chapter 4 for further details.

The classes and interfaces that provide the SAAJ and JAXM APIs reside in different packages, and, to emphasize the fact that SAAJ does not require JAXM, they are distributed in separate JAR files, as shown in Table 3-1.

Table 3-1. Packaging of the SAAJ and JAXM APIs

API	Package	JAR file
SAAJ	`javax.xml.soap`	*saaj-api.jar* (API) *saaj-ri.jar* (reference implementation)
JAXM	`javax.xml.messaging`	*jaxm-api.jar* (API) *jaxm-runtime.jar* (reference implementation)

In the case of J2EE 1.4, JAXM is not supported and the SAAJ classes appear in *lib\j2ee.jar*, which contains almost all of the packages in the reference implementation.

Although we have used the terms "client" and "server" to describe the participants in a SAAJ message exchange, the programming model for both SAAJ and JAXM does not make a strong distinction between these two roles, because most of the API is concerned with the details of handling the messages rather than actually sending and receiving them. In fact, SAAJ messaging represents more of a peer-to-peer model in which it might be more appropriate to use the terms "sender" and "receiver" instead of "client" and "server." In this chapter, however, all of the examples use SAAJ and are therefore limited to synchronous request/reply exchanges. For the sake of clarity, I will continue to use the term "client" to refer to the initiator of a service request, which will always be a freestanding J2SE application, and I will use "server" to mean the entity that receives and replies to the request.

For the JWSDP, to compile client applications that use SAAJ, your `CLASSPATH` needs to include only the *saaj-api.jar* file. However, the `CLASSPATH` required to run a client application is much larger, consisting of the following JAR files, which can be found in various subdirectories of the JWSDP installation:

saaj-api.jar	*saaj-ri.jar*	*activation.jar*	*commons-logging.jar*
dom4j.jar	*mail.jar*	*jaxp-api.jar*	*dom.jar*
sax.jar	*xalan.jar*	*xercesImpl.jar*	*xsltc.jar*

To run a SAAJ client application with J2EE 1.4, your `CLASSPATH` needs to include *lib\j2ee.jar*, together with the following four files from the endorsed directory:

dom.jar	*sax.jar*	*xalan.jar*	*xercesImpl.jar*

Aside from the handling of SOAP messages, SAAJ includes the ability to synchronously send a completed message to a given destination using HTTP as the underlying transport mechanism* and receive the reply (see "Creating and Sending a SOAP Message Using SAAJ," later in this chapter), but no specific provision is made for receiving a SOAP request in the server role. Servers are expected to reside in web containers or application servers, and need to make their own arrangements to receive SOAP messages. The JAXM specification includes a servlet (`javax.xml.messaging.JAXMServlet`) that can be used as the basis for a web container–based message receiver, but service providers are not required to use it. The example source code in this chapter uses a servlet that is very similar to `JAXMServlet`, so that we can demonstrate SAAJ without introducing a dependency on JAXM. This servlet is described later in this chapter in the section "Receiving a SOAP Message."

SOAP Messages

The format of SOAP messages is defined by a note submitted to the World Wide Web Consortium (W3C) in May 2000 by a group of companies including

* Support of HTTP as the transport mechanism for SOAP messages is mandatory for all SAAJ implementations. Vendors are free to provide other transport mechanisms, but are not required to do so.

Microsoft and IBM. This note, which describes Version 1.1 of SOAP and can be downloaded from *http://www.w3.org/TR/SOAP*, is not a formally adopted W3C specification but it is, nevertheless, the specification on which all existing SOAP implementations, including SAAJ, are based. The W3C is working on a formal definition of the next revision of SOAP, to be called Version 1.2. At the time of this writing, the SOAP 1.2 specification is available for public review, and Sun plans to include support for SOAP 1.2 in SAAJ (and JAX-RPC) when the specification is finalized.*

SOAP defines a way to wrap information represented in XML so that it can be transmitted between peer entities that know how to interpret that information and, presumably, act on it to provide a service. Other than the fact that the useful content of the message (which is known as the *payload*) must be encoded in XML, SOAP does not mandate a particular set of XML elements and attributes to be used to represent primitive items of data (such as integers, floating-point numbers, and strings) in terms of XML constructs, although it does specify an encoding mechanism for these data types and others, the use of which is encouraged. In practice, these encoding rules, commonly referred to as "SOAP section 5 encoding," have become a *de facto* standard and are the default encoding used by JAX-RPC to represent method arguments and return values. Although detailed coverage of these encoding rules is outside the scope this book, there are several examples of their use in this chapter; you'll find a complete discussion of the SOAP encoding rules in *Java and SOAP*, by Robert Englander (O'Reilly). Applications are free to use privately defined encodings instead of the SOAP section 5 rules if they wish.

What SOAP *does* specify is the overall structure of a message, together with rules for wrapping SOAP messages when the underlying transport protocol is HTTP. There are two slightly different ways to construct a SOAP message depending on whether the message has any attachments (the meaning of which is defined later). The SOAP Version 1.1 specification covers only the case where there are no attachments and requires that a message be constructed as shown in Figure 3-1. The outermost layer of the message is a protocol-specific wrapper, the nature of which is defined in the specification only for HTTP and which we'll see later in this chapter. Inside this wrapper is the SOAP message itself, consisting of an envelope, a header part, a body part, and optional additional content.

The SOAP *envelope* is, as its name suggests, a top-level XML element that acts as a container for the rest of the message. The SOAP header is optional but, if present, must be the first element in the envelope. It is intended to be used to carry information that can be used in the processing or routing of the message payload, such as a digital signature to guarantee the integrity of payload data or authentication information to validate the identity of the message sender. There are specifications, referred to in the JAXM documentation as profiles, that define specific SOAP headers to be used to communicate information to applications that are

* The acronym SOAP originally stood for "Simple Object Access Protocol." In SOAP Version 1.2, the letters no longer have any meaning. This is probably a good idea, because the original name is arguably not really an accurate description of what SOAP is: although the S and the P could be justified, there is really nothing in SOAP that provides object access, at least not in any sense that would be acceptable to an object-oriented programmer.

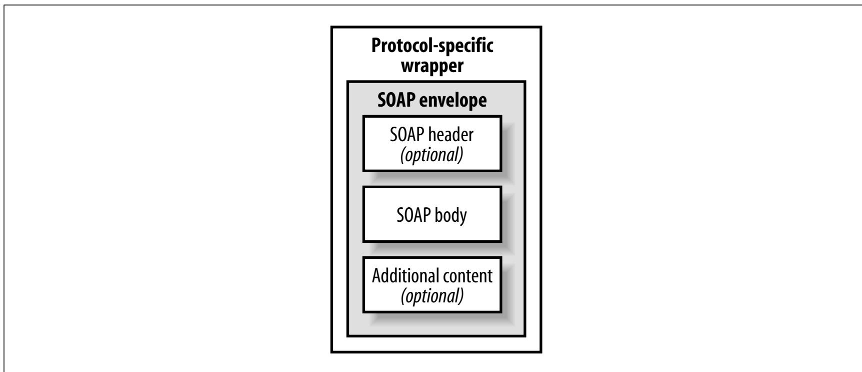


Figure 3-1. A SOAP message with no attachments

aware of the profile. The reference implementation of JAXM includes messaging providers that can create SOAP messages that have pre-installed headers as required by the SOAP-RP or ebXML TRP specifications. For further information on these two profiles, see Chapter 4.

The SOAP *body* is the only mandatory part of the envelope and contains the actual payload intended for the ultimate recipient of the message. It must either follow the SOAP header or, if the header is omitted, be the first element in the envelope. Following the body, it is possible to include additional content, the interpretation of which, like the payload itself, is entirely dependent on the sending and receiving entities.

Everything within the SOAP envelope *must* be encoded in XML. For many applications, this is not an acceptable restriction. For example, an online book store offering a web service interface might want to supply book details that include a photograph of the book's cover, scanned copies of pages from the book, or a sample chapter in the form of a PDF document. In order to enable applications such as this, Microsoft and Hewlett Packard created an additional specification called "SOAP Messages with Attachments," which can be downloaded from *http://www.w3.org/TR/SOAP-attachments*. Like the SOAP 1.1 document, this specification has been submitted to the W3C, and although it has not been approved by W3C, it has become the *de facto* standard for packaging SOAP messages that require some element of non-XML content.

A SOAP message that has attachments is formatted as shown in Figure 3-2. *Attachments* typically contain non-XML data, such as images, audio, or plain text. The first part of the message contains the SOAP envelope and its content, constructed as described by the SOAP 1.1 specification. Each object to be attached is then added as MIME content, and the whole message is packaged as a MIME `Multipart/Related` message as defined by RFC 2387 (which can be found at *http://www.ietf.org/rfc/rfc2387.txt*). Each separate message part has its own MIME wrapper that specifies, among other things, the type of data that it contains—it may contain an identifier that can be used within the SOAP envelope to refer to it. Using SAAJ, you can construct SOAP messages with attachments that contain any type of data that has a MIME encoding. This topic is discussed in more detail in "SOAP with Attachments," later in this chapter, where you'll also see exactly what a typical message with attachments looks like when bound to the HTTP protocol.

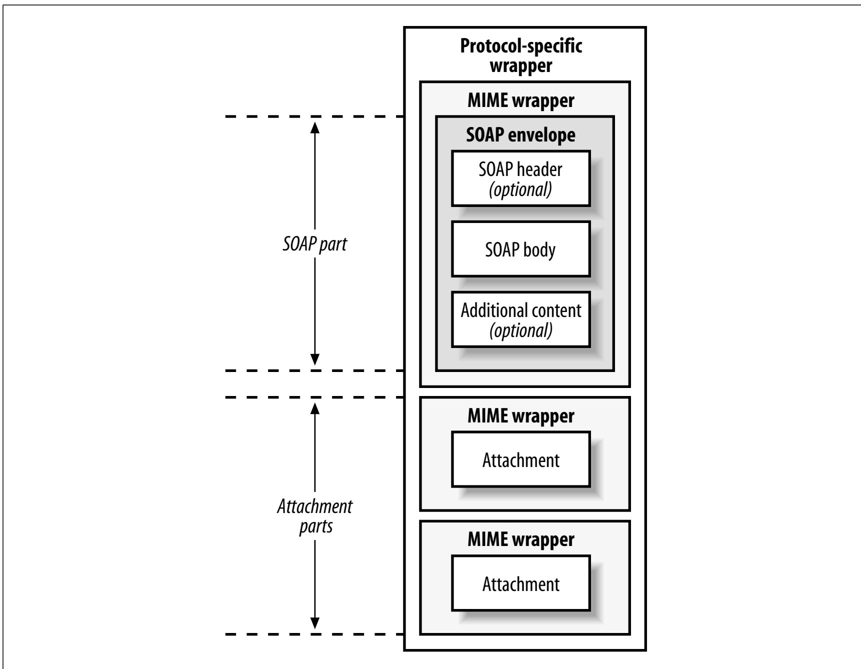


Figure 3-2. A SOAP message with two attachments

Creating and Sending a SOAP Message Using SAAJ

In order to allow vendors to supply their own implementations, almost all of the SAAJ API is made up of interfaces and abstract classes. In particular, the `SOAPMessage` class, which represents a SOAP message, is abstract and therefore cannot be directly instantiated. To create a SOAP message, you need to use a `MessageFactory`. `MessageFactory`, which is part of the `javax.xml.soap` package, is itself an abstract class, an instance of which can be obtained using its static `newInstance( )` method:

```
public static MessageFactory newInstance( ) throws SOAPException;
```

To permit vendors to plug in their own implementations of `MessageFactory`, this method looks at the system property `javax.xml.soap.MessageFactory` and in a couple of other places to select the subclass of `MessageFactory` that will be used. Refer to the description of `javax.xml.soap.MessageFactory` in the reference section of this book for a complete description of the steps taken to find a `MessageFactory`.

If a problem is encountered while instantiating the `MessageFactory`, a `SOAPException` is thrown. `SOAPException`, which is one of only four concrete classes in the `javax.xml.soap` package, is a checked exception that is thrown by many of the methods in the SAAJ API. It contains a message describing the reason for the exception and, for cases where the root cause is an exception thrown from a lower level, provides a `getCause( )` method to provide access to the original exception.

MessageFactory has two methods that create SOAP messages:

```
public SOAPMessage createMessage( ) throws SOAPException;
public SOAPMessage createMessage(MimeHeaders headers, InputStream is)
     throws SOAPException;
```

The second of these two methods is used typically in a servlet to deserialize a message received from an input stream, an example of which we'll see in "Receiving a SOAP Message," later in this chapter. To create a SOAP message for transmission, you need to use the first method:

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

The message that createMessage( ) returns does not contain anything useful, but rather than pause here to introduce the API needed to add some content to it, let's skip ahead a little and look at how to transmit the skeleton message. In order to send a message, you need an instance of the SOAPConnection class, which, since it is abstract, must be obtained from a factory:

```
SOAPConnectionFactory connFactory = SOAPConnectionFactory.newInstance( );
SOAPConnection conn = connFactory.createConnection( );
```

Just like MessageFactory, SOAPConnectionFactory is an abstract class that vendors can supply their own implementations of. The actual class that the newInstance( ) method uses is determined by a procedure that is virtually identical to that used by the MessageFactory class, and which is described in the reference section for javax.xml.soap.SOAPConnectionFactory, later in this book. If all else fails, the reference implementation returns its own implementation of SOAPConnectionFactory (or throws a SOAPException if this class is not available).*

Once you have a SOAPConnection, you can use its call( ) method to transmit a message:

```
public abstract SOAPMessage call(SOAPMessage request, Object destination)
throws SOAPException;
```

The destination object determines where the message will be sent. The SAAJ specification requires that the following types be supported as valid destination arguments:

- An instance of the java.net.URL class.
- An instance of String, provided that its value can be converted to a valid URL using its URL(String url) constructor.
- An instance of the class javax.xml.messaging.URLEndpoint. This class, which wraps a URL, is part of the JAXM API and is therefore not likely to be used by pure SAAJ applications. It is accepted as a valid destination because Version 1.0 of the JAXM specification, which did not have SAAJ separated out as a freestanding API, defined the destination parameter of this method as being

* Despite its name, SOAPConnection does not actually represent a connection to a message receiver. Instead, think of it as representing a connection between application code and the SAAJ runtime that can be used to send SOAP messages. In Chapter 4, we'll see that JAXM has a similar class, called ProviderConnection, which provides an association between a JAXM application and a messaging provider, but similarly does not imply that an immediate network connection is made.

of type `javax.xml.messaging.Endpoint`. This is an abstract base class of which `URLEndpoint` is the only concrete implementation; therefore, supporting a destination of type `URLEndpoint` provides backward-compatibility.

The reference implementation supports all of these possibilities; vendor implementations are free to add their own destination types as required. All of these destination types resolve to a URL, but the structure of this URL depends entirely on the implementation of the receiver and the environment in which it is hosted. We'll see a typical example in "Receiving a SOAP Message," later in this chapter.

As noted earlier in this section, a SAAJ client can only use a synchronous request/response programming model; therefore, the `call()` method blocks having sent the message until a reply sent to the message is received, or until an error causes it to throw a `SOAPException`. The reply is returned to the method caller. Having received a reply, if you don't need to make further use of the `SOAPConnection` object, you should use its `close()` method to release it; this method can be called only once. Once it is, any further invocations of `call()` method result in a `SOAPException`.

The example source code for this chapter includes a client that creates a SOAP message, prints its content, and then sends it to a servlet that echoes it straight back. To run this example, start your web server or application server, open a command window and make *chapter3\echoservice* your working directory, and then use the following command to build and deploy the servlet:

```
ant deploy
```

Next, compile and run the client application using the command:

```
ant compile-client run-client
```

In the command window, you'll see quite a lot of output, including the XML for the `SOAPMessage` that was transmitted (which has been reformatted to make it more readable):

```
1   <soap-env:Envelope xmlns:soap-env=
          "http://schemas.xmlsoap.org/soap/envelope/">
2         <soap-env:Header/>
3       <soap-env:Body/>
4   </soap-env:Envelope>
```

This is what the basic message returned by the `createMessage()` method looks like if you don't make any changes to it. Lines 1 and 4 contain the XML elements that represent the SOAP message envelope. As you can see, the element name is `Envelope` and it is qualified using the namespace prefix `soap-env`, which is associated with the URL *http://schemas.xmlsoap.org/soap/envelope*. This URL identifies the entire SOAP message as being formatted according to the rules of SOAP Version 1.1. If a SOAP message whose envelope is qualified with any other namespace is received, then it should be treated as a SOAP version mismatch, and the receiver is required to reject the message by generating a SOAP fault (as described in "SOAP Fault Handling," later in this chapter).

Inside the envelope are the header and body elements, for which the element names are `Header` and `Body`, respectively. These two elements are also qualified with the same namespace tag as the envelope. In this case, since we didn't actually add anything to the message returned by the `MessageFactory` `createMessage()`

method, the header and body parts are both empty. Under normal circumstances, if you do not need any header content, you completely remove the header part. You'll see how to do this later in this chapter.

SOAP messaging makes heavy use of XML namespaces and XML schema. A detailed discussion of these topics is outside the scope of this book and, although we'll explain the use of namespaces and schemas as we encounter them, for a full treatment of XML namespaces and XML schema, refer to *XML In a Nutshell,* Second Edition, by Elliotte Rusty Harold and W. Scott Means (O'Reilly).

Receiving a SOAP Message

The `SOAPConnection` class provides the `call()` method to allow a SOAP message to be transmitted, but there is no corresponding API that takes care of receiving a message. The example just shown uses a servlet as the target of the message. This is convenient because the `SOAPConnection` class uses HTTP as the default protocol when transmitting messages, and the servlet API contains everything that you need to handle a payload delivered over an HTTP connection, including a convenient API to handle the HTTP headers.

The `java.xml.messaging` package includes a servlet called `JAXMServlet` that can be used to receive and handle SOAP messages. However, since this package is part of the JAXM API, using it would introduce a dependency on JAXM as well as SAAJ, which is not desirable because JAXM is not part of the J2EE 1.4 platform. To avoid this, the servlet used in the `echoservice` example is based on a slightly simpler version of `JAXMServlet` that is provided in the sample code supplied with the JWSDP. The source code for this servlet can be found in the file *chapter3\servlet\ora\jwsnut\saaj\SAAJServlet.java* relative to the installation directory of the example code for this book. Most of the code is also shown in Example 3-1.

