Google, Amazon, and Beyond: Creating and Consuming Web Services

ALEXANDER NAKHIMOVSKY AND TOM MYERS

Apress™

Google, Amazon, and Beyond: Creating and Consuming Web Services
Copyright ©2004 by Alexander Nakhimovsky and Tom Myers

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-131-3

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewer: Jeff Barr

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wright, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Sofia Marchant

Copy Editor: Nancy Depper

Production Manager: Kari Brooks

Production Editor: Laura Cheu

Proofreader: Linda Seifert

Compositor: Diana Van Winkle, Van Winkle Design

Indexer: Kevin Broccoli

Artist: Joan Howard

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

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

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

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

CHAPTER 4

DBService and a Book Club

Now that we have created a few SOAP clients, we are going to create a SOAP server. The server will provide a database service, accept SQL queries wrapped in SOAP messages, and return query results (or SOAP faults). We will call it DBService.

Why would such a service be useful? We can think of at least four reasons, and others may well come up in later chapters.

- Interoperability: The database can provide its service to clients written in any language and running on any platform.

- Security: As some system administrators have found over the past few years, it is a bad idea to let your database be visible to the outside world. Our DBService will expose its database only to pre-approved queries from pre-approved places, that is, only to the extent that we're willing to risk.

- Local-cache: The DBService can be used as a local cache to reduce bandwidth use. Because both Google and Amazon restrict access to their services, a local cache can be very useful.

- Flexibility: DBService can easily combine data from other services with local data from approved database users.

For instance, a book club can use the DBService to combine Amazon data with locally produced book reviews and ratings. We present such a book club application in this chapter. Its client code is Javascript based on xmlhttp.js, and the server is written in Java, but even if you have not done any programming in Java you should be able to read and understand its code.

In this chapter, we will cover the following:

- The application demonstrated

- The main components of the application

- Server initialization from an XML configuration file
- Socket management
- HTTP and SOAP processing: sending and receiving messages
- Database management from Java: the JDBC APIs
- Returning the response to the client

We don't discuss the client code at all because it has relatively few new ideas.

The Book Club Application

The client for the book club application looks as shown in Figure 4-1.

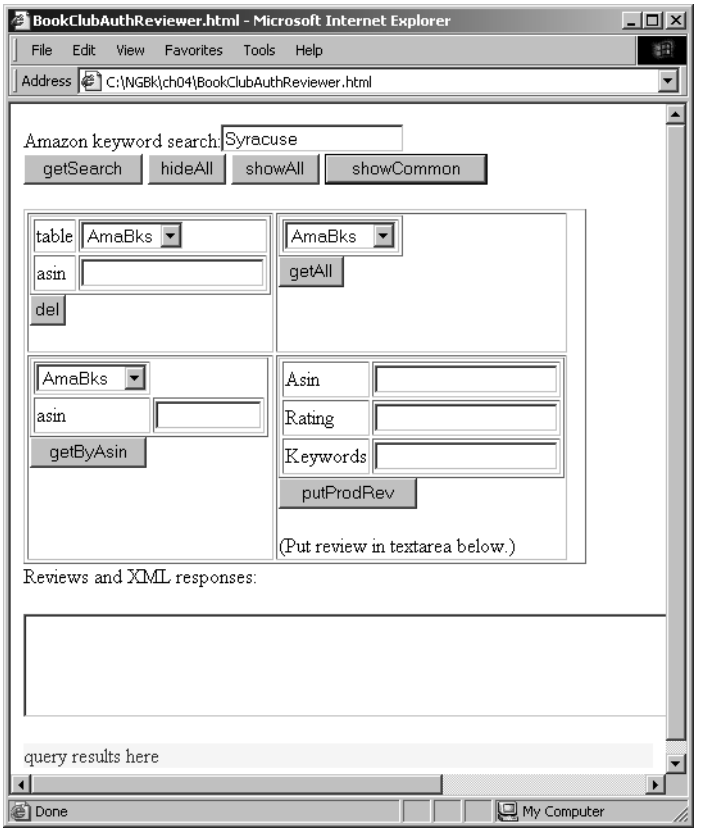

Figure 4-1. Book club application client

This client supports two main use cases. In one of them, a registered book club member adds a record to the book club database. In the other, a reader queries the database for information. The first use case proceeds as follows:

1. The user queries the Amazon database using keywords, as in the last application of Chapter 3.

2. The results of the query are displayed in a table similar to the one shown in Chapter 3 except that each row of the table has an additional cell with a button in it. Clicking the button copies the row's data to the Book Club database. (Only fields present in the Amazon table of the Book Club database are copied.)

3. The user adds local information to the book record copied from Amazon. Specifically, the user fills out the fields for User ID, Rating, Keywords, and a review. (The review is entered in a text area with simple formatting capabilities for a paragraph break, boldface, italics, and a hypertext link.) Clicking the Submit button sends the information to the Reviews table of the Book Club database.

To make sure the review is in the database, the user may want to go through the second use case and enter some query data into the search form and submit. The table of query results is displayed.

Although the two use cases are quite different, they are processed by the same code and follow the same path through software components and data formats. We will trace that path in the next section.

Main Components, in Order of Appearance

The application springs into action when the client generates a SOAP envelope and sends it as the body of an HTTP message using xmlhttp, exactly as in Chapter 3. The differences begin on the other end. In Chapter 3, when an HTTP message is sent to a specific port on a specific host, the receiver is a Web Server that listens to HTTP messages on that port. More precisely, at the byte level, the software object that listens to traffic on a given port is called a socket; the socket passes the raw bytes to the Web server to interpret as an HTTP message (shown in Figure 4-2).

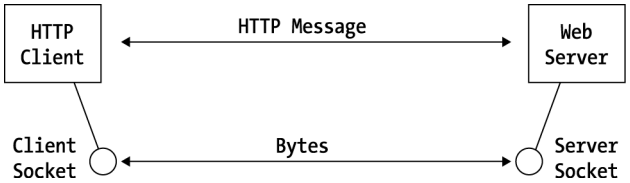

Figure 4-2. HTTP and sockets

The default port for HTTP communication is 80 but you can specify any port in the URL, as in `http://localhost:8080/index.jsp`. Even when a port other than 80 is used, the software listening to that port is usually a Web Server—but not in the Book Club application. In this application, we take control of the socket (which is easy to do in Java) and pass the raw bytes to our DBService directly (as shown in Figure 4-3) to interpret as a SOAP message inside an HTTP message.