Example 3-1. A servlet that can receive SOAP messages

```
package ora.jwsnut.saaj;

import java.io.IOException;
import java.io.OutputStream;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.StringTokenizer;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.MimeHeader;
import javax.xml.soap.MimeHeaders;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPMessage;

/**
 * A servlet that can be used to host a SAAJ
```

Example 3-1. A servlet that can receive SOAP messages (continued)

```
 * service within a web container. This is based
 * on ReceivingServlet.java in the JWSDP tutorial
 * examples.
 */
public abstract class SAAJServlet extends HttpServlet {

    /**
     * The factory used to build messages
     */
    protected MessageFactory messageFactory;

    /**
     * Initialisation - create the MessageFactory
     */
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        try {
            messageFactory = MessageFactory.newInstance( );
        } catch (SOAPException ex) {
            throw new ServletException("Failed to create MessageFactory", ex);
        }
    }

    /**
     * Handles a POST request from a client. The request is assumed
     * to contain a SOAP message with the HTTP binding.
     */
    public void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

        try {
            // Get all the HTTP headers and convert them to a MimeHeaders object
            MimeHeaders mimeHeaders = getMIMEHeaders(request);

            // Create a SOAPMessage from the content of the HTTP request
            SOAPMessage message = messageFactory.createMessage(mimeHeaders,
                                                    request.getInputStream( ));

            // Let the subclass handle the message
            SOAPMessage reply = onMessage(message);

            // If there is a reply, return it to the sender.
            if (reply != null) {
                // Set OK HTTP status, unless there is a fault.
                boolean hasFault = reply.getSOAPPart().getEnvelope( )
                                        .getBody( ).hasFault( );
                response.setStatus(hasFault ?
                                   HttpServletResponse.SC_INTERNAL_SERVER_ERROR :
                                   HttpServletResponse.SC_OK);

                // Force generation of the MIME headers
                if (reply.saveRequired( )) {
```

SAAJ

Example 3-1. A servlet that can receive SOAP messages (continued)

```
                reply.saveChanges( );
            }

            // Copy the MIME headers to the HTTP response
            setHttpHeaders(reply.getMimeHeaders( ), response);

            // Send the completed message
            OutputStream os = response.getOutputStream( );
            reply.writeTo(os);
            os.flush( );
        } else {
            // No reply - set the HTTP status to indicate this
            response.setStatus(HttpServletResponse.SC_NO_CONTENT);
        }
    } catch (SOAPException ex) {
        throw new ServletException("SOAPException: " + ex);
    }
}

/**
 * Method implemented by subclasses to handle a received SOAP message.
 * @param message the received SOAP message.
 * @return the reply message, or <code>null</code> if there is
 * no reply to be sent.
 */
protected abstract SOAPMessage onMessage(SOAPMessage message)
    throws SOAPException;

// HEADER HANDLING CODE NOT SHOWN.....

}
```

When SAAJServlet receives a SOAP message, it hands it to the abstract onMessage( ) method and sends the reply message returned by this method to the message sender. To provide specific message handling, subclass SAAJServlet and implement the required processing in the onMessage( ) method.

The doPost( ) method of this servlet demonstrates how to receive a SOAP message from an HTTP connection. When transmitted over HTTP, the protocol-specific wrapper shown in Figure 3-1 and Figure 3-2 is represented as HTTP headers, and the rest of the message is written out as a stream of XML or, if the message has attachments, as a Multipart/Related MIME payload containing the XML and the encoded attachment data. Example 3-2 shows what an empty SOAP message looks like when bound into an HTTP request message.

Example 3-2. An empty SOAP message bound to the HTTP protocol

```
Content-Length: 17 1
SOAPAction: ""
User-Agent: Java1.4.0
Host: localhost:5050
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

Example 3-2. An empty SOAP message bound to the HTTP protocol (continued)

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
    <soap-env:Header/>
    <soap-env:Body/>
</soap-env:Envelope>
```

When this HTTP request message is delivered to a servlet, the HTTP headers are automatically stripped off and made available via the `HttpServletRequest` object, while the rest of the data, namely the XML itself, can be obtained by reading the `InputStream` provided by the `HttpServletRequest getInputStream( )` method. To convert the XML back into a `SOAPMessage`, the servlet uses a `MessageFactory` that it creates when it is initialized, and calls the second variant of the `createMessage( )` method that we saw in "Creating and Sending a SOAP Message Using SAAJ" earlier in this chapter, passing it the HTTP headers and the input stream containing the XML-encoded SOAP message:*

```
// Create a SOAPMessage from the content of the HTTP request
SOAPMessage message = messageFactory.createMessage(mimeHeaders, request.
getInputStream( ));
```

The first argument supplied to this method is an object of type `javax.xml.soap.MimeHeaders`, which encapsulates the HTTP headers that were received as part of the HTTP binding. We'll look at this class and the way in which it is populated from the HTTP request in "SOAP Messages and MIME Headers." later in this chapter.

Once the `SOAPMessage` is created, it is passed to the `onMessage( )` method. If this method returns a reply, it needs to be sent back to the caller in the servlet's HTTP response message. Naturally, the reply needs to be wrapped in HTTP in the same way that the request was when it was transmitted by the sender. The original message was created as a result of calling the `SOAPConnection call( )` method, which opens an HTTP connection to the servlet and takes care of building the HTTP wrapper for the message. Here, however, we can't use this method because we want to return the message using the HTTP connection originally created by the client, so we have to create the HTTP binding for the reply message ourselves. This requires three steps:

1. Get any HTTP headers to be sent with the reply message from the message itself. This can be done by calling the `SOAPMessage getMimeHeaders( )` method, which returns them in the form of a `MimeHeaders` object.
2. Use the methods of the servlet's `HttpServletResponse` object to install the HTTP headers in the reply.
3. Get the XML representation of the `SOAPMessage` itself and write it to the `OutputStream` obtained from the `HttpServletResponse getOutputStream( )` method.

* In case you are wondering why the `createMessage( )` method needs access to the HTTP headers as well as the XML content, it is because it uses the `Content-Type` header to decide whether the SOAP message has attachments, and decodes the rest of the HTTP request appropriately.

Before we can start this process, however, we need to ensure that the HTTP headers for the SOAPMessage have been created. As you'll see later, a newly created SOAPMessage object does not actually contain any XML or any of the HTTP headers that will eventually be used when transmitting it over HTTP. Instead, it consists of a hierarchy of objects that represents the envelope, the header, the body, and so on, in much the same way as a DOM model represents an XML document. In order to force the headers to be created, we use the following code:

```
if (reply.saveRequired( )) {
    reply.saveChanges( );
}
```

The saveChanges( ) method creates not only the headers, but also the XML representation of the message itself. The saveRequired( ) method is used to discover whether it is actually necessary to perform this step—once saveChanges( ) is called, it does not need to be called again, and saveRequired( ) returns false until some change is made to the SOAPMessage that causes either the headers or the XML to need to be updated. The actual code that gets the HTTP headers from the reply SOAPMessage and installs them in the HTTP response is shown in the later section "SOAP Messages and MIME Headers."

The XML representation of a SOAPMessage can be written to an OutputStream using the following SOAPMessage method:

```
public void writeTo(OutputStream os);
```

It is not necessary to call saveChanges( ) before invoking this method, since writeTo( ) calls it for itself (if necessary). Hence, the following code is all that is necessary to write the reply message back to the caller:

```
// Send the completed message
OutputStream os = response.getOutputStream( );
reply.writeTo(os);
os.flush( );
```

Incidentally, the writeTo( ) method is a useful debugging aid because you can use code like the following to dump the XML represented by a SOAPMessage to the standard output stream:

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

Note, however, that this method writes only the XML—it does *not* include the MIME headers, which must be obtained separately by calling getMimeHeaders( ).

This leaves only one open issue regarding the handling of SOAP messages in a servlet: what destination URL should the client application supply to the SOAPConnection call( ) method to arrange for the message to be delivered to the servlet? The simple answer is that it depends on how the servlet is deployed. In the case of the echoservice example used here, the message echoing is provided by a simple servlet that is derived from SAAJServlet and overrides its onMessage( ) method to return the SOAPMessage that it is called with, having first written its content to the web containers log for debugging purposes, as shown in Example 3-3.

Example 3-3. A servlet that logs and echoes a received SOAP message

```
public class EchoServlet extends SAAJServlet {

    /**
     * Output stream used to save a SOAP message
     * for logging.
     */
    private ByteArrayOutputStream os = new ByteArrayOutputStream( );

    /**
     * Handles a received SOAP message by simply
     * returning it.
     */
    public SOAPMessage onMessage(SOAPMessage message) {

        // Convert the message to string representation
        // and log it.
        try {
            message.writeTo(os);
            log("Received SOAP message:\n" + os.toString( ));
            os.reset( );
        } catch (Exception ex) {
            log("Exception", ex);
        }

        // Return the received message to the caller.
        return message;
    }
}
```

This servlet (together with its base class, `SAAJServlet`) is wrapped in a WAR file and then deployed as a web application called `SAAJEchoService`. The web application's *web.xml* file looks like this:

```
<web-app>
    <display-name>SAAJ Echo Service</display-name>
    <description>SAAJ Message Echo Service</description>

    <servlet>
        <servlet-name>EchoService</servlet-name>
        <display-name>Servlet for the SAAJ Message Echo Service
        </display-name>
        <servlet-class>ora.jwsnut.chapter3.echoservice.EchoServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>EchoService</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

The `url-pattern` element means that all URLs that map to the web application `SAAJEchoService` are directed to the servlet. Assuming that the web application is

deployed in a web server listening at port 8000 on your local machine, the client application may therefore use the following URL to send SOAP messages to the servlet:

http://localhost:8000/SAAJEchoService

If the url-pattern element looks like this:

```
<url-pattern>/EchoServlet/*</url-pattern>
```

the result is that the required URL is:

http://localhost:8000/SAAJEchoService/EchoServlet

If you are writing a SAAJ client that is intended to communicate with an existing server, then the URL that you need to use should be obtained from the service provider. In some cases, the address of a service is not fully specified by a URL. An example of this might be a server implementation where all requests are initially handled by a single servlet and then routed internally to the appropriate web service based on something in the message itself. In order to aid with such an arrangement, the SOAP 1.1 specification defines an HTTP header called SOAPAction that can be used to provide a string that holds further information that can be used to identify the target web service. If a service does not require SOAPAction, its value should be set to the empty string. This is, in fact, the default value for the SOAPAction header when you create a new SOAPMessage, as you can see from Example 3-2. If the service you need to communicate with requires a specific SOAPAction value, you can install it using the MimeHeaders setHeader( ) method:

```
message.getMimeHeaders( ).setHeader("SOAPAction", "ServiceInfo");
```

The Anatomy of a SAAJ SOAP Message

Although a SOAP message is transmitted as a set of characters encoded in UTF-8, holding the message in character form would make it difficult to manipulate. Instead, SOAPMessage contains a hierarchy of objects that represent the various parts of the message and the XML elements and text (if any) that it contains. The message is kept in this form until it is ready to be transmitted, at which point a UTF-8 byte stream is created from its content. Figure 3-3 shows a typical hierarchy of elements that might make up a SOAPMessage object.

A SOAPMessage object consists of a single SOAPPart and zero or more AttachmentParts. The default message returned by the MessageFactory createMessage( ) method contains a SOAPPart but no attachments. AttachmentParts, if they are required, must be explicitly created and added to create a SOAP with attachments message. The SOAPPart, in turn, contains a SOAPEnvelope object, which represents the envelope shown in Figure 3-1 and Figure 3-2. Within the envelope is the SOAP message header, represented by an object implementing the SOAPHeader interface and the SOAP message body in the form of a SOAPBody object. Both the body and the header contain XML markup that constitutes, respectively, the message payload and any information required to qualify or process the payload, represented by SOAPElements or Text nodes.

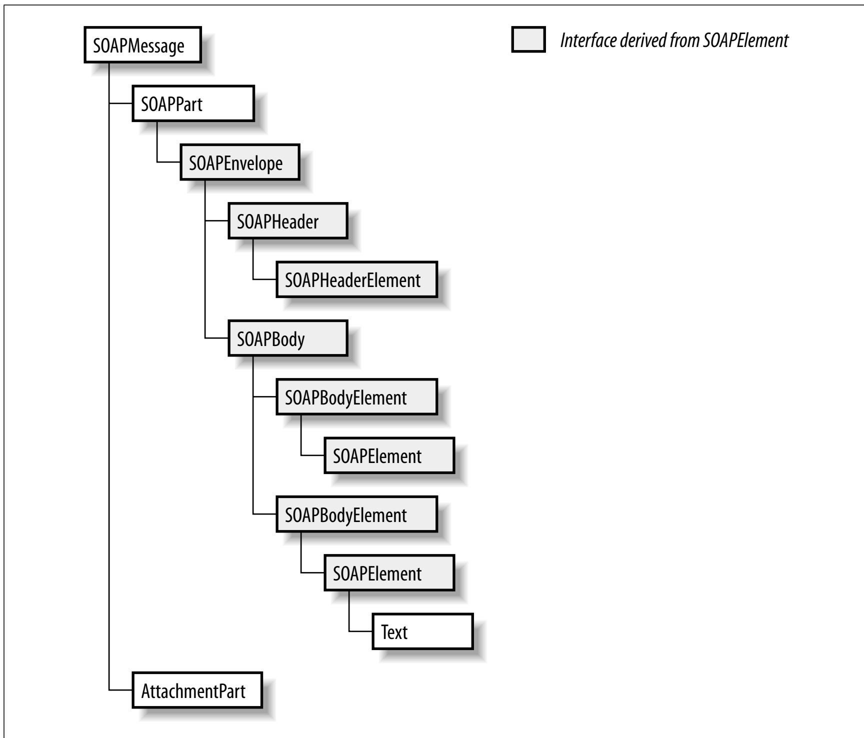


Figure 3-3. Component parts of a SOAPMessage object with a single attachment

SAAJ

SOAPMessage

`SOAPMessage` is an abstract class that represents the whole SOAP message. It contains methods that deal with the following:

MIME headers
: When a SOAP message is wrapped in a MIME-aware transport protocol such as HTTP or SMTP, MIME-related information must be included to identify the data being carried as an XML message. A `SOAPMessage` object supplies default values for the MIME `Content-Type`, `Content-Length`, and `SOAPAction` headers, but it also provides a method that returns the underlying `MimeHeaders` object that contains the associated MIME headers. You can use this object to retrieve the headers for a received message or to add additional ones for an outgoing message. See "SOAP Messages and MIME Headers" later in this chapter, for details.

  `SOAPMessage` also provides a convenience method to set or retrieve the value of the optional `Content-Description` header, which is used to attach descriptive text to the message.

Conversion between object and byte format
: A SOAP message is created and manipulated by application code as a tree of objects like that shown in Figure 3-3. For transmission, however, it must be converted to a byte stream that represents the XML itself. The `saveChanges()`

method creates a byte array that represents the object tree and stores it within the SOAPMessage object ready to be written by the writeTo( ) method. Since the creation of the byte representation is an expensive process, it should be performed only if it has not already been done, or if the content of the object tree has been modified since saveChanges( ) was last called. The saveRequired( ) method can be used to determine whether it is necessary to call saveChanges( ). The writeTo( ) method calls both saveRequired( ) and saveChanges( ) as necessary, so it is rarely necessary for client code to do anything other than call writeTo( ).

Access to the SOAPPart object

The SOAPMessage getSOAPPart( ) method returns a reference to the SOAPPart that contains the XML parts of the message:

```
public SOAPPart getSOAPPart( ) throws SOAPException;
```

Note that it is not possible to replace the SOAPPart associated with a SOAPMessage.

Attachments

A SOAPMessage can have any number of attachments, each of which may contain data of any type that has a MIME encoding. SOAPMessage provides the methods necessary to create the AttachmentPart objects that represent individual attachments, add them to the message, retrieve them, and remove them. Attachments must be used if it is necessary to include non-XML content in a message; their use is described in detail in "SOAP with Attachments," later in this chapter.

SOAPPart

The SOAPPart object is a wrapper for the SOAP envelope and therefore contains the part of the message that must be represented in XML. For simple SOAP messages, SOAPPart does not really add anything other than the ability to access the SOAPEnvelope object, but for a SOAP message with attachments, it provides the ability to manipulate the content of the MIME wrapper for the SOAP part of the message, as shown in Figure 3-2.

Access to the SOAPEnvelope object

For most practical purposes, the getEnvelope( ) method is the most important feature of SOAPPart:

```
public SOAPEnvelope getEnvelope( ) throws SOAPException;
```

There is no direct way with the SAAJ API to create your own SOAPEnvelope object and use it to replace the one installed when a SOAPMessage is created. However, you can replace the entire content of the SOAPPart, as described next, which has the same effect but is less convenient.

MIME headers

Most of the SOAPPart API is concerned with creating, retrieving, and manipulating MIME headers. These headers, however, are not related to those associated with the SOAPMessage—they are used only when the message has one or more attachments, and become part of the MIME wrapper for the SOAP part of the message. Although you can still use this API for a message that does not have attachments, it will have no practical effect on the message that is finally transmitted.

SOAPPart content replacement

The `getContent( )` and `setContent( )` methods allow access to and replacement of the content of the `SOAPPart` in a form that can be manipulated by the JAXP APIs. The `setContent( )` method in particular can be used to replace the SOAP envelope with an alternative that is represented by a DOM tree or the content of an input stream encoded in XML. While this might have some practical applications, it would be more useful to be able to import XML in either of these forms into the SOAP body, but there is currently no API that supports this.*

SOAPEnvelope

SAAJ

This object represents the SOAP envelope, which must contain a single `SOAPBody` object and may contain a `SOAPHeader` and additional XML elements. If these parts are present, they must appear in the following order: header, body, and additional elements. Like `SOAPPart`, `SOAPEnvelope` is of little direct use to most applications, except as a means of accessing the header and body parts.

Access to the SOAP header and body parts

`SOAPEnvelope` provides methods that allow references to the header and body objects that it contains to be retrieved:

```
public SOAPBody getBody( ) throws SOAPException;
public SOAPHeader getHeader( ) throws SOAPException;
```

The message created by the `MessageFactory createMessage( )` method contains empty header and body parts.

Replacement of the SOAP header and body parts

It is possible to create and add a new header or a new body part to a SOAP message. This is useful when modifying the content of a received message in order to forward it or to use it as the basis for the reply to the sender:

```
public SOAPBody addBody( ) throws SOAPException;
public SOAPHeader addHeader( ) throws SOAPException;
```

These methods create, install, and return an empty `SOAPBody` or `SOAPHeader` object, respectively. These methods represent the only way to directly create instances of these objects. You can only install a new `SOAPHeader` or a new `SOAPBody` if you have already removed the previous one. There are, however, no methods named `removeHeader( )` or `removeBody( )` that allow you to do this. Instead, you have to rely on the fact that both `SOAPBody` and `SOAPHeader` (like `SOAPEnvelope`) are derived from the `javax.xml.soap.Node` interface, which provides the `detachNode( )` method to allow it to be removed from whatever it is contained in. Here, for example, is how you would remove the `SOAPHeader` from a `SOAPMessage` and install a new, empty one:

```
// Remove the message header
SOAPEnvelope envelope = message.getSOAPPart( ).getEnvelope( );
envelope.getHeader( ).detachNode( );
```

* The only way to import XML into the SOAP body is to use the `SOAPPart getContent( )` method to get access to the content of the `SOAPPart`, transform it into a DOM tree, import the XML into the DOM tree, and then use `setContent( )` to replace the entire SOAP part with the modified content. If you really need to do this, you'll find the necessary code in *Java Web Services*, by David Chappell and Tyler Jewell (O'Reilly).