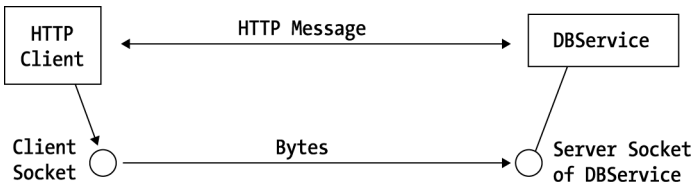

Figure 4-3. HTTP, sockets, and DBService

In other words, the DBService has an outer layer that functions as a very simple Web Server in that it parses the HTTP message. It separates its headers from the body and sends the body (which is a SOAP envelope) to the next layer, a SOAP service. The SOAP service parses the SOAP envelope, extracts the procedure call and parameters, and invokes the procedure. Then the action moves on to the database component of the service that handles the traffic between the SOAP service and the database. The service sends an SQL query and receives the query result, represented as a text string that is, in fact, an XHTML table element. The service integrates the query result into a SOAP message, integrates the SOAP message into an HTTP message, and sends it back to the client. This sequence of actions is shown in Figure 4-4.

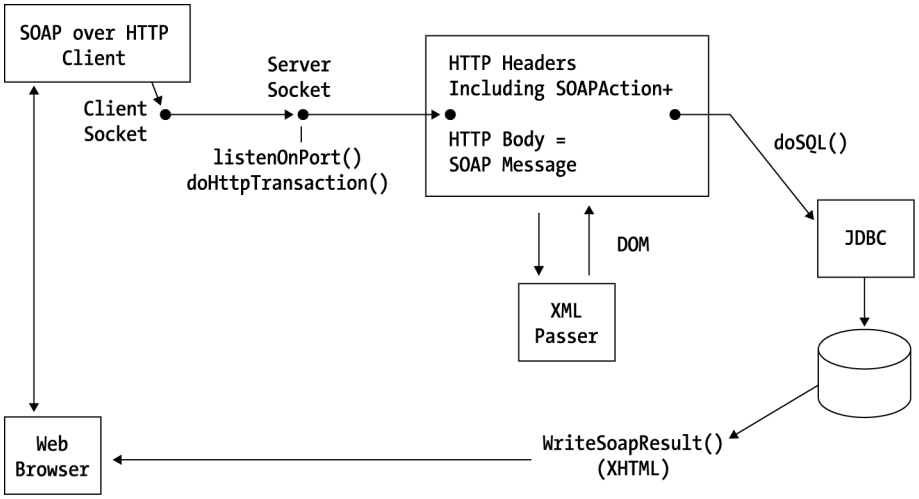

Figure 4-4. Main components and data transformations

Much of the rest of the chapter is a more detailed examination of the data path shown in Figure 4-4. Transitions along that path are executed by methods of the `DBService` class. Before data can begin to flow, we have to start the service by creating an instance of the class and telling it to listen for incoming SOAP messages. This is done by starting the application from the command line on the server.

Service Startup

Java command-line applications start in the `main()` method of the class whose name is the name of the application. If your application is called MyApp, there must be a class called `MyApp` whose object code is in the file `MyApp.class`, compiled from the source file `MyApp.java`. The class must have a `main()` method with a specific signature. (The signature of a method is the data type of its return value and the data type(s) of its argument(s)). In the case of `main()`, there is no return value and its single argument is an array of strings.

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

This says that the method is public and therefore visible from outside the class. It is static, which means you don't have to create an instance of the class in order to call the method. Its argument is an array of string objects that are command-line arguments to the application call. For instance, note the following example:

```
java MyApp abc 17 true
```

In this example, `args[0]` is "abc," `arg[1]` is "17," and `args[2]` is "true." If you run your application without any command arguments, the args array is empty and its length is 0.

In our case, the application is called DBService, and it can be invoked in one of the following three ways:

```
java DBService
java DBService someXmlConfigFile.xml
java DBService someXmlConfigFile.xml serviceArg1 serviceArg1 ...
```

If the application call has any arguments at all, the first argument (which becomes `args[0]`) must be the name of an XML configuration file that is used to create an instance of the `DBService` class. If there are no arguments, a default XML configuration file is used. If there is more than one argument, the arguments following the XML file are interpreted as arguments to a SOAP remote procedure call, and the first call to the service takes place immediately at startup.

The `main()` method of the DBService class is in shown in Listing 4-1.

Listing 4-1. The `main()` *Method of DBService*

```
public static void main(String[]args)throws Exception{
  String fileName="DBServiceConfig.xml"; // default XML configuration file
  if(args.length > 0) // an alternative configuration file is supplied
    fileName=args[0];
// create an instance of DBService using an XML decoder
  XMLDecoder xmlDecoder=
      new XMLDecoder(new FileInputStream(fileName));
  DBService dbService=(DBService)xmlDecoder.readObject();
// if there are additional arguments, call the service
  if(args.length > 1){
// copy command-line args to a new array
    ArrayList aL=new ArrayList();
    for(int i=1;i<args.length;i++)
      aL.add(args[i]);
// call doSQL method of the service, output result to the screen
```

```
    dbService.doSQL(aL,new PrintWriter(System.out,true));
  }
// start DBService listening to incoming SOAP requests
// on port specified in the XML configuration file
  dbService.listenOnPort();
} // end of main()
```

We will revisit this code after we work through the rest of the application, and many details will become more meaningful in the accumulated context. In the meantime, note the following:

- The method checks to see if there are any arguments; if so, an alternative XML config file is used.

- To create an instance of the class from an XML file, we use an XML decoder object, which is an instance of the XMLDecoder class. As of version 1.4, this class is part of the Standard Edition of the Java Development Kit, j2sdk1.4. We will inspect the XML encoding of Java objects in a separate section.

- If there are extra command-line arguments, they are copied to a new array and the service is invoked. To invoke the service, you call its `doSLQ()` method. The method takes two arguments: an array we create and a destination for the returned result. A later section of this chapter explains Java IO and streams.

Finally, the method tells the service to begin listening on a specific port. This is because our service is not invoked via a standard Web Server but through a direct socket-to-socket communication. In effect, our service *is* a web server. This is explained in the next section.

Sockets and Ports

As explained earlier, the byte-level connection between two nodes on the Internet connects objects called sockets. There are two kinds of sockets: client sockets and server sockets. Client sockets initiate a connection by contacting a server socket. A server socket is associated with a specific host and a specific port on that host (so that hosts can carry on more than one communication at a time). Client sockets associated with HTTP clients indicate the server socket they want to contact in their destination URIs, which contain the name of the host or its IP address and the port number, for example, `http://localhost:65432`.

Client sockets are represented in Java by `java.net.Socket` class objects. Server sockets are represented `java.net.ServerSocket` class objects. To create a server socket on port 65432, for example, you say the following:

```
serverSocket=new ServerSocket(65432);
```

To have our server socket listen for HTTP messages on its port, we start an infinite loop that calls the socket's `accept()` method (shown in Listing 4-2). That method blocks (is not executed) until a client socket is detected trying to connect. At that point, the method returns a socket object representing the client. We call the `badSocket()` method, which checks that the client is not one we don't want to talk to (currently the method does nothing and returns false). If the client is not bad, we call `doHttpTransaction()` and give it the client socket as an argument.

Listing 4-2. Server Socket Listening

```
protected void listenOnPort()throws Exception{
  if(serverSocket==null) // socket has not been created yet
    initSocket();
  Socket clientSocket=null;
  try{
    while(true){ // infinite loop, listening
      try{
        clientSocket=serverSocket.accept();
        if (badSocket(clientSocket)){ // security checks can go here
           closeSocket(clientSocket);
           continue;}
      }catch(Exception ex){
        System.out.println("DBService failed to accept on "+
                           getPortNumber()+": "+ex);
        System.exit(1);
      }
      doHttpTransaction(clientSocket);
      }
    }catch(IOException e){e.printStackTrace();}
}
```

To understand how `doHttpTransaction()`works, we need to make a detour to cover Java IO and its collection of streams.

Java IO and Streams

All Java IO is performed in a uniform way, by using APIs via objects called streams. Whether the data comes from a keyboard, a file, or an Internet connection, it arrives via an input stream that is associated with the data source. It works similarly for output. All input streams have a `read()` method that reads the next byte or the next character, depending on the stream. Specialized streams have methods like `readLine()`, which reads the next line of characters; `readDouble()`, which reads the next double value; or even `readObject()`, which reads the next (serialized) object from an object stream.

All Java streams are divided into four groups based on two characteristics: input vs. output streams, and byte vs. character streams. There are, correspondingly, four general stream classes.

- Byte stream: InputStream
- Byte stream: OutputStream
- Character stream: Reader
- Character stream: Writer

In addition to these general classes, there are a number of other streams with more specific capabilities. For instance, to read data from files, you use `FileInputStream` or `FileReader`; to read data in chunks that are accumulated in a buffer (for improved performance, among other reasons) you use `BufferedReader or BufferedInputStream`, and so on. To create a buffered reader that can read characters line by line from the text file `textData.txt`, you proceed as follows:

```
FileReader fReader = new FileReader("textData.txt");
BufferedReader bfReader = new BufferedReader(fReader);
```

You can combine these into a single line of code.

```
BufferedReader bfReader = new BufferedReader(new FileReader("textData.txt"));
```

With a buffered reader in place, you can read the first line of the file.

```
String firstLine = bfReader.readLine();
```

Bytes and Characters

The distinction between byte and character streams deserves an explanation. Java characters are Unicode. Each Unicode character is a 16-bit number called a "code point." These 16-bit numbers can be represented in different ways called Unicode Transfer Formats or UTF for short. Two most common formats are UTF-16, in which each code point is represented by two bytes, and UTF-8, which is a variable-length format—the most frequently used characters are represented by a single byte, whereas less frequently used characters are represented by two or three bytes. The most frequently used characters are, of course, those of the ASCII table; UTF-8 represents them by a single byte and is backward compatible with ASCII.

In addition to Unicode characters, Java character streams support (more or less successfully) a variety of single byte character encodings such as ISO or ANSI standards, and proprietary (but widely used) formats from IBM, Microsoft and Apple. (These are sometimes called code pages or CP for short, such as the CP-1251 for Windows Cyrillic characters.)

Suppose you have a byte input stream `myInStream` that is carrying character data. In order to read data from the stream as characters rather than bytes, you have to convert that byte input stream into a character stream, as follows:

```
Reader myReader = new InputStreamReader(myInStream, "utf-8");
```

This tells the program to read the bytes from myInStream as characters encoded in UTF-8 standard.

Socket Communications

Back to sockets. The Socket class hides all the complexities of Internet communication into two methods, `getInputStream()` and `getOutputStream()`. The methods return byte streams attached to client socket, and all subsequent input operations use the standard stream APIs so neither the program nor the programmer has to remember where a particular stream of data is coming from. We would, however, know that it is an HTTP connection delivering and expecting character data in UTF-8 encoding (a reasonable choice for Web interactions). So our `doHttpTransaction()` starts by creating appropriate character streams (shown in Listing 4-3).

Listing 4-3. `doHTTPTransation()` *Part 1: Create Streams for Sockets*

```
public void doHttpTransaction(Socket clientSocket)
    throws Exception{
  OutputStream os=null;
  BufferedOutputStream bos=null;
  PrintWriter pW=null;
  try{
    InputStream is=clientSocket.getInputStream();
    BufferedInputStream bis = new BufferedInputStream(is);
    InputStreamReader reader=new InputStreamReader(bis,"utf-8");
    os=clientSocket.getOutputStream();
    bos = new BufferedOutputStream(os);
    pW=new PrintWriter(new OutputStreamWriter(bos,"utf-8"),true);
```

With a buffered `InputStreamReader` and a `PrintWriter` in place, we can read in the HTTP request, process it, and send the response back to the client socket. We encapsulate these tasks into two methods, `readHttpData()` and `doPost()`, shown in Listing 4-4.

Listing 4-4. `doHTTPTransation()` *Part 2: Use Streams to Do HTTP*

```
// read HTTP data into a hashtable
    Hashtable httpData=readHttpData(reader);
    if(!authorized(httpData)) // a hook to do authorization
      throw new Exception("authorization failure:\n"+httpData);
    String cmd=(String)httpData.get("METHOD");
    if("POST".equals(cmd))
// the rest of the processing done here
      doPost(httpData, pW);
    else // other commands are used in the next chapter's version
      throw new Exception("unknown command ["+cmd+
                          "]; only POST supported in this version");
    reader.close(); bis.close();  is.close();
    pW.close(); pW=null; bos.close(); os.close();
  }catch(Exception ex){
    if(pW!=null){
      sendExceptionFault(ex,pW);
      pW.close();bos.close();os.close();
    }
  }
} // end of doHttpTransaction
```

If something goes wrong with processing HTTP request, we call `sendExceptionFault()` to construct and send back a SOAP fault message. It follows the familiar pattern: the SOAP message is constructed as a string and sent back to the client, after the command line and the headers of the HTTP response message. In Listing 4-5, we skip a few familiar lines of the SOAP header.