```
// Add a new header
SOAPHeader header = envelope.addHeader( );
```

Nodes are discussed in "Nodes, Elements, and Names," later in this chapter.

Creation of `Name` *objects*
: For reasons of backwards-compatibility with Version 1.0 of JAXM (which contained the API that is now known as SAAJ), `SOAPEnvelope` includes factory methods that create `Name` objects for use with XML elements within the envelope. In SAAJ 1.1, the preferred way to create `Name` objects is by using the `SOAPFactory` class. `Name` objects are described in "Nodes, Elements, and Names," later in this chapter.

SOAPHeader

`SOAPHeader` is a container for headers that determines the way in which the message payload is processed or interpreted. There are standardized uses of SOAP, such as SOAP-RP or ebXML-TRP, that specify the format of certain headers that are of meaning to receivers that implement these standards, but applications are free to define and use their own private headers.* SOAP itself does not specify any standard headers, and therefore the API provided by `SOAPHeader` is restricted to methods that add a new header entry or allow existing header entries to be accessed. SOAP headers are discussed in "SOAP Headers," later in this chapter.

SOAPBody

`SOAPBody` is the container that holds the real payload of the SOAP message. Logically, the payload consists of one or more XML document fragments. However, in the SAAJ API, the payload is constructed as a hierarchy of elements and text nodes, represented by the `SOAPElement` and `Node` interfaces (described in the next section). The payload is typically included in the body by creating a `SOAPBodyElement` to represent the root element of each fragment and then adding nested elements and text as necessary:

```
public SOAPBodyElement addBodyElement(Name name) throws SOAPException;
```

The body may also contain a `SOAPFault` object that reports a failure to properly process either a message header or the message payload itself. `SOAPBody` provides methods to add a `SOAPFault` object to the body and to handle one in a message that has been received. This API is covered in "SOAP Fault Handling," later in this chapter.

Nodes, Elements, and Names

The SAAJ interfaces that we have seen so far provide the framework within which SOAP messages can be built, but the actual message content is constructed by

* The terminology can get a little confusing when discussing headers. A SOAP message can only contain a single `SOAPHeader`, but a `SOAPHeader` may contain any number of XML fragments that represent application-defined headers. When the context does not make it clear whether "header" refers to one such XML fragment or the entire `SOAPHeader`, we use the term "header entry" instead.

using four more basic interfaces that are described in this section. Here's a brief description of these interfaces, which we'll expand on by reference to an example in the rest of this section:

`Name`
: In XML terms, a `Name` object represents a *qualified element name* (or *QName*)—that is, a local name together with an optional namespace prefix and the associated namespace URI. For example, in the case of the XML tag that represents the SOAP envelope:

```
<soap-env:Envelope xmlns:soap-env=
     "http://schemas.xmlsoap.org/soap/envelope/">
```

`Envelope` is the local name of the element, `soap-env` is the namespace prefix, `http://schemas.xmlsoap.org/soap/envelope/` is the URI that uniquely identifies the namespace, and the combination `soap-env:Envelope` is the qualified element name (qualified because it contains a namespace prefix).

`Node`
: `Node` is the base interface for the nodes in the object tree that make up the content of a SOAP message. It provides the methods necessary to link itself to or remove itself from its parent node as well as to allow application code to discover its parent node. However, this interface does not provide a way to add child nodes and therefore represents a leaf in the object tree. SAAJ applications usually do not deal directly with `Node`s—instead, they handle objects of the derived types `SOAPElement` and `Text`.

`Text`
: `Text` is a type of `Node` that holds a text string. `Text` is always a leaf node and its content may represent an XML comment.

`SOAPElement`
: `SOAPElement` is a subinterface of `Node` that adds the methods required to attach child nodes, and therefore need not always be a leaf in the object tree.

`SOAPElement` has a number of subinterfaces that represent entities that can appear within the `SOAPPart`. These subinterfaces and their relationships to `SOAPElement` and `Node` are shown in Figure 3-4.

It is interesting to note that `SOAPEnvelope`, `SOAPBody`, and `SOAPHeader` are all `SOAPElement`s, which implies that you can add content directly to them. This is, of course, true, although we'll see later that both `SOAPBody` and `SOAPHeader` have their own specific `SOAPElement` variants (`SOAPBodyElement` and `SOAPHeaderElement`) that are used as their immediate children and take special action when you attempt to add an arbitrary `SOAPElement` instead. We'll look at how a typical SOAP message is constructed by examining an example application that gets a list of O'Reilly book titles from a server and displays them in list form in a Swing-based user interface. Later in this chapter, we'll extend this example so that it can fetch images of the front covers of these books from the server, as an illustration of the use of non-XML SOAP message attachments.

Before running this example, you need to compile and deploy the service implementation, which is based on `SAAJServlet`. To do so, start your web server or application server, open a command window and make *chapter3\bookimageservice* your working directory, and then use the following command to build and deploy the servlet:

```
ant deploy
```

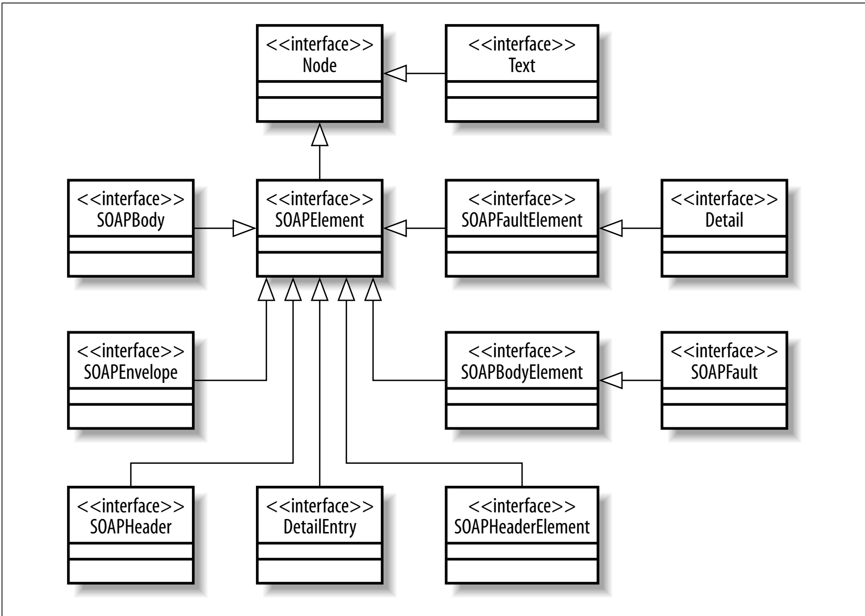


Figure 3-4. SAAJ interfaces used in building SOAP messages

Next, compile and run the client application using the command:

```
ant compile-client run-client
```

Once the client starts, you'll see the list of books that it has obtained from the server, as shown in Figure 3-5.

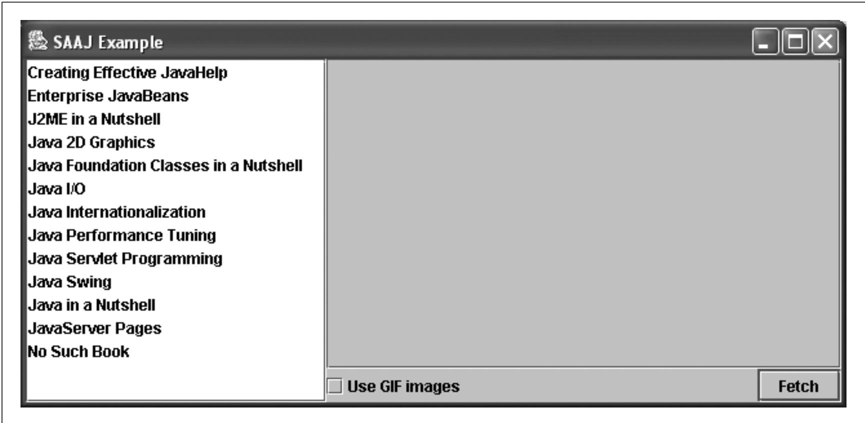


Figure 3-5. SAAJ client showing a list of book titles obtained from a SAAJ server

Constructing the SOAP message to get the book titles

The SOAP message that is sent to get the list of book titles is shown in Example 3-4.

Example 3-4. A SOAP message sent to retrieve book titles from a web service

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
3                      xmlns:tns="urn:jwsnut.bookimageservice">
4       <soap-env:Body>
5           <tns:BookList/>
6       </soap-env:Body>
7   </soap-env:Envelope>
```

There are three differences between this message and the empty one shown in Example 3-2:

- The SOAP header part has been removed because headers are not required by this application.
- The addition of a namespace declaration on the `Envelope` element.
- The SOAP body contains an application-defined element called `BookList`.

The `BookList` element is, of course, not defined by the SOAP 1.1 specification—it is a private element that is recognized by the web service as a request to send the list of book titles that it knows about. Since this request does not need any parameters, all that is required is an empty element. However, since the `BookList` element is private, we define an XML namespace for the web service, assign a prefix to it, and associate the prefix with the element name, so that there is no ambiguity about the meaning of the request.

The namespace itself is defined by the `xmlns` attribute on the `Envelope` element, as shown on line 3 of Example 3-4. In order to declare a namespace, we have to assign a prefix and associate it with the namespace's URI. In this case, we chose to use the prefix `tns`. Although arbitrary, it is quite common to use this particular prefix for the namespace that refers to the service's private elements, since the letters `tns` serve as a useful mnemonic for "this namespace," as distinct from the namespace used for the elements that are part of the SOAP envelope itself. The namespace URI is, again, arbitrary, although it should be unique, and the same value must be used by both the client and the web service implementation.* Here, we choose to use a URN with a string value that identifies the service as the book image service from this book, but we could have used any other valid URI.

The SOAP 1.1 specification does not actually require elements in the body of the SOAP message to be namespace-qualified, although it recommends that they are. From the point of view of the service implementation, it is just as well to require that the client supplies the namespace so it can unambiguously check that if it receives a `BookList` element from a client, the element actually represents the `BookList` request that it provides, rather than a similarly named request for an entirely different service. You'll see shortly how the service implementation makes this check. Incidentally, you don't have to associate a namespace prefix with every

* Although the client and the service implementation must use exactly the same URI for the namespace, they do not have to use the same prefix, because the prefix serves only as a shorthand for the URI itself.

SOAP element. You can, if you wish, declare a default namespace on an element, like this:

```
<?xml version="1.0" encoding="UTF-8"?>
  <soap-env:Envelope xmlns:soap-env=
       "http://schemas.xmlsoap.org/soap/envelope/"
       xmlns="urn:jwsnut.bookimageservice">
      <soap-env:Body>
          <BookList/>
      </soap-env:Body>
 </soap-env:Envelope>
```

An `xmlns` attribute without an associated namespace prefix specifies the namespace that will be associated with any element, such as `BookList`, that does not have an explicit namespace prefix. This default is in operation for the scope of the element to which the `xmlns` attribute is attached.

Now let's look at the code that created the `SOAPMessage` shown in Example 3-4. This code is shown in Example 3-5.

Example 3-5. Constructing a SOAP message using the SAAJ APIs

```
1        // Build the message
2        SOAPMessage message = messageFactory.createMessage( );
3
4        // Remove the message header
5         SOAPEnvelope envelope = message.getSOAPPart( ).getEnvelope( );
6         envelope.getHeader( ).detachNode( );
7
8         // Set up the namespace declaration
9         envelope.addNamespaceDeclaration(SERVICE_PREFIX, SERVICE_URI);
10
11        // Add the element for the book list request
12        SOAPBody soapBody = envelope.getBody( );
13        soapBody.addBodyElement(BOOK_LIST_NAME);
```

The first part of this code (lines 2 to 6) creates an empty message, gets a reference to the `SOAPEnvelope`, and then removes the empty header part, which this application does not use. Line 9 adds the declaration of the private namespace used by this service using the `SOAPElement` `addNamespaceDeclaration( )` method:

```
public SOAPElement addNamespaceDeclaration(String prefix, String uri)
  throws SOAPException;
```

In this case, the prefix and the URI supplied are declared as static strings within the class definition:

```
// The URI used to qualify elements for this service
private static final String SERVICE_URI = "urn:jwsnut.bookimageservice";

// The namespace prefix used in elements for this service
private static final String SERVICE_PREFIX = "tns";
```

The `addNamespaceDeclaration( )` method can be used to attach a namespace declaration to any `SOAPElement`. Its effect is scoped to that element and its child elements, and returns a reference to the `SOAPElement` on which it is invoked. In this case, we apply the declaration to the `SOAPEnvelope` itself, so that it applies to the entire message.

Lines 12 and 13 are responsible for adding the BookList element to the SOAP body, using the SOAPBody addBodyElement( ) method:

```
public SOAPElement addBodyElement(Name name) throws SOAPException;
```

The Name object passed to this method determines the element name and the namespace within which it is defined. As you'll see later in this chapter, there are several ways to create a Name object. In this case, the Name is created using one of the methods provided by the SOAPFactory class:

```
// The name of the element used to request a book name list
private static Name BOOK_LIST_NAME;

// SOAPFactory for message pieces
private static SOAPFactory soapFactory;

// Create the BookList element
soapFactory = SOAPFactory.newInstance( );
BOOK_LIST_NAME = soapFactory.createName("BookList", SERVICE_PREFIX,
  SERVICE_URI);
```

SOAPFactory is a factory for objects that can be added to SOAP messages, including SOAPElements, Detail objects (which are used in connection SOAP faults), and Names.* A Name object is a representation of a fully qualified element name and therefore requires the following attributes:

- The local name of the element—in this case, BookList
- The URI of the namespace within which the name is defined
- The prefix used to represent the namespace (tns in this example)

It is also possible to have Names that are not namespace-qualified. In these cases, the URI and namespace prefix are both null.

Once the Name is created, the SOAPBody addBodyElement( ) method creates the actual SOAPBodyElement and installs it in the body. Since this element does not have any nested elements, it ends up looking like this:

```
<tns:BookList/>
```

You can see that the namespace prefix and the local name that were used to create the Name object become part of the element.

Handling the BookList request in the service implementation

The book image service is implemented as a servlet derived from SAAJServlet, much of the code for which is shown in Example 3-6.

* SOAPFactory is a new class added to SAAJ Version 1.1. The earlier API (JAXM 1.0) used the SOAPEnvelope createName( ) methods, which require access to the SOAPEnvelope object, to create Name objects. In many cases, however, an application will want to construct part of a SOAP message without having to have access to the SOAPEnvelope within which it will eventually be enclosed, so the context-free SOAPFactory class was added to make this possible. SOAPFactory is a generalization of the JAXM 1.0 SOAPElementFactory class, which is deprecated as of SAAJ Version 1.1.

Example 3-6. A servlet that uses SAAJ to provide the book image web service

```
/**
 * A servlet that uses SAAJ attachments to
 * serve images to a client.
 */
public class BookImageServlet extends SAAJServlet {

    // The XML Schema namespace
    private static final String XMLSCHEMA_URI =
      "http://www.w3.org/2001/XMLSchema";

    // The XML Schema instance namespace
    private static final String XMLSCHEMA_INSTANCE_URI =
      "http://www.w3.org/2001/XMLSchema-instance";

    // Namespace prefix for XML Schema
    private static final String XMLSCHEMA_PREFIX = "xsd";

    // Namespace prefix for XML Schema instance
    private static final String XMLSCHEMA_INSTANCE_PREFIX = "xsi";

    // The namespace prefix used for SOAP encoding
    private static final String SOAP_ENC_PREFIX = "SOAP-ENC";

    // The URI used to qualify elements for this service
    private static final String SERVICE_URI = "urn:jwsnut.bookimageservice";

    // The namespace prefix used in elements for this service
    private static final String SERVICE_PREFIX = "tns";

    // MessageFactory for replies from this service
    private static MessageFactory messageFactory;

    // SOAPFactory for message pieces
    private static SOAPFactory soapFactory;

    // The name of the element used to request a book name list
    private static Name BOOK_LIST_NAME;

    // The name of the element used to reply to a book name list request
    private static Name BOOK_TITLES_NAME;

    // The name of the element used to request a book image
    private static Name BOOK_IMAGE_REQUEST_NAME;

    // The name of the element used to respond to a book image request
    private static Name BOOK_IMAGES_NAME;

    // The name of the attribute used to hold the image encoding
    private static Name IMAGE_TYPE_ATTRIBUTE;

    // The name of the href attribute
    private static Name HREF_ATTRIBUTE;
```

Example 3-6. A servlet that uses SAAJ to provide the book image web service (continued)

```
    /**
     * Handles a received SOAP message.
     */
    public SOAPMessage onMessage(SOAPMessage message) throws SOAPException {

        if (messageFactory == null) {
            // Create all static data on first call
            messageFactory = MessageFactory.newInstance( );
            soapFactory = SOAPFactory.newInstance( );
            BOOK_LIST_NAME = soapFactory.createName("BookList", SERVICE_PREFIX,
               SERVICE_URI);
            BOOK_TITLES_NAME = soapFactory.createName("BookTitles",
               SERVICE_PREFIX, SERVICE_URI);
            BOOK_IMAGE_REQUEST_NAME =
                        soapFactory.createName("BookImageRequest",
               SERVICE_PREFIX, SERVICE_URI);
            BOOK_IMAGES_NAME = soapFactory.createName("BookImages",
               SERVICE_PREFIX, SERVICE_URI);
            IMAGE_TYPE_ATTRIBUTE = soapFactory.createName("imageType",
               SERVICE_PREFIX, SERVICE_URI);
            HREF_ATTRIBUTE = soapFactory.createName("href");
        }

        // Create the reply message and define the namespace
        // and encoding for the elements used in the reply.
        SOAPMessage reply = messageFactory.createMessage( );
        SOAPEnvelope replyEnvelope = reply.getSOAPPart( ).getEnvelope( );
        replyEnvelope.getHeader( ).detachNode( );
        replyEnvelope.addNamespaceDeclaration(SERVICE_PREFIX, SERVICE_URI);
        replyEnvelope.addNamespaceDeclaration(SOAP_ENC_PREFIX,
                                    SOAPConstants.URI_NS_SOAP_ENCODING);
        replyEnvelope.addNamespaceDeclaration(XMLSCHEMA_PREFIX, XMLSCHEMA_URI);
        replyEnvelope.addNamespaceDeclaration(XMLSCHEMA_INSTANCE_PREFIX,
                                          XMLSCHEMA_INSTANCE_URI);
        replyEnvelope.setEncodingStyle(SOAPConstants.URI_NS_SOAP_ENCODING);
        SOAPBody replyBody = reply.getSOAPPart().getEnvelope( ).getBody( );

        // There are two requests - one for the list of
        // book titles, the other for the image for a book.
        SOAPBody requestBody = message.getSOAPPart().getEnvelope( ).getBody( );
        Iterator iter = requestBody.getChildElements( );
        if (iter.hasNext( )) {
            // The child element contains the request
            SOAPElement element = (SOAPElement)iter.next( );
            Name elementName = element.getElementName( );
            if (elementName.equals(BOOK_LIST_NAME)) {
                handleBookListRequest(replyBody);
            } else if (elementName.equals(BOOK_IMAGE_REQUEST_NAME)) {
                handleBookImageRequest(element, reply);
            } else {
                // Unrecognized request - this is a fault.
```

SAAJ

Example 3-6. A servlet that uses SAAJ to provide the book image web service (continued)

```
                createFault(replyBody, "soap-env:Client.UnknownRequest",
                  "Unrecognized request", SERVICE_URI, elementName.
                  getLocalName( ));
            }
        } else {
            // No request - this is a fault
            createFault(replyBody, "soap-env:Client.MissingRequest",
               "Missing request", SERVICE_URI, "No request found");
        }
        return reply;
    }

    // HELPER METHODS NOT SHOWN...
}
```

The first part of the onMessage( ) method is concerned with creating objects such as Names that will be used to build a reply message the first time the service is used, and then creating the envelope for the reply message, which we'll look at more closely very shortly. Before the reply can be constructed, however, the received message must be examined to identify the client request that it contains. To do this, it is necessary to find the first child element of the SOAP body and extract its element name. Here is the code that does this:

```
            SOAPBody requestBody = message.getSOAPPart( ).getEnvelope( )
               .getBody( );
            Iterator iter = requestBody.getChildElements( );
            if (iter.hasNext( )) {
                // The child element contains the request
                SOAPElement element = (SOAPElement)iter.next( );
```

Although SOAPBody does not provide any methods for accessing its content, the SOAPElement interface from which it is derived does provide this facility. In particular, the getChildElements( ) method returns an Iterator that can be used to step through all of the immediate child elements of the SOAPElement to which it is applied. In this case, we expect to find exactly one child element; if we don't find any, then the message has been constructed incorrectly and the request will not be processed. Instead, a reply message containing a SOAP fault will be returned to the client, as described in "SOAP Fault Handling," later in this chapter.