Listing 4-5. SOAP Fault Message

```
protected void sendExceptionFault(Exception ex,PrintWriter pW){
StringBuffer sB=new StringBuffer();
sB.append("<?xml version='1.0' encoding='UTF-8'?>\n")
  .append("<SOAP-ENV:Envelope \n")
...
  .append(" <SOAP-ENV:Body>\n")
  .append("   <SOAP-ENV:Fault>\n")
  .append("     <SOAP-ENV:faultcode>")
  .append(42).append("</SOAP-ENV:faultcode>\n")
  .append("     <SOAP-ENV:faultstring>internal error</SOAP-ENV:faultstring>\n")
  .append("     <SOAP-ENV:detail>\n").append(ex.toString())
  .append("</SOAP-ENV:detail>\n")
  .append("   </SOAP-ENV:Fault>\n")
  .append(" </SOAP-ENV:Body>\n")
  .append("</SOAP-ENV:Envelope>\n");
  String msg=sB.toString();
  pW.print("HTTP/1.0 500 Server Error: Malformed HTTP Request \r\n");
  pW.print("Content-Type: text/xml; charset=utf-8\r\n");
  pW.print("Content-Length: "+msg.length()+"\r\n\r\n");
  pW.print(msg);
  pW.flush();
}
```

At this point we are done with socket communications. The action moves to processing HTTP data, extracting the SOAP message from it, and acting upon it.

Processing HTTP Request

As you recall, an HTTP request has the following structure:

- Line 1: *command uri http-version*
- Several non-blank lines: *header-name* : *header-value*
- A blank line consisting of two characters, CarriageReturn and LineFeed, ASCII 10 and 13, denoted \r and \n in Java code
- The body of the message, possibly empty, but in our case containing the SOAP payload

The `readHttpData()` method processes an HTTP request and stores its data in a Java hashtable object for easy retrieval. Its operation is fairly transparent, but there is one little quirk: instead of the `readLine()` method of Java-buffered character streams, we use our own `readLine()`, shown in Listing 4-6. The reason is that Java's built-in `readLine()` recognizes system dependent end-of-line markers that are different in different operating systems, whereas we want to recognize the HTTP-specific `\r\n` sequence.

Listing 4-6. `readLine()` *for HTTP Data*

```
public String readLine(Reader reader) throws Exception{
  StringBuffer sB=new StringBuffer();
  int ch;
  for(ch=reader.read();ch>=0 && ch!='\r';ch=reader.read())
    sB.append((char)ch);
  if(ch<0) // read() returns -1 on end of file
    return sB.toString();
  ch=reader.read();
  if(ch!='\n')
    throw new Exception("line ["+sB.toString()+"] lacks \\n");
  return sB.toString();
}
```

The readHttpData() method parses the first line and then parses and stores header lines until it finds the blank line separating the headers from the body. The body is stored as PAYLOAD. In this method, we don't use any headers except Content-Length to control the loop that reads in the body of the message. To make sure ContentLength has a value, we initialize it to 0, as shown in Listing 4-7.

Listing 4-7. Read HTTP Request from Stream, Store in Hashtable

```
public Hashtable readHttpData(Reader reader)throws Exception{
  ArrayList httpHeaderList=new ArrayList();
  Hashtable hashtable=new Hashtable();
  hashtable.put("Content-Length","0"); // default
// read and process the first line (command line)
  String cmdLine=readLine(reader);
  int firstBlank=cmdLine.indexOf(' ');
  int lastBlank=cmdLine.lastIndexOf(' ');
  if(firstBlank < 0 || firstBlank == lastBlank)
    throw new Exception("Invalid HTTP method line ["+cmdLine+"]");
  hashtable.put("METHOD",cmdLine.substring(0,firstBlank));
  hashtable.put("URL",cmdLine.substring(1+firstBlank,lastBlank));
  hashtable.put("HTTP",cmdLine.substring(1+lastBlank)); // HTTP version
// read headers, store in hashtable
  String hdr=readLine(reader);
  while(hdr.length()>0){
    String[]nameVal=hdr.split(": ",2); // split at colon, two items at most
    if(nameVal.length > 1)hashtable.put(nameVal[0],nameVal[1]);
    hdr=readLine(reader);
  }
// get Content-Length from the Hashtable
  int len=Integer.parseInt((String)hashtable.get("Content-Length"));
// read and store HTTP body in Hashtable
  String httpBody = readUpToLength(reader,len);
  hashtable.put("PAYLOAD", httpBody);
  return hashtable;
}
```

With the HTTP request parsed and packaged and an output character stream attached to the client socket, we can leave the world of HTTP behind. The hashtable and the PrintWriter are all we need for SOAP messaging and database access. They are the two arguments that are passed to doPost() (shown in Listing 4-8). doPost(), in turn, passes them on to doDBServiceCall() after checking that the HTTP message has a soapAction header.

Listing 4-8. doPost()

```
public void doPost(Hashtable httpData,PrintWriter pW)throws Exception{
  String soapAction=(String)httpData.get("SOAPAction");
  if("DBServerCall".equals(soapAction))
    doDBServiceCall(httpData,pW);
  else
    throw new Exception("POST with unknown SOAPAction:["+soapAction+"]");
}
```

doDBServiceCall() does all the remaining work.

Parse SOAP, Return Query Result

The code of doDBServiceCall() can serve as an outline for the rest of the chapter.

- Extract the SOAP message from the HTTP message
- Parse the SOAP message as an XML document
- Extract the parameters of the SOAP call into an array
- Run the database query
- Write the results of the query to output

Except for the database query, we will cover each of these items in the current section, starting with doDBServiceCall() itself (shown in Listing 4-9).

Listing 4-9. doDBServiceCall()

```
public void doDBServiceCall(Hashtable httpData,PrintWriter pW)
    throws Exception{
// retrieve the HTTP body ("PAYLOAD") from the hashtable
  String docString=(String) httpData.get("PAYLOAD");
// parse SOAP message as XML, get DOM Document
  Document doc=readDocument(docString);
// extract parameters of SOAP call into an array
  ArrayList arrayList=getSOAPParams(doc);
// do database query, return result as an XML String
  String soapRes=doSQL(arrayList);
// Integrate the SOAP result as SOAP body into
  writeSOAPResult(soapRes,pW);
} // end doPost()
```

As you can see, each item is encapsulated into a separate procedure. We will now discuss each item in the order of its appearance, one subsection per procedure.

XML Parsing in Java

You saw XML parsing in Mozilla Javascript and IE JScript in Chapter 2, Listing 2-8. We repeat it here for comparison with the Java version.

Listing 2-8. Parse an XML String into a DOM Object

```
function parseXML(str){
  var doc=null;
  if(inIE){ // IE version
    doc=new ActiveXObject("Microsoft.XMLDOM");
    doc.loadXML(str); // does the parsing
  }
  else { // Mozilla/Netscape
    var domParser=new DOMParser();
    doc=domParser.parseFromString(str, "text/xml");
  }
  return doc; // .documentElement;
}
```

As we will explain in Chapter 8, the parsing process usually consists of two steps: obtain a parser object (which may or may not be called parser) and call its parse method (which may or may not be called parse). In Java, there is one more preliminary step: you obtain a "parser factory" that can be configured to produce a parser and set its properties. When you need a parser, you ask the parser factory for it. This way, you can have more than one parser available and your code is the same no matter which parser you use. Only the system property that specifies the parser factory needs to be changed to switch from one parser to another. The parser, incidentally, is called `DocumentBuilder` and the parser factory is `DocumentBuilderFactory`, both in the javax.xml package.

In the `DBService` class, `DocumentBuilderFactory` is a class variable, declared at the top level:

```
DocumentBuilderFactory dbf;
```

We obtain an instance of `DocumentBuilderFactory` in the `getDBF()` method, using the following line of code:

```
dbf=DocumentBuilderFactory.newInstance();
```

With a parser factory in place, the `readDocument()` method can obtain a parser object and parse (shown in Listing 4-10). An additional Java complication is that the argument to the `parse()` method has to be an object of class `InputSource`. To obtain such an object, we first convert our XML string to a character stream (StringReader), and give that stream as an argument to the InputSource constructor.

Listing 4-10. Parse an XML string into a DOM object using DocumentBuilder

```
public Document readDocument(String str)throws Exception{
  try{
    DocumentBuilder db=getDBF().newDocumentBuilder();
    InputSource is=new InputSource(new StringReader(str));
    Document doc=db.parse();
    return doc;
  }catch(Throwable ex){
    System.out.println("readDocument failure:"+ex);
    return null;
  }
}
```

Now that the SOAP message is parsed into a DOM object, we can take it apart and extract what we need from it.

SOAP Parameters

To extract the parameters of our SOAP call, we pass the DOM object to `getSOAPParams()`, which uses the DOM `getElementsByTagName()` method to extract all `<dbParam>` elements into a NodeList object. That object has a `getLength()` method to find out how many items it contains and an `item()` method to yield an item at the specified index. The text content of an item is not stored in the item directly but in its child Text node. We go through a loop picking out each item in turn until we get to its first (and only) child to obtain the SOAP parameter (shown in Listing 4-11).

Listing 4-11. Extract SOAP Parameters to an Array of String

```
public ArrayList getSOAPParams(Document doc){
  NodeList nodeList=doc.getElementsByTagName("dbParam");
  ArrayList arrayList=new ArrayList();
  for(int i=0;i<nodeList.getLength();i++){
   Node child=nodeList.item(i).getFirstChild();
    String param="";
    if(child!=null)
      child.getNodeValue();
    arrayList.add(param);
  }
  return arrayList;
}
```

The parameters of the SOAP call now travel to the database to run the query, and the result of the query is returned as a string. The string becomes part of the output produced by `writeSOAPResult()`.

Output the Result of SOAP Call

This is largely familiar code: construct the SOAP envelope, construct the HTTP message, and send (shown in Listing 4-12).

Listing 4-12. Output the Result of SOAP Call

```
public void writeSOAPResult(String soapRes,PrintWriter pW)
    throws Exception{
// construct SOAP envelope that contains soapRes
  StringBuffer sB=new StringB uffer();
  sB.append("<?xml version='1.0' encoding='UTF-8'?>\n")
    .append("<SOAP-ENV:Envelope \n")
// SOAP-ENV attributes are appended here; start SOAP body
    .append(" <SOAP-ENV:Body>\n")
// append doSQLResponse element that contains soapRes
    .append("  <doSQLResponse>\n")
    .append(soapRes)  // result of SOAP call goes here
    .append("  </doSQLResponse>\n")
// append closing tags
    .append(" </SOAP-ENV:Body>\n")
    .append("</SOAP-ENV:Envelope>\n");
  soapRes=sB.toString();
```

```
// output HTTP command line and headers
  pW.print("HTTP 1.0 200 OK\r\n");
  pW.print("Content-Type: text/xml; charset=utf-8\r\n");
  pW.print("Content-Length: "+soapRes.length()+"\r\n");
  pW.print("Date: "+rfc1123DateFormat.format(new java.util.Date())+"\r\n");
   //  pW.print("Date: Sun, 10 Feb 2002 22:19:37 GMT\r\n");
// output blank line and HTTP body, i.e., SOAP message
  pW.print("Server: DBService 0.11\r\n\r\n");
  pW.print(soapRes);
  pW.flush();
}
```

This completes our travels around the middle layer of the application. It is now time to penetrate into the database core where data is stored and retrieved from the database. The interaction between the middle layer and the core is encapsulated into the `doSQL()` method. Before we can tackle its code, we need to go over the general framework of interaction between a Java program and a database.

Driver, Database, Connection, and Statement

Java libraries for database access are collectively known as JDBC, Java Database Connectivity. JDBC makes it possible to connect to a database from Java code, run SQL queries and process query results. This activity is encapsulated in such classes and interfaces as Connection, Statement, and ResultSet, among others. They are all found in the `java.sql` package, part of the JDK Standard Edition.

JDBC Driver

As you know, different database vendors may implement different functionality and support slightly different flavors of SQL. JDBC aims to be vendor-independent. The key to vendor independence is a **JDBC driver**, a software package produced by database or third-party vendors that encapsulate DBMS-specific features. (For instance, it switches between DBMS-specific data types and JDBC data types.) A list of over 150 JDBC drivers is available at `http://industry.java.sun.com/products/jdbc/drivers`; some of them are free, others are commercial products. In addition, Sun provides a JDBC-ODBC bridge as part of the standard Java distribution. This bridge makes it possible to connect to ODBC data sources using JDBC APIs. This is the driver you use to connect to an Access database or an Excel spreadsheet.

In our code, we store the name of the driver in the `dbDriver` variable. This variable is initially set to sun.jdbc.odbc.JdbcOdbcDriver, but this assignment is overridden by the XML configuration file. In that file, we specify a driver for the MySQLdatabase, which is what we use in the book.

JDBC drivers are usually distributed as .jar (Java archive) files. To install it, simply put the .jar file on your Java classpath. To use it, include a line like this in your code:

```
Class.forName(your--driver-name);
```

Because the name of the driver is stored in the `dbDriver` variable and the `getDbDriver()` method returns the value of that variable, in our code this line comes out (in our `getDBConnection()` method) as

```
Class.forName(getDbDriver());
```

Once the driver is instantiated, you can use JDBC code. A typical database access proceeds as follows:

1. Obtain a connection, typically from a connection pool.

2, Create a statement object to execute SQL statements.

3. Run a database session, execute statements, process result sets, and so on.

4. Close connection (or return it to the connection pool).

The following sections explain these steps and illustrate them with examples from our code. The first step is to obtain a connection to your database.