Assuming that we find a child element, the next step is to find out what it represents. So far, we have only seen the BookList request, but in fact, this servlet can also handle a request for a cover image for a book with a given title, the implementation of which we'll look at when we discuss how to use SOAP attachments. It is therefore necessary to work out which of these two requests has been received:

```
    Name elementName = element.getElementName( );
    if (elementName.equals(BOOK_LIST_NAME)) {
        handleBookListRequest(replyBody);
    } else if (elementName.equals(BOOK_IMAGE_REQUEST_NAME)) {
        handleBookImageRequest(element, reply);
    } else {
```

```
        // Unrecognized request - this is a fault.
        createFault(replyBody, "soap-env:Client.UnknownRequest",
          "Unrecognized request", SERVICE_URI, elementName.getLocalName( ));
    }
```

The SOAPElement getElementName( ) method returns the Name object that identifies the element, which contains the local element name and the URI for its associated namespace. To determine which request has been received, we compare the element name against two fixed values, BOOK_LIST_NAME and BOOK_IMAGE_REQUEST_NAME, which are constructed (using the SOAPFactory class) with the appropriate local names and the private URN associated with this service. This is, of course, the same URN (urn:jwsnut.bookimageservice) used by the client. Notice that the comparison is performed using the equals( ) method, which, for Name objects, returns true if both the local names and the namespace URIs are the same.* Depending on the result of the test, the handleBookListRequest( ) or the handleBookImageRequest( ) method is called, or, if the element name does not match either of the expected values, a SOAP fault is returned to the client. Here, we are interested only in the handleBookListRequest( ) method. Before we look at its implementation, however, let's examine the SOAP message that the service sends in reply to the client's BookList request, which is shown in Example 3-7.

Example 3-7. A SOAP message containing a list of book titles

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:tns="urn:jwsnut.bookimageservice"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        soap-env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <soap-env:Body>
        <tns:BookTitles xsi:type="SOAP-ENC:Array"
                        SOAP-ENC:arrayType="xsd:string[]">
            <item>Java in a Nutshell</item>
            <item>J2ME in a Nutshell</item>
            <item>Java 2D Graphics</item>
            <item>Java I/O</item>
            <item>JavaServer Pages</item>
            <item>Java Internationalization</item>
            <item>Java Foundation Classes in a Nutshell</item>
            <item>Java Performance Tuning</item>
            <item>Creating Effective JavaHelp</item>
            <item>Enterprise JavaBeans</item>
            <item>Java Servlet Programming</item>
            <item>Java Swing</item>
        </tns:BookTitles>
    </soap-env:Body>
</soap-env:Envelope>
```

* In the case of Names that do not have namespace qualifiers, this test still works because the namespace URI for such a Name is null.

If you compare this to the BookList request message shown in Example 3-4, you'll see that the Envelope of the reply message declares many more namespaces than the request. Each of these namespaces, which are listed in Table 3-2, is added to the SOAPEnvelope object using the addNamespaceDeclaration( ) method that we saw earlier; therefore, they apply to the whole message.

Table 3-2. Namespaces commonly used in SOAP messages

URI	Description
http://schemas.xmlsoap.org/soap/envelope/	Associated with the SOAP message envelope. This namespace is used on all of the standard envelope elements.
http://www.w3.org/2001/XMLSchema	Definitions relating to the W3C XML Schema standard. Typically assigned the prefix xsd and used to indicate built-in datat ypes defined by this standard, such as xsd:string.
http://www.w3.org/2001/XMLSchema-instance	Another namespace associated with W3C XML Schema, this is typically assigned the prefix xsi and is attached to attributes such as xsi:type that define the data types of elements in the SOAP message.
http://schemas.xmlsoap.org/soap/encoding/	A namespace that indicates definitions taken from the SOAP section 5 encoding rules.

The implementation shown in Example 3-6 defines constants to represent both the URIs and the prefixes that will be associated with them when constructing SOAP messages. The javax.xml.soap.SOAPConstants interface includes constant values (SOAPConstants.URI_NS_SOAP_ENCODING and SOAPConstants.URI_NS_SOAP_ENVELOPE) for the URIs associated with the SOAP encoding rules and the envelope, but does not define the other URIs, even though they are likely to be used just as often.

The Envelope element also has an attribute that you haven't seen before:

```
soap-env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```

The value of this attribute is a URI that indicates the encoding rules used to determine the representation of the data within the element to which it is attached. In this case, the message is encoded using SOAP section 5 encoding rules. It is, of course, necessary for both the sender and receiver of the message to understand the encoding rules that are actually used within the message, which is why the SOAP 1.1 specification includes these rules that applications are encouraged (but not absolutely required) to use.

Since the encodingStyle attribute is attached to the Envelope element, it applies to everything within the envelope unless explicitly overridden at the level of a nested element, in which case that element and its child elements can be encoded according to different rules:

```
<OuterElement soap-env:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/">
    <!-- SOAP section 5 encoding rules apply here -->
    <InnerElement soap-env:encodingStyle="urn:MyPrivateStyle">
        <!-- Private encoding rules apply here -->
    </InnerElement>
    <!-- SOAP section 5 encoding rules apply again -->
</OuterElement>
```

Nested encodings such as this might be convenient if the message contains all or part of an XML document that uses a different representation than that used by the parties exchanging SOAP messages and that ultimately will probably be processed by a different application.

The body of the reply message is simply an array of strings, where each string represents the title of a book. In order for the client to understand the reply, it must know how the server will represent an array of strings. Fortunately, the SOAP section 5 encoding rules provide a standard way of representing an array of objects in XML. For simple types such as strings, the canonical form looks like this:

```
<tns:ElementName xsi:type= "SOAP-ENC:Array" SOAP-ENC:arrayType= "arrayType">
    <item>itemValue1</item>
    <item>itemValue2</item>
    <!-- and so on -->
</tns:ElementName>
```


The attribute `xsi:type` is defined by the W3C XML Schema standard and indicates that the element contains an array as defined by the SOAP encoding rules, since the namespace prefix for the `Array` attribute is `SOAP-ENC`, which is mapped, in this example, to the URI for these encoding rules. The `arrayType` attribute specifies the type of the array. For an array of strings, this will typically have the value:

```
SOAP-ENC:arrayType="xsd:string[]"
```

Here, the value `xsd:string` is the XML Schema attribute value that defines a string,* and the square brackets obviously indicate an array. In this case, the square brackets are empty, which means that the number of elements in the array should be determined by counting the child elements. It is also possible to specify fixed bounds by composing the attribute like this, to indicate a string array with 12 elements:

```
SOAP-ENC:arrayType="xsd:string[12]"
```

The elements of the array are all represented as nested elements, the names of which (commonly referred to as accessor names) must all be the same but can otherwise be arbitrary. In this example, we use the element name `item`, but any legal XML name will suffice. In the actual reply message shown in Example 3-7, the array element is actually called `BookTitles` and, not surprisingly, is qualified with the namespace associated with this web service. Notice, however, that the array elements use an element name (`item`) that is not namespace-qualified. This is explicitly allowed by the SOAP encoding rules.

Now let's look at the code in the `handleBookListRequest( )` method that creates this representation of a string array, which is shown in Example 3-8.

* You can see that `xsd:string` is an attribute defined by XML Schema because the `xsd` prefix in this example (represented in Example 3-6 by the constant value `XMLSCHEMA_PREFIX`) is mapped to the XML Schema URI (represented by the constant value `XMLSCHEMA_URI`).

Example 3-8. Creating a SOAP message containing an array of book titles

```
/**
 * Handles a request for list of book names.
 */
private void handleBookListRequest(SOAPBody replyBody) throws SOAPException {
    // Create a BookTitles element containing an entry
    // for each book title.
    SOAPBodyElement bodyElement = replyBody.addBodyElement(BOOK_TITLES_NAME);

    // Add 'xsi:type = "SOAP-ENC:Array"'
    bodyElement.addAttribute(
        soapFactory.createName("type", XMLSCHEMA_INSTANCE_PREFIX,
            XMLSCHEMA_INSTANCE_URI), SOAP_ENC_PREFIX + ":Array");

    // Add 'SOAP-ENC:arrayType = "xsd:string[]"
    bodyElement.addAttribute(
        soapFactory.createName("arrayType", SOAP_ENC_PREFIX,
            SOAPConstants.URI_NS_SOAP_ENCODING), XMLSCHEMA_PREFIX +
            ":string[]");

    // Add an array entry for each book
    String[] titles = BookImageServletData.getBookTitles( );
    for (int i = 0; i < titles.length; i++) {
        SOAPElement titleElement = bodyElement.addChildElement("item");
        titleElement.addTextNode(titles[i]);
    }
}
```

We're not going to show how the book titles themselves are obtained—the logic for this is hidden in a separate class called `BookImageServletData`, which provides a method that returns all of the titles in the form of an array of strings. Instead, we're going to concentrate on how to build the SOAP representation of the titles list. The first step is to create the `BookTitles` element, which is done using the `SOAPBody` `addBodyElement( )` method, using the qualified name of the element as its argument. This is the same way that the client created the `BookList` request. Next, we need to add to this element the `xsi:type` and `SOAP-ENC:arrayType` attributes required by the SOAP-encoding rules to indicate that the element represents an array of strings. To add an attribute to a `SOAPElement`, the `addAttribute( )` method is used:

```
public SOAPElement addAttribute(Name name, String value);
```

`name` is the qualified name of the attribute, and `value` is the value to be associated with it. As before, in this example, we use the `SOAPFactory` class to create `Name` objects for the attributes, which creates the `Name` object for the attribute `xsi:type`, which is then passed to the `addAttribute( )` method:

```
// Add 'xsi:type = "SOAP-ENC:Array"'
bodyElement.addAttribute(
            soapFactory.createName("type", XMLSCHEMA_INSTANCE_PREFIX,
                XMLSCHEMA_INSTANCE_URI), SOAP_ENC_PREFIX + ":Array");
```

The addAttribute( ) method returns a reference to the SOAPElement to which the attribute was added, which makes it possible to add multiple attributes by chaining calls:

```
bodyElement.addAttribute(...).addAttribute(...).addAttribute( );
```

Having installed the BookTitles element in the reply body, we now need to add a child element for each book title. The SOAPElement interface provides five methods that allow a nested element to be added. In this case, since the element is part of a SOAP-encoded array definition, its name does not need to be namespace-qualified, so we can use the variant of the addChildElement( ) method that requires a string to specify the local name of the element to be created and returns a reference to the SOAPElement itself:

```
SOAPElement titleElement = bodyElement.addChildElement("item");
```


At this point, we have SAAJ message structures that amount to the following XML:

```
 <tns:BookTitles xsi:type="SOAP-ENC:Array" SOAP-
ENC:arrayType="xsd:string[]">
        <item/>
<tns:BookTitles>
```

What is missing here is the book title within the item element. Unlike item itself, the book title should not be represented by an XML element—it is, in fact, just a text string. To add text to a SOAPElement, we need to create and add a Text node, using the SOAPElement addTextNode( ) method:

```
public SOAPElement addTextNode(String text) throws SOAPException;
```

You may be surprised to see that this method is defined to return a SOAPElement instead of an object of type Text, despite its name (especially since Text is not derived from SOAPElement; see Figure 3-4). In fact, this method returns a reference to the SOAPElement on which it was invoked, not the Text object that was created. In most cases, you don't actually need to get access to the Text object itself; the only way to do so is to use the SOAPElement getChildElements( ) method to get an Iterator over all of the element's children and search for the one that implements the Text interface. Adding the following method call:

```
titleElement.addTextNode(titles[i])
```

finally completes the array element for the book title, so that now the message body contains the following:

```
<tns:BookTitles xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[]">
        <item>Creating Effecttive JavaHelp</item>
<tns:BookTitles>
```

The rest of the loop in the handleBookListRequest( ) method adds item elements for each of the books that the service knows about. When this method completes, the reply message is completely built and is returned from the servlet's onMessage( ) method, to be sent back to the client.

By now, you have probably realized that implementing a web service using SAAJ involves much more code that deals directly with details such as the exact layouts of SOAP messages than you see if you simply used JAX-RPC. In fact, you might wonder how the author of a client application that needs to connect to a web service implemented by somebody else is supposed to know what message types the service supports and how to build the required element structure. Here, for example, the client application knows how to build the requests for the book image service and how to interpret the replies only because they were both implemented by the same person. In the real world, service providers are supposed to publish the interfaces to their web services in the form of WSDL documents. As you'll see in Chapter 5, WSDL includes XML elements that describe the operations that a web service supports and the formats of the messages required to access them.

Client processing for the reply to the BookList request

The processing performed by the client when it receives the reply to its `BookList` request is very simple—all it has to do is loop over the SOAP array in the message body and extract the book titles from each nested element. Example 3-9 shows the implementation.

Example 3-9. Handling the server's reply to the BookList request

```
SOAPBody replyBody = reply.getSOAPPart().getEnvelope( ).getBody( );
if (replyBody.hasFault( )) {
    SOAPFault fault = replyBody.getFault( );
    throw new SOAPException("Fault when getting book titles: " +
      fault.getFaultString( ));
}

// The body contains a "BookTitles" element with a nested
// element for each book title.
Iterator iter = replyBody.getChildElements(BOOK_TITLES_NAME);
if (iter.hasNext( )) {
    ArrayList list = new ArrayList( );
    SOAPElement bookTitles = (SOAPElement)iter.next( );
    iter = bookTitles.getChildElements( );
    while (iter.hasNext( )) {
         list.add(((SOAPElement)iter.next( )).getValue( ));
    }
}
```

The first part of this code checks whether the server detected an error with the request by looking for a SOAP fault element in the reply body. `SOAPBody` provides the convenience methods `hasFault( )`, which returns `true` if there is a fault present, and `getFault( )`, which looks for and returns the `SOAPFault` element. Since the presence of a fault indicates either an error in the client code that constructed the request or a problem at the server, there is no reasonable recovery possible in this case and a `SOAPException` is thrown.

If there is no fault, then the body of the reply contains a BookTitles element. Earlier, we saw that SOAPElement has a getChildElements( ) method that returns all of the children of the element that it is invoked on, and which we used previously to get all of the top-level elements in the body of a SOAP message. Here, we use an overloaded variant of that method, which accepts a Name as its argument:

```
public Iterator getChildElements(Name name) throws SOAPException;
```

This method returns only those elements whose name matches the one given. In this case, we get back an Iterator over all of the BookTitles elements in the SOAPBody—there should be exactly one. Once we have a reference to this element, we invoke the getChildElements( ) method on it to gain access to all of the nested array elements. In this case, we use the variant of getChildElements( ) that returns all of the children, both because we know that we need to process all of them and because the name associated with the element is arbitrary (and therefore we cannot construct an appropriate Name object). Finally, we iterate over all of the nested elements and extract the title by using the getValue( ) method, which returns the text associated with a Node, if there is any.

More on SOAPElements

The example that you have just seen shows some of the ways in which a SOAPElement can be created. It also demonstrates some of its API. In this section, we take a closer look at how to create and use SOAPElements.

Creation of SOAPElements

So far, you have seen how to create new SOAPElements and incorporate them into the node tree for a SOAPMessage in a single step. To create a SOAPElement that appears directly in the SOAPBody, in a SOAPHeader, or in a SOAPFault, use convenience methods provided by these classes, of which the one provided by SOAPBody is typical:

```
public SOAPBodyElement addBodyElement(Name name) throws SOAPException;
```

This method returns not a SOAPElement but an instance of SOAPBodyElement, which is derived from SOAPElement. All top-level elements in the SOAP body must be SOAPBodyElements, rather than simply SOAPElements. Similarly, top-level elements inside the SOAPHeader must be of type SOAPHeaderElement, and top-level elements in a SOAPFault will be of type SOAPFaultElement. As far as the SAAJ API is concerned, there is no real difference between a SOAPElement and a SOAPBodyElement or a SOAPHeaderElement, since neither of these interfaces add any methods in addition to those defined by SOAPElement.

Once you have a SOAPElement, you can add child elements to it using one of the five overloaded variants of its addChildElement( ) method, all of which return the newly created SOAPElement:

public SOAPElement addChildElement(Name name)

This variant adds a new SOAPElement for which the name is obtained by the supplied Name argument. If the Name is a qualified name, then the element as written to the XML output stream has both a local name and an associated namespace prefix; otherwise, it has an unqualified local name.

`public SOAPElement addChildElement(String localName)`
: Creates a new `SOAPElement` with the given unqualified name and adds it to the child list of the element on which this method is invoked.

`public SOAPElement addChildElement(String localName, String prefix, String uri)`
: Creates a new `SOAPElement` with a fully qualified name created from the supplied arguments and adds it to the child list of the element on which this method is invoked. Using this method has the same result as creating a `Name` using the supplied arguments and then calling the `addChildElement(Name name)` variant.

`public SOAPElement addChildElement(String localName, String prefix)`
: This is similar to the previous variant, except that it does not provide the URI for the namespace that the `localName` argument is associated with. However, if the supplied prefix can be resolved to a namespace URI by examining the `SOAPElement` on which this method or its ancestors are invoked, then that URI is used when creating the `SOAPElement`. If this is not possible, then a `SOAPException` is thrown. For example, consider the following code, where `soapBody` is a reference to a `SOAPBody` object:

```
Name name = soapFactory.createName("bodyElement", "tns", "urn:service");
SOAPBodyElement bodyElement = soapBody.addBodyElement(name);
SOAPElement childElement = bodyElement.addChildElement("childElement",
   "tns");
```

Here, the `addChildElement( )` call to create `childElement` does not specify the namespace URI. However, since the namespace prefix `tns` supplied as the second argument is defined by the parent node, a mapping to the URN given in that element is inferred and `childElement` is associated with this URI.

`public SOAPElement addChildElement(SOAPElement element)`
: Requests that the given `SOAPElement` be added to the list of children of the element on which it is invoked. Depending on the type of the parent element, however, a different `SOAPElement` may be added to the parent.