Connections and Connection Pooling

The simplest way to obtain a connection is via a public static `getConnection()` method of the `java.sql.DriverManager` class. The method takes three parameters: the database URI, the username, and the password. The format of the database URI is specific to the JDBC driver in use. In the case of the JDBC-ODBC bridge, you must create a DSN (Data Source Name) using the ODBC manager and use that name as the database URI. Assuming that the DSN is “wsbkdb” and the user name and password are empty strings, you would obtain a connection by the following line of code:

```
Connection con=DriverManager.getConnection("jdbc:odbc:wsbkdb","","");
```

For the MySQL database and its driver, you specify the database URI like this (quoted from the XML configuration file):

```
jdbc:mysql://localhost:3036/wsbkdb
```

Here localhost is the host name of the computer on which the database is running, and wsbkdb is the name of the MySQL database. MySQL DBMS runs on port 3036 by default, so we could remove :3036 from the URL and it would make no difference.

Just as with the driver name, we store the database URI, the username, and the password in Java variables that we initialize from XML. Each variable has a `get()` and a `set()` public methods, as shown in Listing 4-13.

Listing 4-13. Variables for Database Connection

```
protected String dbURL="jdbc:odbc:wsbkdb";
  public void setDbURL(String S){dbURL=S;}
  public String getDbURL(){return dbURL;}
protected String dbUser="";
  public void setDbUser(String S){dbUser=S;}
  public String getDbUser(){return dbUser;}
protected String dbPwd="";
  public void setDbPwd(String S){dbPwd=S;}
  public String getDbPwd(){return dbPwd;}
```

With variables and access methods in place, we obtain a connection by the expression

```
  DriverManager.getConnection(getDbURL(),getDbUser(),getDbPwd());
```

This expression is in our own `getDBConnection()` method. This method can be changed to use a more sophisticated method of managing connections called "connection pooling." Obtaining a connection is a computationally expensive operation that should be done as rarely as possible. With connection pooling, the application obtains a pool of connections in one action, typically at startup, and asks for another connection to be added to the pool only if the pool dries up. When a user asks for a connection, it is allocated from the pool, and when a user releases a connection, it is returned to the pool. In our code, releasing a connection is also encapsulated as a method, `freeConnection()`, that currently does nothing but can be rewritten to use connection pooling.

There are several connection pool implementations available, and the latest version of JDBC implements a standard API for managing a connection pool. To

incorporate connection pooling in our application, you would only have to install the appropriate software and rewrite our `getConnection()` and `freeConnection()` methods.

SQL Statements and Result Sets

Once you have a connection, you can obtain a statement object and run SQL statements. A typical sequence is shown in Listing 4-14.

Listing 4-14. Statement and Query String

```
      Statement stmt = con.createStatement();
// SQL to insert an author into record specified by Asin ID number
      String sqlStr=
"INSERT INTO AmaAuth (Asin,Author) VALUES (7346,'Jay Jones')";
      int numberUpdates=stmt.executeUpdate(sqlStr);
```

Note that character strings (but not integers) have to be quoted in the query string, so we use single quotes within double quotes. The returned integer, in this case 1, indicates the number of rows affected by the update. (If you were to delete 3 rows, the method would return 3.)

To run a SELECT query, you would use `executeQuery()` rather than `executeUpdate()`, and the returned value would be a ResultSet object that provides sequential access to the returned records. ResultSet has a `next()` method that returns the next record or null if you have reached the end of the record set. Within each record, individual fields are retrieved by `getXX()` methods, where XX stands for a data type: `getInt()`, `getString()`, and so on. The argument for all these methods is either an integer giving the number of the field in the record or the field's name in the database table.

In many situations, a better alternative to Statement is the JDBC PreparedStatement. PreparedStatement has an SQL query imprinted on it at construction. It is more efficient than plain Statement because its SQL query is compiled once and can be reused many times with different parameters.

Consider a simple example. You have a database table that includes names and e-mail addresses and you want to retrieve the addresses by name. You created a database connection as described in the preceding section and you are ready to create a query string and a statement object. Assume that the name to search by is in the `currentName` variable. With plain Statement, you would create a query string like this

```
"SELECT addr FROM addrBook WHERE name='" + currentName + "'"
```

Remember, you need single quotes within double quotes so the value of `currentName` comes out quoted in the resulting string. This is error-prone and may result in nasty complications: what if the name is "O'Donnell"? In addition, you have to remember to quote strings but not integers or dates unless you insert dates as strings. The PreparedStatement, is not only more efficient, it provides a simple and uniform way of filling in arbitrary parameters.

First, you create a query string with question marks as placeholders for parameters to be filled in, and you use the query string in creating your PreparedStatement.

```
String queryStr = "SELECT addr FROM addrBook WHERE name=?";
PreparedStatement prepStmt = conn.prepareStatement(queryStr);
```

Next, you fill in the value of the parameter using one of many datatype-specific procedures that are provided for that purpose: `setString()`, `setInt()`, `setBlob()`, `setBoolean()`, `setDate()`, or even `setObject()` in which we provide an arbitrary object and tell the database what standard SQL type to convert it to. In this case, we need `setString()`.

```
// set the value of the first parameter of PreparedStatement to currentName
prepStmt.setString(1,currentName);
// run the query
ResultSet rs= prepStmt.executeQuery();
```

Note that `executeQuery()` does not take any arguments because the query string is already imprinted on the PreparedStatement and its parameter is already set. For UPDATE queries, you would again use `executeUpdate()` rather than `executeQuery()`. If you don't know what kind of query to expect, you can use the general `execute()` method that can execute both SELECT and UPDATE queries. It returns a Boolean value—true for SELECT queries and false for UPDATE ones. To extract the update count or the result set, use `getUpdateCount()` or `getResultSet()`. Let's work through an example from our code to see how it all fits together. In the course of the example, we will also take a first look at the XML encoding of Java code.

Prepared Statements and Our Method to Query Data

Because each PreparedStatement is associated with a specific query, if you name a PreparedStatement, you in effect name a query. If the user can access the database only by selecting from a list of query names, you have restricted the database access to pre-approved queries that can be specified in the XML configuration file.

We develop this idea a little further by creating a hashtable of DBQueryData objects, where each object contains the following:

- The ID of a query
- The SQL text of the query
- The array of data types of the parameters of the query

For instance, recall the query of Listing 4-14:

```
"INSERT INTO AmaAuth (Asin,Author) VALUES (7346,'Jay Jones')"
```

The variables of the DBQueryData object for that query would be as shown in Listing 4-15.