Although the methods all declare the return value to be of type `SOAPElement`, in reality they may return a subinterface of `SOAPElement`. In particular, calling `addChildElement( )` on a `SOAPBody` object (which is possible because `SOAPBody` is itself a `SOAPElement`) will always return a `SOAPBodyElement`. Consider the following code:

```
SOAPElement bodyElement = soapBody.addBodyElement(name);
    // "name" is of type Name
SOAPElement secondElement = soapBody.addChildElement("localName");
```

Here, `bodyElement` is obviously of type `SOAPBodyElement` because `addBodyElement( )` is defined to return an object of that type. However, `secondElement` is also a `SOAPBodyElement`, since it is a top-level element within a `SOAPBody`. Similar results are obtained when `addChildElement( )` is invoked on an element of type `SOAPHeader` or `SOAPFault`.

Now suppose that `element` is an existing `SOAPElement` called `TopElement`, and then consider the following code, in which `soapBody` again refers to an object of type `SOAPBody`:

```
SOAPElement childElement = soapElement.addChildElement("ChildElement");
SOAPBody addedElement = body.addChildElement(element);
```

This attempts to add an existing SOAPElement, complete with a child element of its own, to a SOAPBody. There seems to be a problem here, since the immediate child of a SOAPBody must be a SOAPBodyElement, but the element being added is actually a SOAPElement. What actually happens is that a new SOAPBodyElement is created as a copy of the supplied SOAPElement, added to the SOAPBody instead of it, and returned from the addChildElement( ) method. The copied element also has as its child a copy of the child element associated with the original.

Since all of the SOAPElements that you have seen so far have been created as children of other SOAPElements, you might wonder how we could create the unattached SOAPElement used in the example just shown. To create a freestanding SOAPElement, use one of the following methods of SOAPFactory:

```
public SOAPElement createElement(Name name) throws SOAPException;
public SOAPElement createElement(String localName) throws SOAPException;
public SOAPElement createElement(String localName, String prefix,
    String uri) throws SOAPException;
```

The SOAPElementFactory class also has identical methods to those shown here. However, SOAPElementFactory is deprecated as of SAAJ 1.1 and the SOAPFactory methods (to which those in SOAPElementFactory actually delegate) should be used instead.

These methods can be used, as shown in a small way in the previous example, to build a message fragment as a freestanding entity and then add it to a SOAPMessage, without having to supply a reference to any part of the message to the code that builds the fragment.

Removal of SOAPElements

Although SOAPElement does not provide a method that allows any of its child elements to be removed, the Node interface, from which SOAPElement is derived, has a detachNode( ) method that removes the Node on which it is invoked from its parent. For example, the following code shows how to use this method to remove the first child of a SOAPElement:

```
SOAPElement parent = soapFactory.createElement("Element");
parent.addChildElement("Child1");
parent.addChildElement("Child2");
parent.addChildElement("Child3");
parent.addChildElement("Child4");

// Get the first child and remove it
Iterator iter = parent.getChildElements( );
SOAPElement child = (SOAPElement)iter.next( );
child.detachNode( );
```

Incidentally, it is not possible to remove more than one SOAPElement at a time by using an Iterator in this way, since the next( ) method throws a ConcurrentModificationException when called after removal of the first child element. To remove more than one SOAPElement, use the Iterator remove( ) method instead. The following code shown next, for example, removes all the children of a SOAPElement.

```
Iterator iter = parent.getChildElements( );
while (iter.hasNext( )) {
    iter.remove( );
}
```

SOAP Fault Handling

A SOAP fault is a specific type of XML element placed in the body of a reply message to convey status information or, more usually, to report a fault while processing the original request. SAAJ represents a fault using the `SOAPFault` interface that, since it must be a top-level element in the SOAP body, is derived from `SOAPBodyElement` (see Figure 3-4). The SOAP specification requires that there be no more than one SOAP fault in the message body. Faults can be mixed with other top-level elements if necessary.

SOAP faults have three attributes that can be set to provide information about the condition being reported. Each of these attributes has corresponding methods in the `SOAPFault` interface to allow its value to be set or retrieved:

Attribute	SOAPFault methods	Description
Fault code	`getFaultCode( )` `setFaultCode( )`	A code that indicates the reason for reporting the fault. Applications may define their own private fault codes or use the set of standard values defined by the SOAP 1.1 specification (described later). All fault codes must be namespace-qualified.
Fault string	`getFaultString( )` `setFaultString( )`	A human-readable description of the reason for the fault. The value of this attribute typically is written to a log file or displayed in a user interface.
Fault actor	`getFaultActor( )` `setFaultActor( )`	The URI of the participant in the SOAP message path (referred to in the SOAP specification as an *actor*) that caused the fault element to be generated.

Fault Codes

SOAP defines a small set of fault codes that can be used as appropriate. All of these fault codes belong to the same namespace as the SOAP envelope itself. The SAAJ API does not define constant values for these codes, so the application code must hardcode them.

Fault code	Description
`VersionMismatch`	This code indicates that a SOAP message was received in which the namespace for the SOAP envelope did not match the version of SOAP that the receiver supports. At the time of this writing, this means that the namespace was something other than *http://schemas.xmlsoap.org/soap/envelope/*.
`MustUnderstand`	This error is generated when a SOAP actor receives a message containing a header that contains the `mustUnderstand` attribute with value 1, which it does not know how to process. For further details, see "SOAP Headers," later in this chapter.

Fault code	Description
`Client`	Indicates that the message was improperly constructed by its originator. It is possible to more closely specify the nature of the problem (albeit in an application-dependent way) by appending one or more qualifiers to the fault code. For example, the book image service uses the fault code `Client.UnknownRequest` if it receives a message in which the top-level element of the body is not one of the requests that it recognizes.
`Server`	Indicates that a processing error occurred within the server. This might, for example, be because the resources required within the server to handle the message are not currently available. This code should not be used when the cause of the error is related to content or construction of the message. Like the `Client` fault code, it is common to return a more specific error code by appending a qualifier.

SAAJ

Fault Actor

Although in this chapter our examples have involved only a client application and the web service that it is using, in many cases, a SOAP message passes through and is processed by intermediate systems before reaching its final destination. These intermediate systems, referred to as actors, usually perform message validation or processing based on the content of SOAP headers attached to the message. If an error is detected during header processing, the actor may return a fault to the message originator, and must identify itself as the source of the report by using the fault actor attribute, which is technically a URI, but is typically the URL of the system concerned.

When an error is detected at the system that is the ultimate destination of the message, the fault actor attribute need not be set. Note, however, that the SAAJ API does not allow you to set this attribute to `null`—instead, you need to use an empty string to indicate that the fault actor is not specified.

Fault Details

In addition to the three attributes just described, a SOAP fault also requires a detail element to be present if the fault relates to the content of the message body. No detail element is permitted if the fault relates to header processing. The content of the detail element is application-dependent, but the top-level element must be namespace-qualified.

Creating Fault Elements

The book image web service generates a fault in response to a number of conditions. Later in this chapter, you'll see how the client application and the web service use SOAP attachments to transfer the cover image for one or more books, given a request containing the book titles. If the web service is asked for the image of a book whose title it does not recognize, then it generates a fault. To see an example of this, start the client using the command line:

```
ant run-client-debug
```

Now select `No Such Book` in the book title list and press the "Fetch" button. The client requests the book cover for a book whose title the web service does not know. In the command window from the start of this example, you'll see the server's reply message, which contains a SOAP fault entry in the body:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/
     envelope/"
```

```
        xmlns:tns="urn:jwsnut.bookimageservice"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        soap-env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <soap-env:Body>
        <soap-env:Fault>
            <faultcode>soap-env:Client.Title</faultcode>
            <faultstring>Unknown title</faultstring>
            <faultactor>urn:jwsnut.bookimageservice</faultactor>
            <detail>
                <tns:BookFaultDetail>No Such Book</tns:BookFaultDetail>
            </detail>
        </soap-env:Fault>
    </soap-env:Body>
</soap-env:Envelope>
```

As you can see, the `Fault` element (which is qualified with the SOAP envelope namespace) contains child elements for the fault code, a fault string, the fault actor, and the fault detail:

Fault code
: Since the error in this case results from an invalid book title in the received request, the web service uses a fault code from the `soap-env:Client` set, and qualifies it with the word "Title" to indicate that the error was with the book title. Obviously, for this to be of any use, the client and the service both need to be aware of the possible error codes and their meanings.

Fault string
: This is simply a text message that describes the error, for the purposes of logging.

Fault actor
: In this case, the fault was detected by the ultimate destination of the message, so it was not mandatory to specify the fault actor. The value supplied here is the URN of the web service.

Detail
: The `detail` element contains an application-defined child element called `BookFaultDetail` that provides further information regarding the error. This element must be namespace-qualified, and it is appropriate here to qualify it with the URN of the web service. Child elements (if there are any) of the `BookFaultDetail` element do not need to be namespace-qualified. Applications are free to include any kind of private content in the detail part.

The code used to generate this fault is shown in Example 3-10.

Example 3-10. Creating a SOAP fault element

```
/**
 * Creates a fault in the reply body.
 */
private void createFault(SOAPBody replyBody, String faultCode,
     String faultString, String faultActor, String detailString)
     throws SOAPException {
```

Example 3-10. Creating a SOAP fault element (continued)

```
    SOAPFault fault = replyBody.addFault( );
    fault.setFaultCode(faultCode);
    fault.setFaultString(faultString);
    fault.setFaultActor(faultActor);
    if (detailString != null) {
        Name detailName = soapFactory.createName("BookFaultDetail",
          SERVICE_PREFIX, SERVICE_URI);
        Detail detail = fault.addDetail( );
        DetailEntry detailEntry = detail.addDetailEntry(detailName);
        detailEntry.addTextNode(detailString);
    }
}
```

SAAJ

The `Fault` element itself is created using the `SOAPBody` `addFault( )` method:

```
public SOAPFault addFault( ) throws SOAPException;
```

Three setter methods are used to set the fault code, fault string, and fault actor. If this method is used to report a fault during header processing, it is not permissible to include a detail element; therefore, the `detailString` argument is `null`. In this case, however, the problem is with the message body, so a detail entry is mandatory. To create a detail entry and add it to the `SOAPFault` object, the `SOAPFault` `addDetail( )` method is used:

```
public Detail addDetail( ) throws SOAPException;
```

Alternatively, a `Detail` object can be obtained from `SOAPFactory`, in which case it must be explicitly added to the `SOAPFault` object:

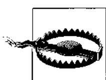
```
Detail detail = soapFactory.createDetail( );
fault.addChildElement(detail);
```

Finally, to add a top-level element to the `Detail` object, use its `addDetailEntry( )` method, passing it the fully qualified element name (in this case, `tns:BookFaultDetail`):

```
public DetailEntry addDetailEntry(Name name) throws SOAPException;
```

`DetailEntry` is a `SOAPElement`; therefore, further elements can be nested inside it (and need not be namespace-qualified), or text can be added, as shown in Example 3-10.

It is also permissible to add other application-defined elements directly to the `SOAPFault`, rather than inside the `Detail` element. These elements can be created using the `addChildElement( )` methods that `SOAPFault` inherits from `SOAPElement` (see Figure 3-4 for the inheritance hierarchy of `SOAPFault`) or using the `SOAPFactory` class in the usual way. All elements added to the `SOAPFault` element are of type `SOAPFaultElement` (from which `Detail` is derived) and must be namespace-qualified.



The SOAP specification requires that a SOAP reply message containing a `Fault` element must, when sent over HTTP, have the HTTP response code 500, which indicates an internal server error. `SAAJServlet` fulfills this requirement by inspecting the message returned by the `onMessage( )` method and setting the appropriate response code if the body contains a fault—see Example 3-1.

To make it simple to handle SOAP faults, the SOAPBody interface provides two convenience methods:

```
public boolean hasFault( );
public SOAPFault getFault( );
```

These methods remove the need for application code to search the body looking for a SOAPFault element, which may not be the first element in the SOAP body.

SOAP Messages and MIME Headers

When a SOAP message is wrapped for transmission in HTTP (or another protocol, such as SMTP), suitable MIME headers must be created. At minimum, the wrapper must include the Content-Type and Content-Length headers along with the SOAPAction header that was described in "Receiving a SOAP Message," earlier in this chapter. Similarly, when a message is received, any MIME headers that accompany it must be extracted from the protocol-specific wrapper and be made part of the created SOAPMessage. The MIME headers associated with a SOAPMessage are held in an object of type javax.xml.soap.MimeHeaders.

In terms of the structure of an encapsulated SOAP message, the MIME headers logically appear outside the envelope (as shown in Figure 3-1), where they form the protocol-specific wrapper. In the case of a SOAP message with attachments, in addition to the headers in the outer wrapper, the SOAP message part and each of the attachment parts have their own collection of MIME headers that are distinct from those of the wrapper, as shown in Figure 3-2.

The MimeHeader and MimeHeaders Classes

MimeHeader and MimeHeaders are two of four concrete classes in the java.xml.soap package. The MimeHeader class represents a single MIME header and contains the header name and its associated value, which are set at construction time and cannot subsequently be changed.

MimeHeaders is a collection of MimeHeader objects. When it is created, a MimeHeaders object is empty; headers can be added using one of the following methods:

public void addHeader(String headerName, String headerValue)
: Creates a MimeHeader object with the given header name and value, and adds it to the collection of headers.

public void setHeader(String headerName, String headerValue)
: If the MimeHeaders object does not contain a header with the given name, this method behaves in the same way as addHeader( ). Otherwise, the value for the first header in the collection that has the given name is replaced with the given value, and all other MimeHeader entries with the same name are removed.

As implied by the description of the setHeader( ) method, it is possible to have more than one header with the same name in a MimeHeaders object. The following code results in two header entries, both with the name HeaderName:

```
MimeHeaders headers = new MimeHeaders( );
headers.addHeader("HeaderName", "Value1");
headers.addHeader("HeaderName", "Value2");
```

Calling setHeader( ) method results in exactly one header with the supplied name being present in the collection, no matter how many there were before the call was made. The following line:

```
headers.setHeader("HeaderName", "NewValue");
```

replaces the MimeHeader with name HeaderName; it replaces value Value1 with the value NewValue; and it removes the MimeHeader for the entry with value Value2. Note that this method, like all of the methods of MimeHeaders, uses case-insensitive comparison when searching for headers by name, so that the names HeaderName and HEADERNAME are equivalent.

There are a number of methods that can be used to query the content of a MimeHeaders object:

public Iterator getAllHeaders( )
: Gets an Iterator over all of the headers in the collection. Each item returned by the Iterator is of type MimeHeader.

public String[] getHeader(String headerName)
: Gets all of the values associated with headers whose names match the given name. Using the MimeHeaders object just constructed as an example, the method call getHeaders("HeaderName") would return an array of two strings containing the values Value1 and Value2.

public Iterator getMatchingHeaders(String[] headerNames)
: Gets an Iterator that returns all of the MimeHeader objects that have a name that matches those in the given array.

public Iterator getNonMatchingHeaders(String[] headerNames)
: Gets an Iterator that returns all of the MimeHeader objects that have a name that does *not* match those in the given array.

Finally, there are two methods that remove entries from the collection:

public void removeAllHeaders( )
: Removes all of the MimeHeader objects, leaving an empty collection.

public void removeHeader(String headerName)
: Removes all MimeHeader objects whose name matches the supplied argument.

Setting MIME Headers When a SOAP Message Is Transmitted

When a SOAPMessage is created using the no-argument createMessage( ) method of MessageFactory, it has an empty MimeHeaders object associated with it. You can add headers at any time by retrieving this object from the SOAPMessage and using the addHeader( ) or setHeader( ) method:

```
message.getMimeHeaders( ).addHeader("Header1", "Value1");
```

The protocol-specific wrapper for a protocol such as HTTP requires that the Content-Length, Content-Type, and SOAPAction headers be included. These headers can be generated automatically by calling the SOAPMessage saveChanges( ) or writeTo( ) methods. The SOAPAction method can be explicitly set by application code and is not overwritten if it already has a value; the other two headers are always set by the saveChanges( ) and writeTo( ) methods. The SOAPMessage class

also has a convenience method called setContentDescription( ) that allows the optional Content-Description method to be set. The content description is treated as comment only.

If a SOAP message is transmitted using the call( ) method of SOAPConnection, the saveChanges( ) method is automatically called to create the appropriate headers, and the content of the resulting MimeHeaders object is copied to the outgoing HTTP message, as shown in Figure 3-6.

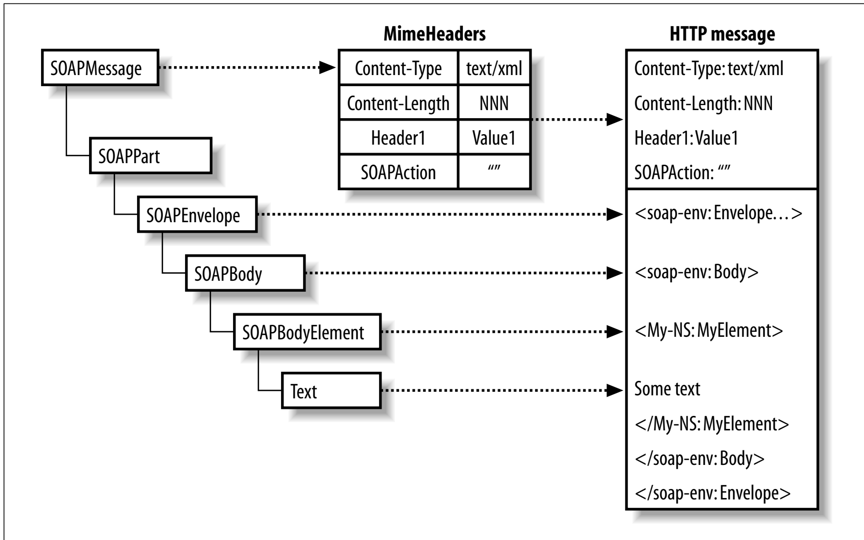


Figure 3-6. Mapping of MIME headers to HTTP headers

On the other hand, if you are implementing a server in a servlet environment and sending a reply message, you won't have a SOAPConnection, and therefore you cannot use its call( ) method to wrap the SOAPMessage in the HTTP reply. In this case, you have to manually insert the MIME headers in the HTTP reply and then write the content of the SOAPMessage to the servlet's output stream. Much of the code to handle this was shown in the doPost( ) method in Example 3-1. Note that it is first necessary to ensure that the MimeHeaders object is actually populated before copying the headers to the HTTP reply, by calling the saveChanges( ) method. Once this is done, it is a simple matter to install the headers in the reply. The code for the SAAJServlet setHttpHeaders( ) method is shown in Example 3-11. This method is invoked as follows:

```
// Copy the MIME headers to the HTTP response
setHttpHeaders(reply.getMimeHeaders( ), response);
```

where response is the HttpServletResponse object passed to the doPost( ) method.

Example 3-11. Copying MIME headers to an HTTP message

```
private void setHttpHeaders(MimeHeaders mimeHeaders,
     HttpServletResponse response) {
    Iterator iter = mimeHeaders.getAllHeaders( );
    while (iter.hasNext( )) {
```

Example 3-11. Copying MIME headers to an HTTP message (continued)

```
        MimeHeader mimeHeader = (MimeHeader)iter.next( );
        String headerName = mimeHeader.getName( );
        String[] headerValues = mimeHeaders.getHeader(headerName);

        int count = headerValues.length;
        StringBuffer buffer = new StringBuffer( );
        for (int i = 0; i < count; i++) {
            if (i != 0) {
                buffer.append(',');
            }
            buffer.append(headerValues[i]);
        }
        response.setHeader(headerName, buffer.toString( ));
    }
}
```


This method uses the SOAPMessage getAllHeaders( ) method to get an Iterator over all of the MimeHeader objects in the set of headers that is passed, and it uses the HttpServletResponse setHeader( ) method to install the header name and value corresponding to each MimeHeader. As a space optimization, if there is more than one header with the same name, this method gathers all of their values together and writes a single header in which all of the values are comma-separated. This is so that, for example, if a header with name HeaderName appears twice with values Value1 and Value2, the result is a single HTTP header that looks like this:

```
HeaderName: Value1,Value2
```

Obtaining MIME Headers When a SOAP Message Is Received

When a SOAPMessage is created from an HTTP request received by a servlet, the MessageFactory createMessage( ) method is used, as shown in Example 3-1. This method requires two arguments:

```
public SOAPMessage createMessage(MimeHeaders headers, InputStream
     inputStream);
```

The MimeHeaders argument supplies the headers to be installed in the SOAPMessage, while the InputStream is used to read the XML that makes up the SOAP envelope and, in the case of a message with attachments, the attachments themselves. The createMessage( ) method associates the supplied MimeHeaders object with the SOAPMessage so that it can be retrieved later by calling the getMimeHeaders( ) method. The Content-Type header from the collection is also used by createMessage( ) to determine whether the SOAP message is in the format shown in Figure 3-1 (where the content type is text/xml) or has attachments as shown in Figure 3-2 (and content type Multipart/Related).

The servlet has to create an appropriate MimeHeaders object from the HTTP wrapper before it can call createMessage( ). The code used by SAAJServlet to do this is shown in Example 3-12.

Example 3-12. Creating a MIMEHeaders object from an HTTP message

```
private MimeHeaders getMIMEHeaders(HttpServletRequest request) {
    MimeHeaders mimeHeaders = new MimeHeaders( );
```

Example 3-12. Creating a MIMEHeaders object from an HTTP message (continued)

```
    Enumeration enum = request.getHeaderNames( );
    while (enum.hasMoreElements( )) {
        String headerName = (String)enum.nextElement( );
        String headerValue = request.getHeader(headerName);
        StringTokenizer st = new StringTokenizer(headerValue, ",");
        while (st.hasMoreTokens( )) {
            mimeHeaders.addHeader(headerName, st.nextToken( ).trim( ));
        }
    }
    return mimeHeaders;
}
```

In this code, the HTTP headers are obtained from the HttpServletRequest object. In most cases, one MimeHeader will be created from each header in the HTTP message. However, if a header contains multiple comma-separated values, such as:

```
HeaderName: Value1,Value2
```

then a separate MimeHeader will be added for each of the values.

SOAP with Attachments

A SOAP message constructed according to the SOAP 1.1 specification can only contain data encoded as XML. The SOAP with attachments specification defines an extension of the SOAP that allows additional data items—which may or may not be XML—to be transferred along with a SOAP message. SAAJ supports SOAP with attachments and provides a simple API that allows the transfer of arbitrary data. However, as we'll see, at the detailed level this interface is not quite as easy to use as it could be.

An Example SOAP with Attachments Application

As a demonstration of the use of attachments, the book image web service uses the SAAJ API to return the cover images for one or more of the books that it knows about. To see how this works, start the client using the following command:

```
ant run-client-debug
```

When the list of book titles appears, select one or more of them by clicking on them with the mouse (hold down the Control key to select more than one), then press the "Fetch" button. The selected images will appear in the user interface, as shown in Figure 3-7, and the message used to deliver them will be written to the command window. Since the message contains the binary data for the images, you'll see some strange characters in the output window. Leaving out the image data itself, a typical message used to transfer three cover images is shown in Figure 3-8.

Before we look at the SOAP with attachments message, let's briefly look at the request message that the client sends. This message (which does not have any attachments) contains an element called BookImageRequest that represents an array

Figure 3-7. Using SOAP with attachments to transfer images

of strings, where each string is the title of a book. Here is what the body of the SOAP request looks like when requesting the cover images for three books:

```
<tns:BookImageRequest
        xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[]"
        tns:imageType="image/jpeg">
    <item>J2ME in a Nutshell</item>
    <item>Java Swing</item>
    <item>Java in a Nutshell</item>
</tns:BookImageRequest>
```

As you can see, the `BookImageRequest` element contains the standard `xsi:type` and `SOAP-ENC:arrayType` attributes that indicate that it is a string array encoded using the SOAP section 5 encoding rules. It also includes another attribute defined in the book image web service's own namespace, which specifies the format in which the image should be returned:

```
tns:imageType="image/jpeg"
```

The server supports both JPEG and GIF images, and you can select the type required using the "Use GIF Images" checkbox on the user interface.

Now let's examine the reply message shown in Figure 3-8. As you can see, it is broken down into several parts that map directly to those shown in Figure 3-2. The first part consists of the HTTP headers that wrap the entire SOAP message. These headers are generated when the message is constructed using the `saveChanges( )` method, and include a `Content-Type` header that indicates that the message is constructed according to the rules for a MIME `Multipart/Related` message. Any additional MIME headers added by application code to the `MimeHeaders` object obtained from the `getMimeHeaders( )` method of `SOAPMessage` appear here.

The boundaries between the message parts are indicated using a text string that is automatically generated when the message is constructed and are supplied as the value of the boundary attribute of the `Content-Type` header, which in this case is:

```
------=_Part_11_4712040.1027772369874
```

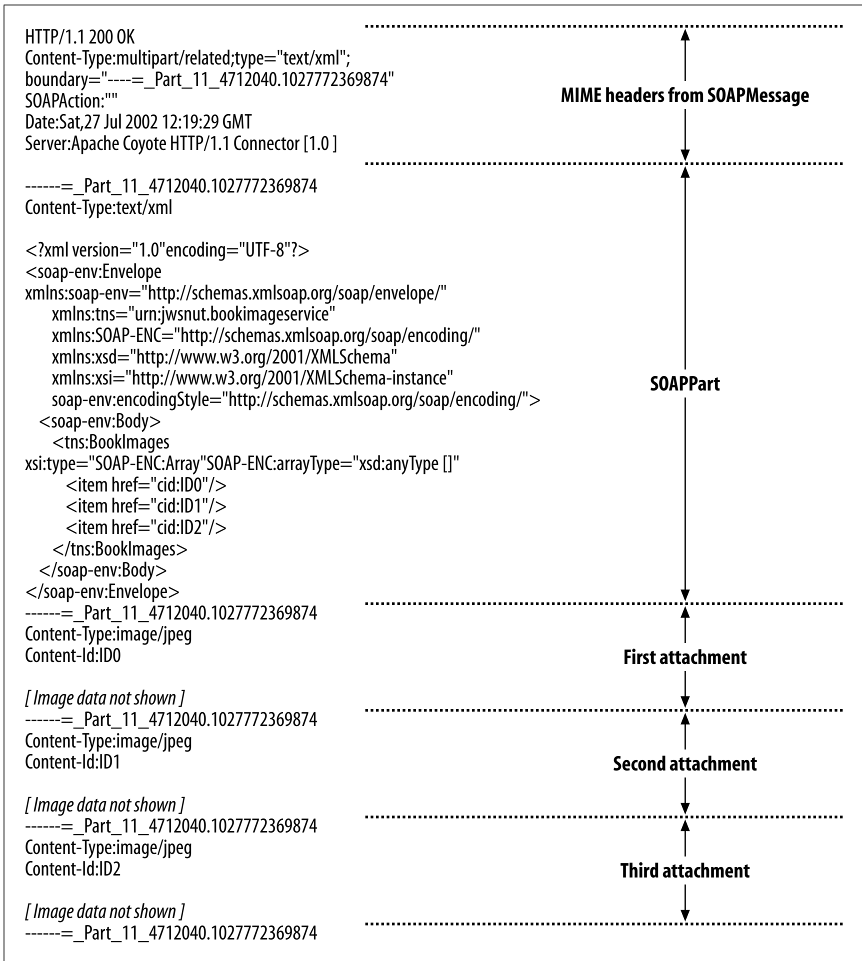


Figure 3-8. A SOAP with attachments message containing three attachments

You can find a complete description of the format of these messages in RFC 2045 and RFC 2387, both of which can be downloaded from *http://www.ietf.org*.

The second part corresponds to the part of the message represented by the `SOAPPart` object and includes the SOAP message body. As you can see, this portion of the message also has MIME headers, one of which is the `Content-Type` header that describes the content as being `text/xml`. Since the SOAP body is always XML-encoded, this is the only legal value that could appear here. The `SOAPPart` object has a group of methods that allow application code to set or query the MIME headers that it contains, which are closely related to the methods of the `MimeHeaders` object described in the previous section of this chapter. These headers are distinct from those associated with the `SOAPMessage` object and are used only when the message includes attachments.

Following the SOAPPart are the three attachments. Again, each attachment has its own collection of MIME headers that, at minimum, must include a Content-Type header that describes the type of data that the attachment contains. In this case, the header is image/jpeg because all three attachments represent a book cover image encoded in JPEG format. Each attachment also has an additional header that looks like this:

```
Content-Id: ID0
```

This header is added by the book image servlet and acts as a label that is used to refer to the attachment from within the XML part of the message, which is shown in Example 3-13.

Example 3-13. The array of images returned by the book image service

```
<tns:BookImages xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:anyType[]">
    <item href="cid:ID0"/>
    <item href="cid:ID1"/>
    <item href="cid:ID2"/>
</tns:BookImages>
```

By now, you should recognize this as a SOAP-encoded array. In this case, though, the array type is given as xsd:anyType[]. This particular specifier indicates that the element content could be any type—the application needs to examine each element to work out its actual type. This construct is essentially the same as declaring an array of Java Objects.

Another difference between this array and those that you have already seen is the actual element content is not inline with the array itself. Since the data in this case is not XML, it cannot be included directly in the message part, so we have to provide a reference that will allow the receiving application to locate the data. In this case, we choose to do so by using the href attribute, the value of which maps directly to the Content-Id of the attachment part that contains the image data apart from the inclusion of the cid:, which indicates that the value is a Content-Id. This distinguishes this representation from an alternative choice, available only for XML content, which places the value in the body of the message itself and for which the reference uses a # symbol instead of cid:, as shown here:

```
<item href="#ID0"/>
```

Here is a typical example where this style of reference is used:*

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns: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:enc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:ns0="urn:jwsnut.chapter2.bookservice/types"
        env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
```

* Another point to note is that you could try to avoid the use of href and id attributes by simply adopting the convention that the image data for the first book is in the first attachment, the data for the second is in the second attachment, and so on. However, this might not be a good idea, since although the SAAJ API includes a method that gets all the attachments for a message, its definition does not state that they are returned in the same order in which they appear in the message.

```
        <env:Body>
            <ans1:getBookInfoResponse xmlns:ans1=
              "urn:jwsnut.chapter2.bookservice/wsdl">
                <result href="#ID1"/>
            </ans1:getBookInfoResponse>
            <ns0:ArrayOfBookInfo id="ID1" xsi:type="enc:Array"
                                enc:arrayType="ns0:BookInfo[12]">
                <item href="#ID2"/>
                <!-- Additional item elements not shown -->
            </ns0:ArrayOfBookInfo>
            <ns0:BookInfo id="ID2" xsi:type="ns0:BookInfo">
                <editor xsi:type="xsd:string">Paula Ferguson, Robert Eckstein
                </editor>
                <author xsi:type="xsd:string">David Flanagan</author>
                <price xsi:type="xsd:double">39.95</price>
                <title xsi:type="xsd:string">Java in a Nutshell</title>
            </ns0:BookInfo>
            <!-- More elements not shown -->
        </env:Body>
    </env:Envelope>
```

This SOAP message is actually created by JAX-RPC, and the data represents an array of `BookInfo` objects, which were used in the previous chapter. The `result` element uses the attribute `href="#ID1"` to reference the data for the array of `BookInfo` objects that make up the result of the request, and the element that contains the data, called `ArrayOfBookInfo`, has an `id` attribute, the value of which is the referenced identifier. The same technique is used to refer to the array content from within the array.

It is important to note that there is nothing automatic about the use of `href` and `id` attributes to cross-reference one part of the message from the other: the references must be included by the application, and the receiver must use the `id` provided by the `href` attribute to find the referenced data, either in the SOAP message body as just shown or in an attachment.

Creating and Managing SOAP Attachments

The SAAJ API represents SOAP attachments using an instance of the `javax.xml.soap.AttachmentPart` class. `SOAPMessage` provides three methods to create attachments:

`public AttachmentPart createAttachmentPart( )`
: Creates an `AttachmentPart` with no associated content. To add some data to the attachment, use one of the methods described in the later section "Attachment Headers and Content."

`public AttachmentPart createAttachmentPart(Object content, String contentType)`
: Creates an `AttachmentPart` with the given content and with the `Content-Type` header set from the second argument. SAAJ makes some assumptions about the data type of the content object passed to it based on the supplied content type, as described in the next section.

`public AttachmentPart createAttachmentPart(DataHandler dataHandler)`
: Creates an `AttachmentPart` where the data for the attachment is supplied by the given `DataHandler`. See the next section for a discussion of this method.

These methods do not connect the `AttachmentPart` to the message. To do this, use the `addAttachmentPart( )` method:

```
public void addAttachmentPart(AttachmentPart part);
```

The number of attachments associated with a `SOAPMessage` can be obtained by calling the `countAttachments( )` method, whereas an `Iterator` that returns some or all of the attachments can be obtained using the following methods:

```
public Iterator getAttachments( )
public Iterator getAttachments(MimeHeaders headers)
```


The `Iterator` returned by the first of these methods will visit all of the attachments, whereas the one returned by the second method returns only those for which the MIME headers include all of those in the given `MimeHeaders` object. For example, to get all of the attachments whose data content is a JPEG image, use the following code:

```
MimeHeaders headers = new MimeHeaders( );
headers.addHeader("Content-Type", "image/jpeg");
Iterator iter = soapMessage.getAttachments(headers);
```

The SAAJ specification does not explicitly require that the order in which the attachments are returned by the `Iterator` matches the order in which they appear in the message.

The `Iterator` returned by these methods can also be used to remove selected attachments from the message. The following code removes all attachments containing JPEG images:

```
MimeHeaders headers = new MimeHeaders( );
headers.addHeader("Content-Type", "image/jpeg");
Iterator iter = soapMessage.getAttachments(headers);
while (iter.hasNext( )) {
    iter.remove( );
}
```

A quicker way to remove all attachments is to use the `removeAllAttachments( )` method.

Attachment Headers and Content

An attachment contains a collection of MIME headers and some data content. This section looks at the API that SAAJ provides for manipulating both of these parts, using the book image web service for illustration purposes.

MIME headers

Like the `SOAPMessage` and the `SOAPPart`, an `AttachmentPart` has associated MIME headers. `AttachmentPart` provides the same API for manipulating these headers (as discussed earlier in "The MimeHeader and MimeHeaders Classes"). An `AttachmentPart` always has a `Content-Type` header that reflects the data type of the

object that it contains, which is set when the AttachmentPart is created, when content is associated with it using one of the methods that we'll cover shortly, or when explicitly using the setContentType( ) method. If the latter is used, then the value supplied must match the actual type of the data in the attachment.

AttachmentPart also supplies convenience methods for setting two specific MIME headers:

```
public void setContentId(String contentId);
public void setContentLocation(String contentLocation);
```

Both of these headers are usually used to refer to the attachment from elsewhere using an element with an href attribute. The difference between these two types of identifier is as follows:

- Content-Id is typically associated with an identifier that has only local scope and is guaranteed to be unique only within the message that contains the attachment. Applications commonly use values such as ID0, ID1, etc.
- The value associated with a Content-Location is a URI (often a URL), which is more likely to be globally meaningful. If a message with an AttachmentPart containing a Content-Location header is received, it can more easily be removed and attached to another message than if Content-Ids is used, since the Content-Id in the AttachmentPart might clash with another already attached to the message.

Refer to RFC 2557 (at *http://www.ietf.org/rfc/rfc2557.txt*) for a complete discussion of the use of these identifier types. For an example that uses Content-Ids to locate attachments, see "Processing Received Attachments," later in this chapter.

It is important to realize that setting the Content-Id or the Content-Location header has no actual effect on the data in the AttachmentPart other than attaching a label to it.

Associating data with an AttachmentPart

There are four ways to install the data for an AttachmentPart:

- Use the createAttachmentPart(Object content, String contentType) method of SOAPMessage.
- Use the setContent(Object content, String contentType) method of AttachmentPart.
- Use the createAttachmentPart(DataHandler handler) method of SOAPMessage.
- Use the setDataHandler(DataHandler handler) method of AttachmentPart.

The first two are equivalent, as are the second two, so we'll discuss these two different mechanisms separately.

Specifying a value and a content type

This is the easiest way to install content in an attachment. For example, the following code installs a string value in a newly created attachment:

```
AttachmentPart part = soapMessage.createAttachment( );
part.setContent("This is in the attachment", "text/plain");
soapMessage.addAttachmentPart(part);
```

You can achieve the same result using an overloaded variant of the createAttachmentPart( ) method:

```
AttachmentPart part = soapMessage.createAttachment("This is in the
attachment", "text/plain");
soapMessage.addAttachmentPart(part);
```

However, this mechanism is not always as convenient as it may seem, because SAAJ places some requirements on the type of the content object, depending on the Content-Type that you supply. If these requirements are not met, then a SOAPException is thrown.

The content types that the SAAJ specification requires all implementations to recognize and the expected type of the Java object that must be supplied when creating the attachment are listed in Table 3-3. When the attachment contains a string, it is simple enough to make use of this API, since the content can be supplied as a java.lang.String object. Of course, this is probably among the least likely of data types to find in an attachment, and in all the other cases the developer has to work a little harder.

Table 3-3. Mapping between content type and object type for SOAP attachments

Content type	Required object type
text/plain	java.lang.String
text/xml	javax.xml.transform.stream.StreamSource
image/gif	java.awt.Image
image/jpeg	java.awt.Image

To attach XML, you can't simply read the XML into a string and pass the string value—in other words, the following does not work:

```
soapMessage.createAttachmentPart("<AnElement>Content</AnElement>",
   "text/xml");
```

Instead, you have to construct a StreamSource object:

```
soapMessage.createAttachmentPart(
        new StreamSource(new StringReader(
            "<AnElement>Content</AnElement>")), "text/xml");
```

Although this is not so convenient when you have the XML in the form of a String, it does make the task of including XML from a file or an InputStream very simple:

```
soapMessage.createAttachmentPart(new StreamSource(new File(
    "c:\\fileName.xml")), "text/xml");
```

As far as images are concerned, in order to attach one to a message, you need to have it in the form of a java.awt.Image object. While this might not seem too onerous a requirement, it probably is not the best design choice for the following reasons:

- Using the java.awt.Image class implies the use of user interface classes. This is really not a good idea in a server environment, where it should not be necessary to load and initialize any user interface classes.

- Typically, in order to get the required Image object, you will open a file containing the image encoded in byte form in either GIF or JPEG format, and then load the file content using methods provided by java.awt.Component together with a java.awt.MediaTracker. This is all unnecessary overhead, since the Image is then converted back to a byte stream when the SOAP message is transmitted.
- Finally, you cannot use this mechanism to attach an image in GIF format to a message, since the process of converting from an Image to a byte stream requires a GIF encoder, which is not supplied as part of the standard Java platform at the time of this writing (although, of course, it is possible to decode a byte stream in GIF format).