Listing 4-15. Variables of a DBQueryData Object

```
qID: INS_AmaAuth (simply because that's what we decided to call it)
qStr: "INSERT INTO AmaAuth (Asin,Author) VALUES (?,?)"
qTypes: {INT, TEXT}
```

We create a hashtable of DBQueryData objects from an array of such objects in the method `setQueryHashtable()`, shown in Listing 4-16.

Listing 4-16. Java Code to Create a Hashtable of DBQueryData Objects

```
public void setQueryHashtable(ArrayList SS){
   queryStrings=SS;
   queryHashtable=new Hashtable();
   if(SS==null) return;
   for(int i=0;i<queryStrings.size();i++){
     DBQueryData dbqd=(DBQueryData)(SS.get(i));
      String keyString=dbqd.getQID().toUpperCase();
     queryHashtable.put(keyString, dbqd);
   }
}
```

This is pretty straightforward, but there are two puzzling things about this method. First, where do we get an array of DBQueryData objects for our queries? Second, where is it called? If you inspect the text of DBService.java, you will see that `setQueryHashtable()` is defined but not invoked in that file. Both of these puzzling questions are answered if we look in the code of the XML configuration file, DBServiceConfig.xml.

XML Encoding of Java Code

To construct the DBQueryData object of Listing 4-15, we could use the Java code of Listing 4-17.

Listing 4-17. Java Code to Create a DBQueryData Object

```
DBQueryData dbqd=new DBQueryData();
dbqd.setQID("INS_AmaAuth");
dbqd.setQStr("INSERT INTO AmaAuth (Asin,Author) VALUES (?,?)");
dbqd.setQTypes('INT,TEXT');
```

Note that this code uses a default no-argument constructor to create the empty shell of an object and a sequence of `setXX()` methods to set the object's properties. This is precisely the kind of code that XML encoding is good at. Instead of placing that code into the Java file, we include the text of Listing 4-18 into the XML configuration file.

Listing 4-18. XML Encoding to Create a DBQueryData Object

```
<object class="soapUtil.DBQueryData">
   <void method="setQID"><string>INS_AmaAuth</string></void>
   <void method="setQStr">
       <string>
         INSERT INTO AmaAuth (Asin,Author) VALUES (?,?)
       </string>
   </void>
   <void method="setQTypes"><string>INT,TEXT</string></void>
</object>
```

Note three simple conventions.

- To create an object of class `MyClass`, you include an XML element `<object class="MyClass">`. This invokes the default constructor of `MyClass`.
- To invoke a method `myMethod()` that returns void, you include an XML element `<void method="myMethod">`.
- The XML elements representing arguments to a method are contained in the element representing the method itself.

The code in Listing 4-18 creates one `DBQueryData` object for a single query. The `setQueryHashtable()` method of Listing 4-16 needs an array of such objects. We use a dynamic array class called `ArrayList` in Java; it has an `add()` method that adds elements to the array and returns void. The XML code to construct an array of `DBQueryData` objects has the following structure (shown in Listing 4-19).

Listing 4-19. XML Encoding for an Array of `DBQueryData` *Objects*

```
<object class="java.util.ArrayList">
   <void method="add">
      <object class="soapUtil.DBQueryData">
         <!-- code to create a single DBQueryData object goes here -->
       </object>
     </void>
    <!-- other <void method = "add"> elements go here, one for each query -->
</object> <!-- end of <object class="java.util.ArrayList"> element -->
```

The XML code that invokes `setQueryHashtable()` has the following structure (shown in Listing 4-20).

Listing 4-20. XML Encoding for an Array of `DBQueryData` *Objects*

```
<void method="setQueryHashtable">
  <!-- code of the preceding listing,
       representing the single argument of the method,
       goes here -->
</void>       <!-- end of void method="setQueryHashtable" element -->
```

The entire DBServiceConfig.xml file invokes the default constructor for DBService and sets the properties of the created DBService object. The overall structure of the file is best seen by looking at that file in the browser with all second-tier elements collapsed, as shown in Figure 4-5.

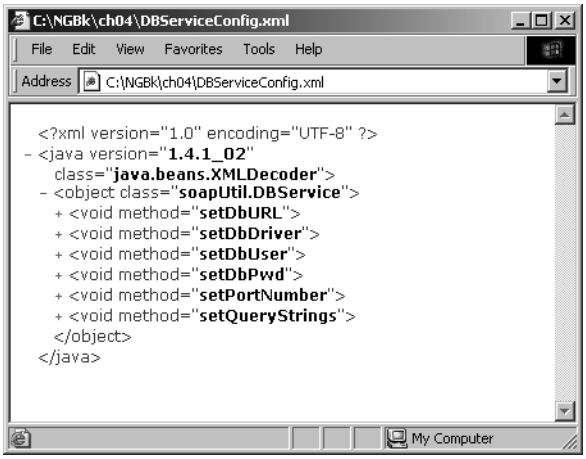

Figure 4-5. XML configuration file in the browser

From this screenshot, you can see the variables of the `DBService` class. XML encoding is described in `http://java.sun.com/products/jfc/tsc/articles/persistence3`, and we will see more of it in the next chapter. For now, you can usefully compare the sample of Listing 4-18 to the Java code of Listing 4-17. We are ready to look at the code of `doSql()` and supporting methods.

Database Access in DBService

The database access code in this chapter contains hooks for further improvements such as connection pooling or different access levels for different categories of users. Some of these improvements will be developed in Chapter 5; in this chapter we'll just get the basic functionality going.

We last saw `doSql()` method in Listing 4-9 where it was invoked from `doDBServiceCall()`, in the following lines of code:

```
// extract parameters of SOAP call into an array
  ArrayList arrayList=getSOAPParams(doc);
// do database query, return result as an XML String
  String soapRes=doSQL(arrayList);
```

We can now explain what parameters the SOAP client sends to the SOAP server. The first parameter is the name of the `DBQueryData` object for the PreparedStatement to execute, and the rest are the parameter values to insert into that statement. Our plan is to use the name of the `DBQueryData` object to retrieve the object itself. From the object, we retrieve the query string and the data types array. We use these two items of information together with the parameter values supplied by the SOAP call to construct the PreparedStatement object, fill in its parameters, and run the query (shown in Listing 4-21).