For all these reasons, if you want to include an image as an attachment, you need to use a DataHandler, as described in the next section.

Using a DataHandler

The javax.activation.DataHandler class is part of the JavaBeans Activation Framework (JAF). It is most commonly used in conjunction with JavaMail to encapsulate access to attachments sent and received with Internet mail, which is essentially the same as handling SOAP message attachments. Whenever you create an AttachmentPart while building an outgoing message, or when an AttachmentPart is created to represent part of a received SOAP message, it has an associated DataHandler.

When a SOAP message is being converted from its internal representation to the byte stream that is ultimately transmitted, the DataHandler is responsible for creating a byte stream representation of the data in the attachment. The way in which this is done depends on the type of data to convert, so the DataHandler uses objects called DataContentHandlers, each of which can perform this conversion for a specific data type. DataContentHandlers are created by an object implementing the DataContentHandlerFactory interface, which returns an appropriate DataContentHandler for a given MIME type, if it has one. The SAAJ reference implementation installs a DataContentHandlerFactory that provides handlers for the data types listed in Table 3-3. Unfortunately, the handlers created by this factory for image/jpeg, image/gif, and text/xml work only when they are asked to convert objects of the types listed in the second column of Table 3-3. When the data you need to attach is already in the form of a byte array, simply using the AttachmentPart setContent( ) method in the obvious way:

```
attachmentPart.setContent(imageData, "image/jpeg");
```

cannot be done without also changing the DataContentHandler used to handle objects with this MIME type. The only way to do this is to write and install your own DataContentHandlerFactory. (This is done by using the DataHandler setDataContentHandlerFactory( ) method), but this requires you to implememt handlers for *all* MIME types that you need to support, since there can only be one factory active at any given time and there is no way to get a reference to the existing factory so that you can delegate to it.

Fortunately, there is a simpler way to solve this problem. A DataHandler can work directly with an object that implements the javax.activation.DataSource interface. A DataSource is used when you have access to the object in a form from

which it is simple to create an InputStream and an OutputStream. When you associate a DataSource with a DataHandler, it uses the DataSource to access the underlying data instead of looking for a DataContentHandler. When you have a GIF or JPEG image in the form of an array of bytes, it is natural to use a DataSource instead of a DataContentHandler, because no further data conversion is required. To describe how this can be done, the code that attaches the book cover images to the SOAP reply for the book image service is shown in Example 3-14.

Example 3-14. Attaching images to a SOAP message

```
private void handleBookImageRequest(SOAPElement element, SOAPMessage reply)
                                        throws SOAPException {
  // The request element contains an attribute that holds the
  // type of image requested and a nested string for each title.
  // The reply body has a BookImages element and a nested item with
  // a reference to the image which is sent as an attachment
    SOAPBody replyBody = reply.getSOAPPart().getEnvelope( ).getBody( );

    // Determine whether to use JPEG or GIF images
    String imageType = element.getAttributeValue(IMAGE_TYPE_ATTRIBUTE);
    boolean gif = imageType.equalsIgnoreCase("image/gif");

    // Build the BookImages element containing all of the replies
    SOAPBodyElement bodyElement = replyBody.addBodyElement(BOOK_IMAGES_NAME);
    bodyElement.addAttribute(
        soapFactory.createName("type", XMLSCHEMA_INSTANCE_PREFIX,
           XMLSCHEMA_INSTANCE_URI), SOAP_ENC_PREFIX + ":Array");
    bodyElement.addAttribute(
        soapFactory.createName("arrayType", SOAP_ENC_PREFIX,
           SOAPConstants.URI_NS_SOAP_ENCODING), XMLSCHEMA_PREFIX + ":anyType[]");

    // Index of the next attachment to use
    int index = 0;

    // Handle each nested element.
    Iterator iter = element.getChildElements( );
    while (iter.hasNext( )) {
        // Get the next child element from the request message
        SOAPElement childElement = (SOAPElement)iter.next( );

        // Get the book title
        String title = childElement.getValue( );

        // Get the image data
        byte[] imageData = BookImageServletData.getBookImage(title, gif);
        if (imageData != null) {
            // Got the data - attach it.
            AttachmentPart attach = reply.createAttachmentPart( );
            attach.setDataHandler(new DataHandler(
                                new ByteArrayDataSource("Image Data",
                                    imageData,
                                    gif ? "image/gif" : "image/jpeg")));
            attach.setContentId("ID" + index);
            reply.addAttachmentPart(attach);
```

Example 3-14. Attaching images to a SOAP message (continued)

```
            // Add an element in the reply pointing to the attachment
           bodyElement.addChildElement("item").addAttribute(HREF_ATTRIBUTE,
             "cid:ID" + index);

            // Increment the index
            index++;
        } else {
            // No data - this is a fault.
            // Clear the reply and install the fault
            reply.removeAllAttachments( );
            bodyElement.detachNode( );
            createFault(replyBody, "soap-env:Client.Title", "Unknown title",
                                   SERVICE_URI, title);
            return;
        }
    }
}
```

This code loops over all of the book titles in the client request message and adds an element to the body of the reply that points to a corresponding attachment, using an `href` attribute with the `Content-Id` of the attachment as the reference value, as shown in Example 3-13. It uses a separate class (`BookImageServletData`, not shown here) to get the data for a book with a given title in the form of an array of bytes that is encoded in either GIF or JPEG form. The section of this code that is relevant to this discussion is highlighted in bold. The first step is to create the `AttachmentPart`:

```
AttachmentPart attach = reply.createAttachmentPart( );
```

To associate the image data with the attachment, we need to replace the default `DataHandler` with one that uses a `DataSource` that takes its content from the image data. Unfortunately, there is no `DataSource` implementation that accepts a byte array as its input, but it is simple enough to create one, as shown in Example 3-15.

Example 3-15. A DataSource that encapsulates access to data in a byte array

```
class ByteArrayDataSource implements DataSource {

    private String contentType;

    private byte[] data;

    private String name;

    ByteArrayDataSource(String name, byte[] data, String contentType) {
        this.name = name;
        this.data = data;
        this.contentType = contentType;
    }

    public String getContentType( ) {
```

Example 3-15. A DataSource that encapsulates access to data in a byte array (continued)

```
        return contentType;
    }

    public InputStream getInputStream( ) throws IOException {
        return new ByteArrayInputStream(data);
    }

    public String getName( ) {
        return name;
    }

    public OutputStream getOutputStream( ) throws IOException {
        throw new IOException(
            "ByteArrayDataSource cannot support getOutputStream( )");
    }
}
```


From the point of view of this example, the DataSource is only required to implement the getContentType( ) method (to return the MIME type of the data that it provides) and the getInputStream( ) method (which needs to return an InputStream to provide access to the data passed to it at construction time). This is easily achieved by wrapping the byte array containing the data in a ByteArrayInputStream. It would be simple to generalize this to also allow the data to be modified by implementing the getOutputStream( ) method, but that is not required here.

Given this DataSource, the image data can be associated with the attachment and then added to the SOAP reply message as follows:

```
attach.setDataHandler(new DataHandler(
                        new ByteArrayDataSource("Image Data",
                            imageData,
                            gif ? "image/gif" : "image/jpeg")));
attach.setContentId("ID" + index);
reply.addAttachmentPart(attach);
```

When the completed reply message is being converted into a byte stream for transmission, the content of the attachments are always read from their DataHandlers. Since we constructed the DataHandler for each attachment with a DataSource, instead of trying to find a DataContentHandler for the attachment's associated MIME type, the data is obtained directly from the input stream returned by the getInputStream( ) method of the ByteArrayDataSource. Not only is this more convenient than converting the data into an Image, it is also much more efficient, and better still, it allows GIF images to be supported. Since there is no need for a GIF encoder to be present, the encoding has already taken place when the image data was created. This web service can now support any type of image, since it simply treats the image data as an opaque byte stream. Whether this is useful depends on the capabilities available to the client to decode the data and display it—obviously, it is not useful to send an image in PNG format to a client that cannot decode PNG images.

This same technique could be used for any type of data for which you have a byte stream representation, for content types that are not listed in Table 3-3, and for which, therefore, the `AttachmentPart setContent( )` method will not work. If, for example, you have an array of bytes that represents an audio clip in *.wav* format (a format that is not directly supported in the reference implementation), you can associate it with an attachment as follows:

```
byte[] audioData = ......; // Load sound bytes (not shown)
 attach.setDataHandler(new DataHandler(
                                     new ByteArrayDataSource("Audio Data",
                                         audioData, "audio/wav")));
```

As well as installing your own `DataHandler` to make use of a custom `DataSource`, you can also use its other two constructors to attach data to a SOAP message:

```
public DataHandler(Object object, String mimeType);
public DataHandler(URL url);
```

The first constructor is of little use because it is equivalent to using the `AttachmentPart setContent( )` method. The second constructor can be used to import data from a given URL, where it is assumed that the `URL getContent( )` method returns an object whose type is consistent with the content types listed in Table 3-3.

Processing Received Attachments

When a SOAP message containing one or more attachments is received, the `SOAPMessage` object will have one `AttachmentPart` per attachment. The interpretation placed on these attachments and the way in which they relate to the XML in the body of the message is, of course, application-dependent. In the case of the book image service, as we saw in Example 3-13, each image returned in response to a `BookImageRequest` has its own entry in the `BookImages` array that appears in the body of the SOAP message sent by the web service to the client. The array entry is bound to the attachment containing the image data by an `href` attribute that contains the value of the `Content-Id` MIME header of the appropriate attachment:

```
<item href="cid:ID0"/>
```

To extract all of the images, the book image client loops over all of the elements in the array, extracts the `href` attribute for each, and finds the `AttachmentPart` that has the corresponding `Content-Id`, as shown in Example 3-16.

Example 3-16. Handling attachments in a SOAP message

```
SOAPBody replyBody = reply.getSOAPPart().getEnvelope( ).getBody( );
if (replyBody.hasFault( )) {
    SOAPFault fault = replyBody.getFault( );
    throw new SOAPException("Fault when getting book images: " +
                    fault.getFaultString( ) +
                     ", actor is [" + fault.getFaultActor( ) + "]");
}

// The body contains a "BookImages" element with a nested
// element for each book title.
```

Example 3-16. Handling attachments in a SOAP message (continued)

```
Iterator iter = replyBody.getChildElements(BOOK_IMAGES_NAME);
if (iter.hasNext( )) {
    ArrayList list = new ArrayList( );
    MimeHeaders headers = new MimeHeaders( );
    SOAPElement bookImages = (SOAPElement)iter.next( );
    iter = bookImages.getChildElements( );
    while (iter.hasNext( )) {
        SOAPElement element = (SOAPElement)iter.next( );
        String imageRef = element.getAttributeValue(HREF_ATTRIBUTE);
        if (imageRef != null) {
            // Get the attachment using the Content-Id, having
            // first removed the "cid:" prefix
            imageRef = imageRef.substring(4);
            headers.setHeader("Content-Id", imageRef);
            Iterator attachIter = reply.getAttachments(headers);
            if (attachIter.hasNext( )) {
                AttachmentPart attach = (AttachmentPart)attachIter.next( );
                Object content = attach.getContent( );
                if (content instanceof Image) {
                    list.add(content);
                }
            }
        }
    }
    int size = list.size( );
    Image[] images = new Image[size];
    list.toArray(images);
    return images;
} else {
    // No BookTitles element was found
    throw new SOAPException("No BookImages element in returned message");
}
```

As each child element of the array is found, its `href` attribute is obtained. Since this is actually in the form `cid:ID0`, it is necessary to first strip away the leading `cid:` to obtain the value that is used for the `Content-Id` header:

```
String imageRef = element.getAttributeValue(HREF_ATTRIBUTE);
if (imageRef != null) {
    // Get the attachment using the Content-Id, having
    // first removed the "cid:" prefix
    imageRef = imageRef.substring(4);
```

As noted earlier, the SAAJ API provides two `SOAPMessage` methods that return the attachments for the message:

```
public Iterator getAttachments( )
public Iterator getAttachments(MimeHeaders headers)
```

There is no direct way to find a single attachment given a content id string. Instead, the second `getAttachments( )` method provides a general facility that returns all attachments that have a given set of MIME headers. Here, we use it to

locate the correct AttachmentPart by looking for the attachment in which the Content-Id header has the value extracted from the message body:

```
MimeHeaders headers = new MimeHeaders( );
headers.setHeader("Content-Id", imageRef);
Iterator attachIter = reply.getAttachments(headers);
if (attachIter.hasNext( )) {
     AttachmentPart attach = (AttachmentPart)attachIter.next( );
```

Given the way that this message is constructed, we expect there to be only one AttachmentPart per element in the message body, but this isn't necessarily the case: it is possible to attach more then one object with the same Content-Id, perhaps to supply the image and an associated sound file containing some marketing information for the book. The two objects would, of course, be distinguished by the content types, which can be obtained from the AttachmentPart using its getContentType( ) method.

Once you have the AttachmentPart, you can get its content using the following method:

```
public Object getContent( );
```

The actual type of the returned object is determined by the attachment's content type and the DataContentHandlers that are installed. In the case of the reference implementation, the mapping from content type to Java object type is as shown in Table 3-3, which means that, in this example, the getContent( ) method should return a java.awt.Image containing the image data extracted from the attachment.

Content types for which a DataContentHandler is not found are returned as an InputStream from which the raw byte data can be read. In the case of the reference implementation, for example, a sound file sent with type audio/wav or a byte stream of type application/octet-stream is returned in this way.

Another way to extract the data from an attachment is to use the DataHandler that is created for each AttachmentPart. Using the DataHandler, you can bypass the DataContentHandlerFactory and get direct access to the raw attachment data, even if the content type is recognized by the SAAJ implementation that you are using. Here, for example, is some code that will create an Image object from any of the image types recognized by the JRE that you are using, even if the SAAJ implementation does not provide a DataContentHandler for it. At the time of this writing, this code works for PNG images as well as those in GIF or JPEG format:

```
if (attach.getContentType( ).startsWith("image/")) {
    javax.activation.DataHandler handler = attach.getDataHandler( );
    InputStream is = handler.getInputStream( );
    int count = is.available( );
    byte[] buffer = new byte[count];
    is.read(buffer, 0, count);
    Image img = java.awt.Toolkit.getDefaultToolkit( ).createImage(buffer);
}
```

Notice that we use the available( ) method of InputStream to find out how much data there is in the attachment, even though AttachmentPart has a getSize( ) method. The reason for this is the AttachmentPart method always seems to return a number that is larger than the actual size of the data.

SOAP Headers

So far, we have only concerned ourselves with the body of a SOAP message and its attachments, both of which are intended solely for the ultimate destination of the message. A message can also include headers, which contain information that is related to the routing or processing of the message, but which are not part of the payload itself. SOAP itself does not define any standard headers, but there are specific uses of SOAP that do, some of which are discussed in Chapter 4. In this section, we'll look in a generic way at the facilities provided by SAAJ for creating and handling headers, and defer detailed discussion of real-world uses of this facility until Chapter 4.

SOAP Actors and the mustUnderstand Attribute

Although the examples that we have shown so far involve only a client and a web service to which the client sends a SOAP message, in practice, a SOAP message may pass through one or more intermediate systems before reaching its ultimate destination. For the purposes of this discussion, an intermediate system is not like a network router, which is concerned only with passing a transport-level packet from node to node until it reaches a destination, but an application-level entity that receives a SOAP message, examines it, and either handles it or forwards it to another SOAP receiver for further processing.

As an example of the use of an intermediate system, suppose that a business wants to make it possible for other businesses to place stock orders by building a SOAP message that contains a purchase order, in response to which it expects confirmation of the order and payment details. Before the order is processed, it might be necessary to check that the initiator is known to the company and has a purchase account to which the transaction can be billed. In order to implement this, the business might separate the handling of the account check from the handling of the purchase order, as follows:

1. When the client sends the purchase order, it is required to include a header that identifies it in a way that is determined by the business providing the service. In practice, this would probably involve the use of some secure mechanism, such as public key cryptography and the inclusion of a certificate to identify the client.
2. The system to which the purchase order is sent looks for this header, extracts the required information, and verifies it. If it is not valid, a fault is returned to the originator. Otherwise, the header is removed and perhaps replaced with one that provides information about the initiator that is in a form that is more useful to the business itself, such as account details.
3. The message is now forwarded to an internal system that handles the purchase order and ultimately returns a response to the order itself.

In the SAAJ API, each `SOAPMessage` can have a single `SOAPHeader` element. By default, this element is empty. Headers are created by adding XML elements to the `SOAPHeader`. Each such header element must be namespace-qualified and may contain an attribute that identifies the system that is intended to process it, known as the *actor* attribute. The value of this attribute is defined to be a URI

and, in practice, it is often a URL, although it need not be. There are three cases to consider:

The actor attribute is missing
: When there is no actor attribute, the header is deemed to be intended for the ultimate recipient of the message.

The actor attribute has the distinguished value http://schemas.xmlsoap.org/soap/actor/next
: A header containing an actor attribute with this value (which a SAAJ application can conveniently refer to by using the constant value `SOAPConstants.URI_SOAP_ACTOR_NEXT`) is intended for the first system that receives it.

The actor attribute has some other value
: In this case, the header is intended for the system whose URI matches the value of the attribute.

When a SOAP message is received, the headers must be checked for any that are intended for the recipient. If there are any, the following rules apply:

- The header must be removed from the message.
- If the receiver understands the header, then it may process the header and act upon it. As a result, it may add one or more new headers to the message (which may be identical to the received header). If the header contains the `mustUnderstand` attribute from the SOAP envelope namespace, and the value of this attribute is 1, the receiver must process the header—or else return a fault to the originator if it chooses not to do so or cannot do so for some reason.
- If the receiver does not understand a header that is addressed to it, it may choose to silently ignore it, unless the `mustUnderstand` attribute is present and has the value 1, in which case it must return a fault to the originator.

If a fault is returned as a result of the `mustUnderstand` attribute, then the fault code must have the value `soap-env:MustUnderstand` and the fault actor must be set to the URI of the system that generated the fault. A system that processes a header may also generate a fault for reasons that are related to the content of the header. In all cases, a fault that arises from header processing may *not* include a detail element.

Creating Header Elements

Header elements are created using the `SOAPHeader addHeaderElement( )` method, which requires a `Name` object from which the element name and its namespace are determined. The SOAP specification requires that header elements are namespaced-qualified. The `addHeaderElement( )` method returns a newly created `SOAPHeaderElement`, which was added as a direct child of the `SOAPHeader`. Example 3-17 shows how to add a header that contains child elements for a username and a password.

Example 3-17. Adding a header entry to a SOAP message

```
SOAPHeader header = message.getSOAPPart().getEnvelope( ).getHeader( );
SOAPHeaderElement headerElement = header.addHeaderElement(
    soapFactory.createName("AuthInfo"));
```

Example 3-17. Adding a header entry to a SOAP message (continued)

```
headerElement.addNamespaceDeclaration(null, "urn:headerDemo");
headerElement.setMustUnderstand(true);
headerElement.addChildElement("UserName").addTextNode("JWSUserName");
headerElement.addChildElement("Password").addTextNode("JWSPassword");
```

The header element is called `AuthInfo` and its namespace is `urn:headerDemo`, which is an arbitrarily chosen URN. The following method call:

```
headerElement.addNamespaceDeclaration(null, "urn:headerDemo");
```

makes this namespace the default for this element and its subelements, so that the element names will appear without a namespace prefix. The use of the default namespace in this way is a convenience and not a requirement.

The `SOAPHeaderElement` interface has four methods that provide easy access to the SOAP actor and `mustUnderstand` attributes:

```
public void setActor(String actorURI);
public String getActor( );
public void setMustUnderstand(boolean cond);
public boolean getMustUnderstand( );
```

The code in Example 3-17 sets the `mustUnderstand` attribute to `1` (by virtue of the fact that its argument has the value true), but does not set the actor attribute at all, which implies that the header is to be processed by the ultimate recipient of the message, and must be understood and properly actioned or a fault returned. Here is what a SOAP message produced by this code looks like:

```
<soap-env:Envelope xmlns:soap-env=
    "http://schemas.xmlsoap.org/soap/envelope/">
    <soap-env:Header>
        <AuthInfo xmlns="urn:headerDemo" soap-env:mustUnderstand="1">
            <UserName>JWSUserName</UserName>
            <Password>JWSPasswordJWSPassword</Password>
        </AuthInfo>
    </soap-env:Header>
    <soap-env:Body/>
</soap-env:Envelope>
```

The default value of the `mustUnderstand` attribute is `0`, which means that the header can be removed without being processed by its target.

Processing Header Elements

When a SOAP message is received, there are three ways to handle the header entries that it contains once a reference to the `SOAPHeader` has been obtained:

- Get an iterator over all of the child elements of the `SOAPHeader`. This returns all of the `SOAPHeaderElement` objects in the message.
- Get an iterator over all of the headers intended for a given actor. This returns a subset of the `SOAPHeaderElement` objects in the message.
- Get an iterator over all of the headers intended for a given actor and also remove them from the message.

A message recipient must process all headers for which the actor attribute is set to `next` or to its own URI, and must remove them once they have been processed. The `SOAPHeader` interface provides two methods that locate the headers intended for a given actor URI:

```
public Iterator examineHeaderElements(String actorURI);
public Iterator extractHeaderElements(String actorURI);
```

The difference between these two methods is that the second removes all of the headers that appear in the `Iterator` from the message, whereas the first does not. Calling either of these methods with the argument `SOAPConstants.URI_SOAP_ACTOR_NEXT` returns all of the headers that have the actor attribute set to `next` and all of those that do not have an actor attribute.* As an alternative to these two methods, you can get an `Iterator` over all of the header entries without removing any of the them, using the `getChildElements( )` method that `SOAPHeader` inherits from `SOAPElement`:

```
// Gets all the SOAPHeaderElements
Iterator  iter = soapHeader.getChildElements( );
```

Using SAAJ with Secure Connections

The web service examples used so far in this book have not attempted to provide any security measures to ensure that they can only be accessed by authorized users, or to provide some level of assurance to the client that the server to which it might be about to pass sensitive information is the one to which it thinks it is connected. At the time of this writing, security for web services is the subject of several in-progress JSRs, such as JSR 105 (XML Digital Signature APIs; see *http://jcp.org/jsr/detail/105.jsp*) and JSR 106 (XML Digital Encryption APIs; see *http://jcp.org/jsr/detail/106.jsp*). Until these JSRs are completed and their implementations become part of the Java platform, you can still make use of the authentication mechanisms already provided for HTTP to add a level of security to your service. In this section, you'll see how to configure the client and server parts of the service to use both HTTP basic authentication, which is relatively weak, and HTTPS, which is much more robust but is slightly more difficult to set up.

Using Basic Authentication

HTTP basic authentication is a simple mechanism that requires the client to supply a username and password to gain access to a service. The authentication information is encoded and sent in an HTTP header to the server, which can then verify whether the user should have access to the service at the URL specified in the request. Although it is easy to configure basic authentication, it is a very weak mechanism because the data exchanged by the client and server is not encrypted and there is no protection against unauthorized modification. Furthermore, the algorithm used to encode the username and password is trivial, thus making it a

* It is not clear from the SAAJ specification whether this is the intended behavior or just a bug. In my opinion, it would be better if this call returned only those headers with the actor attribute set to `next`.

simple matter for a snooper to discover what they are. Nevertheless, in an internal corporate network, basic authentication may be sufficient.*

To demonstrate how to use basic authentication, we'll add it to the book image web service. Configuring basic authentication requires three steps:

1. Define the role or roles that are allowed access to some or all of the web service and the set of users that belong to those roles.
2. Define the URLs within the web service that require protection, and specify which roles should be able to use them.
3. Configure the web service client to send the appropriate authentication information when accessing the service.

SAAJ

Adding roles and users to the web container

The first step involves setting up authentication information for the web container that hosts the web service. The details of this process for the J2EE 1.4 reference implementation and the Tomcat web container in the JWSDP are covered in Chapter 1, where we added a role called `JWSGroup` together with a couple of users allowed to access that role.

Adding URL-based protection

Defining a new role does not add any protection. To achieve this, it is necessary to include authorization information in the *web.xml* file for the web service. The authorization information defines which of the web service's URLs are to be protected and which role or roles are to be allowed to access those URLs. Allowing a role to access a URL has the effect of making it possible for all of the users in that role to access the URL, provided that they can authenticate themselves to the web container by supplying the correct password.

The book image web service can be accessed without requiring authentication at almost any URL that starts with *http://localhost:8000/BookImageService*. In order to illustrate basic authentication, the *web.xml* file for this service is also configured so that only users in the `bookimageservice` role can acccess the protected URL *http://localhost:8000/BookImageService/basicAuth*. The portion of the *web.xml* file that specifies this restriction is shown in bold in Example 3-18.†

Example 3-18. Adding protected URLs to a web service

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE web-app
```

* Basic authentication is what is happening when you log onto a web site for which you preregister for access to member-only areas. The dialog box that pops up to get your username and password is the browser's way to get the information required for the authentication header in the HTTP request.

† The result of this configuration is rather unusual because the web service now has both protected and unprotected URLs, all of which provide access to the same service. In reality, you are most likely to protect all access to the web service by precisely defining which URLs it responds to and mapping them all to the appropriate roles. Here, both a protected and an unprotected service are provided so that we can use it to illustrate the SAAJ APIs without having to first introduce basic authentication.

Example 3-18. Adding protected URLs to a web service (continued)

```
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">


<web-app>
    <display-name>SAAJ Book Image Service</display-name>
    <description>SAAJ Book Image Service</description>

    <servlet>
        <servlet-name>BookImageService</servlet-name>
        <display-name>Servlet for the SAAJ book Image Service</display-name>
        <servlet-class>ora.jwsnut.chapter3.bookimageservice.BookImageServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>BookImageService</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>SAAJ Book Image Service</web-resource-name>
            <url-pattern>/basicAuth/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>JWSGroup</role-name>
        </auth-constraint>
    </security-constraint>

    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>Book Image Service</realm-name>
    </login-config>

</web-app>
```

The `security-constraint` element specifies which URLs are to be protected (using the `url-pattern` element) and the role or roles allowed access to them (in the `auth-constraint` element). It is also necessary to specify how the identity of the caller is to be determined. This is achieved by the `login-config` element, which requires the use of basic authentication.

You can verify that this constraint is active by attempting to access the service with your web browser. With the J2EE 1.4 server running, point your browser at the URL *http://localhost:8000/BookImageService*. You should see an error page resulting from the fact that the service does not support access using the HTTP GET method. If, however, you use the URL *http://localhost:8000/BookImageService/basicAuth*, you are instead prompted to enter a username and password. Only after you correctly enter these do you see the same error page. If you are using the JWSDP with the Tomcat web server, use port number 8080 instead of 8000 in these URLs.

Setting up the client

When it is run using the Ant target `run-client`, the Java client for the book image service that we used earlier in this chapter uses the URL *http://localhost:8000/BookImageService* to access the web service; therefore, it does not need to supply a username and password. The client gets the URL from its command line, so it is possible to arrange for it to access the protected URL *http://localhost:8000/BookImageService/basicAuth* instead. If you do this, however, you get an exception, since the web server expects to receive a username and a password to validate access to this URL, and refuses access if it does not get them (or if they are incorrect). SAAJ uses a slightly different URL syntax to allow the username and password to be included with the URL, which looks like this:

> *http://username:password@host:port/path*

In the case of the book image service, for a user called `JWSUserName` with the password `JWSPassword`, the appropriate URL is:

> *http://JWSUserName:JWSPassword@localhost:8000/BookImageService/basicAuth*

The Ant project file for this example includes a target that can be used to run the client with this URL. To try it out, open a command window, make *chapter3\bookimageservice* (relative to the installation directory of the book's example source code) your working directory, and type the command:

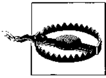
```
ant run-basicauth-client
```

Since `JWSUserName` has access to the role `JWSGroup`, the web container allows the client access to the service, based on the `auth-constraint` element in the *web.xml* file. To prove that access by unauthorized users will be refused, change the value of the `USERNAME` property in the *jwsnutExamples.properties* file in your home directory so that it does not correspond to a valid user, and then run the example again.

Using HTTPS

If you need more security than basic authentication can provide (as you almost certainly will if you intend to publish your web service on the Internet), you can arrange for it to be accessible over HTTPS instead of HTTP. By using HTTPS, you ensure that all the data exchanged between the client and the server is encrypted and protected against unauthorized modification. The client can also be sure that the server it is actually connected to is the one that it thinks it is connected to, and that the server itself can be trusted. This assurance is possible because the SSL protocol used by HTTPS delivers the server's certificate to the client. This certificate can be verified against a set of trusted certificates held by the client. The certificate also contains information, such as the server's hostname, which can be checked to ensure that the certificate belongs to the server that sent it.

In order to use HTTPS, you need to have the Java Secure Sockets Extension (JSSE) installed on both the client and server systems. If you are using Java 2 Version 1.4, JSSE is part of the core distribution so you do not need to take any extra steps. Otherwise, you should download and install the latest version of the JSSE from *http://java.sun.com* before proceeding.



When performing the following configuration steps, it is important to remember that the client and server will, in general, run on different hosts, although for testing purposes they might reside on the same system. In the descriptions that follow, we will use the names *serverhost* and *clienthost* to distinguish these two different hosts. You should, of course, substitute the names of your machines wherever these terms are used.

Enabling HTTPS in the web server

The means by which you enable HTTPS support in your web server is vendor-dependent. In this section, we describe how to do this for the Tomcat web server distributed with the JWSDP. If you are using a third-party web server, you need to consult your vendor's documentation for the appropriate procedure. In particular, if you are using the J2EE 1.4 reference implementation instead of the JWSDP Tomcat web server, then you can skip this section because it has HTTPS enabled by default.

In the following descriptions, we use the shorthand *${EXAMPLES_ROOT}* to refer to the directory in which you installed the example source code for this book, and we use *${JAVA_HOME}* for the installation directory of J2SE. The first step is to create the certificate that the web server sends to any client that connects to it over HTTPS. To create this certificate, open a command window on *serverhost*, make *${EXAMPLES_ROOT}* your working directory, and type the following command (all on one line):

```
keytool -genkey -alias server -keyalg RSA -keypass changeit
-storepass changeit -keystore server.keystore  -dname "CN=serverhost"
```

The certificate is created in a new keystore whose name is given by the *-keystore* argument, which in this case is stored in the installation directory of the example source code for the sake of convenience. The *-storepass* argument supplies the password used to access the keystore. Here, we use the default JRE keystore password. You can choose a different one, as long as you supply the same password when configuring the server to access the keystore, as described shortly.

The *-dname* argument can be used to supply a set of attributes that identify the certificate and its owning organization. Here, we set only the *CN* attribute, which specifies the name of the host to which the certificate belongs. It is important that you use the correct hostname because the client may extract this attribute and check that it matches the name of the host to which it thinks it is connected.*

The *keytool* command creates a self-signed certificate, which is, strictly speaking, appropriate only for development and testing purposes. In the real world, you should instead apply to a certificate authority for a properly signed server certificate, and then import it into the keystore using the *keytool -import* argument. For further details on the use of the *-import* argument (and on the *keytool* command in general), see *Java in a Nutshell*, by David Flanagan (O'Reilly).

* Obviously, in a real-world environment, it is not a good idea to use `CN=localhost`, but this might be appropriate for testing purposes.

The second step required to activate HTTPS is to enable it in the Tomcat web server by editing the *server.xml* file, which you'll find in its *conf* directory. Open this file in an editor and add the lines shown in bold in the following code section, substituting the pathname of the example source code installation directory in the value of the keyStoreFile attribute:

```
<Service className="org.apache.catalina.core.StandardService" debug="0"
         name="Java Web Services Developer Pack">
  <Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
acceptCount="10" ........>
    <Factory className="org.apache.catalina.net.DefaultServerSocketFactory"
/>
  </Connector>

  <!-- Added for SSL -->
  <Connector className="org.apache.catalina.connector.http.HttpConnector"
         port="8443" minProcessors="5" maxProcessors="75"
         enableLookups="false"
         acceptCount="10" connectionTimeout="60000" debug="0"
         scheme="https" secure="true">
       <Factory className="org.apache.catalina.net.SSLServerSocketFactory"
           keystoreFile="${EXAMPLES_ROOT}\server.keystore"
           keystorePass="changeit"
           clientAuth="false" protocol="TLS"/>
  </Connector>
  <!-- End of SSL section -->
```

Note the following:

- SSL is enabled by adding a second Connector element to the Service element for "Java Web Services Developer Pack." There is also another Service element in this file for internal services—make sure you modify the correct Service element.
- The filename supplied using the keystoreFile attribute must be the name of the keystore in which the certificate was created (or imported) by *keytool*. Similarly, the correct keystore password must be supplied using the keystorePass attribute.
- HTTPS is enabled on port 8443, rather than port 8080. This is one of the two port numbers used by convention for HTTPS; the other is 443 (which, on Unix-based systems, is accessible only to privileged processes).

Having completed these steps, you can check that all is well by restarting the Tomcat web server and pointing your browser at the URL *https://serverhost:8443* (note that the protocol is *https* instead of *http* and the port number is 8443). Your browser will probably ask you to confirm that you accept the server's certificate and will then display the web server's home page. If you have not obtained and installed a certificate from a certificate authority, you will probably be warned that the certificate is from a company that you have chosen not to trust. There is no need to be concerned about this—it is the result of using a self-signed certificate, which relies upon itself for its trust (and therefore cannot be trusted at all).

Setting up the client system

Now let's move to the client system. In order to use the book image web service client with HTTPS, you have to give it the appropriate URL. If you want to use basic authentication together with HTTPS, use the following URL:

> *https://JWSUserName:JWSPassword@serverhost:7000/BookImageService/basicAuth*

To use HTTPS on its own, use:

> *https://serverhost:7000/BookImageService*

Port number 7000 is used because this is the port on which the J2EE reference implementations listen for HTTPS connections. If you are using the JWSDP with the Tomcat web server, then the port number is 8443 instead of 7000.

When you use either of these URLs, the client connects over HTTPS and expects to receive the server's certificate and validate it. How does the validation work? The complete process is complex and not of any great interest from a web service development viewpoint. However, one of the following two conditions must hold:

- The server's certificate must be installed in a keystore on the client machine to which the client application has access.
- The server's certificate must be issued by a trusted authority whose certificate is installed in the client machine's keystore. In this case, the certificate itself does not need to be in the client's keystore.

In the first case, the certificate for the issuing authority is almost certainly found in the certificate store that is supplied with the JRE, which can be found at *${JAVA_HOME}\jre\lib\security\cacerts*; therefore, there is not any further work to do on the client system. If you created your own self-signed certificate, then you need to import it into a keystore that is accessible to the client.

Although you could import certificates directly into the JRE keystore, we will instead create and use a private keystore in order to demonstrate how simple it is to do this. This also has the advantage that you can experiment with certificates without the possibility of damaging your JRE. On the client machine, copy the *cacerts* file from the JRE to the installation directory of this book's source code and rename it *client.keystore*:

```
copy  ${JAVA_HOME}\jre\lib\security\cacerts
     ${EXAMPLES_ROOT}\client.keystore
```

Next, if you are using a self-signed certificate, you need to get a copy of it and import it into the newly created keystore; you can skip this step if the server is using a certificate issued by a trusted authority. To get the certificate, go to the server machine and proceed as follows:

- If you are using the JWSDP Tomcat web server, go to the directory containing the keystore (*server.keystore*) created in the previous section and type the command:

  ```
  keytool -export -alias server -storepass changeit -keystore server.
      keystore  -file server.cer
  ```

- If you are using the J2EE reference implementation, go to the directory *lib\security* below the J2EE installation directory and type the command:

```
keytool -export -alias server -storepass changeit -keystore keystore.jks
     -file server.cer
```

Copy the newly created file *server.cer* from the server machine to the *${EXAMPLES_ROOT}* directory on the client machine and import it there using the following command:

```
keytool -import -v -trustcacerts -alias JWSNutshell -storepass changeit
     -keystore client.keystore -file server.cer
```

Reply when asked if you want to trust this certificate. Note that you can use any valid name for the alias, as long as it does not clash with one already in use in the keystore. The fact that you do not have to import a server certificate obtained from a trusted certification authority in this way is a great advantage, of course, because it means that clients that want to connect to your service do not need to get a copy of your certificate in advance.

In order to run the client application with HTTPS, it is necessary to supply the correct URL and arrange for it to use the keystore that has just been created to look for certificates. To point the application at the correct keystore, the two system properties listed in Table 3-4 need to be set.

Table 3-4. System properties used by JSSE to access a keystore

Property	Description
javax.net.ssl.trustStore	The pathname of the keystore. In this case, this is *${EXAMPLES_ROOT}\client.keystore*.
javax.net.ssl.trustStorePassword	The password needed to access the keystore. By default, this password is changeit.

A target that runs the web service over HTTPS using basic authentication by setting the appropriate values for both of these properties has been included in the Ant buildfile for this example:

```
<target name="run-httpsserver-client" if="client.present"
      depends="init">
    <java classname="${CLIENT_CLASS_NAME}" fork="yes">
        <sysproperty key="javax.net.ssl.trustStore"
                     value="${EXAMPLES_ROOT}/client.keystore"/>
        <sysproperty key="javax.net.ssl.trustStorePassword"
                     value="changeit"/>
        <arg line="${CLIENT_HTTPS_SERVER_AUTH_ARGS}"/>
        <classpath refid="run.path"/>
    </java>
</target>
```

The property CLIENT_HTTPS_SERVER_AUTH_ARGS is set using properties in the *jwsnutExamples.properties* file to the appropriate URL for the service, which in this case is:

https://JWSUserName:JWSPassword@serverhost:8443/BookImageService/basicAuth

(or port 7000 if you are using the J2EE reference implementation).

To use this target, open a command window, make *chapter3\bookimageservice* (relative to the installation directory of the book's example source code) your working directory, and then type the command:

```
ant run-httpsserver-client
```

You should see the application start up and run as usual, although there will probably be a slight delay because of the additional overhead required to set up an HTTPS connection. If you'd like to see the details of the setup process, use the target `run-httpsserver-client-debug` instead.