Listing 4-21. `doSql()`

```
protected void doSQL(ArrayList nameParams, PrintWriter out)
    throws Exception{
  PreparedStatement pS=null;
  String qName=((String)nameParams.get(0)).toUpperCase();
  DBQueryData dbqd=(DBQueryData) queryHashtable.get(qName);
  if(dbqd==null)
    throw new Exception("ERR NO SUCH QUERY AS "+qName+"");
  String[]qTypes=dbqd.getQTypeArray();
  if(qTypes.length+1 !=nameParams.size())
    throw new Exception("Query ["+qName+"] expects "+qTypes.length+
                        " params, not "+(nameParams.size()-1));
// if we get here, we are ready to run the query
  Connection con = getDBConnection();
  try{
    pS=con.prepareStatement(dbqd.getQStr());
    for(int i=0;i<qTypes.length;i++)
      setParamStr(pS,i+1,(String)nameParams.get(i+1),qTypes[i]);
    if(pS.execute()) writeResultSet(pS.getResultSet(),out);
    else writeResultCount(pS.getUpdateCount(),out);
  } catch(Exception ex){
    throw new Exception("ERR doSQL("+qName+","+ex+")\n");
  }
  finally{
     if(pS!=null)try{pS.close();}catch(Exception ex){}
     pS=null;
    freeDBConnection(con);
  }
}
```

As you can see, setting the parameters is encapsulated into the `setParamStr()` method. The execution results are in a call on `writeResultSet()` (if a ResultSet has been returned) or on `writeResultCount()` (if the returned value is an integer

showing the number of updated records). We will take these up next, in the remainder of this section.

The `setParamStr()` method sets the value of a query parameter in a PreparedStatement. As we mentioned, PreparedStatement has a number of `setXX()` methods for that purpose, each corresponding to a data type or a group of related data types. `setParamStr()` consists of a conditional with many branches that examine the value of the data type argument and invokes the corresponding `setXX()` method. Listing 4-22 shows the first three of the branches.

Listing 4-22. Partial Listing of `setParamStr()`

```
public void setParamStr(
    PreparedStatement pStmnt,
    int i,
    String pVal,
    String pType)  throws Exception{
  String t=pType;
  try{
// conditions for using setString()
    if(t==null || t.length()==0 ||"text".equalsIgnoreCase(t)
     ||"varchar".equalsIgnoreCase(t) || "string".equalsIgnoreCase(t))
      pStmnt.setString(i,pVal);
// conditions for using setObject()// allow for MS Access limitations
    else if("longtext".equalsIgnoreCase(t)||"longvarchar".equalsIgnoreCase(t))
      pStmnt.setObject(i,pVal,java.sql.Types.LONGVARCHAR);
// conditions for using setDate()
    else if(t.equalsIgnoreCase("date")){
      java.util.Date d=null;
      try{d=simpleDateFormat.parse(pVal);}
      catch(Exception ex){d=rfc1123DateFormat.parse(pVal);}
      java.sql.Date dbdate=new java.sql.Date(d.getTime());
      pStmnt.setDate(i,dbdate);
    }
// several more branches
    else pStmnt.setString(i,pVal);
  }catch(java.text.ParseException e){
     throw new
      SQLException("setParamStr failed to parse ["+pVal+"] as ["+t+":"+e);
  }catch(java.lang.NumberFormatException e){
     throw new
      SQLException("setParamStr failed to parse ["+pVal+"] as ["+t+":"+e);
  }catch(Exception e){
     throw new
```

```
      SQLException("setParamStr failed to set param "+i+
                                    ", ["+pVal+"] as ["+t+":"+e);
  }
}
```

Once the query parameters are set, we can run the query with the `execute()` method of PreparedStatement. As we mentioned, this method returns a Boolean value indicating whether the query returns a ResultSet (for SELECT queries) or an integer (for UPDATE queries and data definition operations such as CREATE TABLE). For an integer, we have only to say the following (in `writeResultCount()`):

```
  out.println("<span class='updateCount'>"+count+"</span>\n");
```

For a ResultSet, however, we output a `<table>` (shown in Listing 4-23). In order to output a table, we need to know how many columns it will have, which is the number of fields requested by the query. We obtain this information from the ResultSetMetaData object associated with the ResultSet shown in Listing 4-23.

Listing 4-23. The DBService `writeResultSet()` *Method*

```
protected void writeResultSet(ResultSet res,PrintWriter out)throws Exception {
  try{
    out.println("<table border='1'>");
    ResultSetMetaData rsmd=res.getMetaData();
    int colCount=rsmd.getColumnCount();
    out.println("<tr>");
    for(int i=0;i<colCount;i++)
    out.println("<th>"+rsmd.getColumnName(i+1)+"</th>");
    out.println("</tr>\n");
    while(res.next()){
      out.println("<tr>");
      for(int i=0;i<colCount;i++)
        out.println("<td>"+xmlEncode(res.getString(i+1))+"</td>");
      out.println("</tr>\n");
      }
    out.println("</table>\n");
  }finally {
    if(res!=null)try{res.close();}catch(Exception ex){}
  }
}
```

This concludes our travels through the DBService. In the next chapter, we will re-implement and extend it. Before we end this chapter, we will fulfill the promise we made in the beginning of it and revisit the `main()` method of Listing 4-1, repeated here.

Listing 4-1. The `main()` *Method of DBService*

```
public static void main(String[]args)throws Exception{
  String fileName="DBServiceConfig.xml"; // default XML configuration file
  if(args.length > 0) // an alternative configuration file is supplied
    fileName=args[0];
// create an instance of DBService using an XML decoder
  XMLDecoder xmlDecoder=
      new XMLDecoder(new FileInputStream(fileName));
  DBService dbService=(DBService)xmlDecoder.readObject();
// if there are additional arguments, run the service
  if(args.length > 1){
// copy command-line args to a new array
    ArrayList aL=new ArrayList();
    for(int i=1;i<args.length;i++)
      aL.add(args[i]);
// call doSQL method of the service, output result to the screen
    dbService.doSQL(aL,new PrintWriter(System.out,true));
  }
// start DBService listening to incoming SOAP requests
// on port specified in the XML configuration file
  dbService.listenOnPort();
} // end of main()
```

As you can see, an instance of the service is created by running XMLDecoder on our XML configuration file. This invokes the default constructor and sets the properties of the service. If additional arguments are provided, we explicitly store them in an ArrayList and invoke `doSql()` with the ArrayList as the first argument. The second argument of the invocation is a PrintWriter wrapped around the System.out standard output stream that sends data to the screen. Finally, we start the normal operation of the service on its default port.

Conclusion

In this chapter, we built a lightweight Web Service that is not specific to the kind of information it sends and receives. DBService is a fairly generic tool for integrating information from different sources. Because it was built in Java by using XML encoding, it is easy to configure in XML without changing the code. Because it is a wrapper for an all-purpose DBMS, it can do very sophisticated data retrieval and modification, including finely graduated access levels for different categories of users based on HTTP's standard password authentication. Because it is a SOAP service, it can communicate with any SOAP client.

In the next chapter, we will further pursue some of these options. We will also re-implement the service in pure HTTP without a SOAP level on top of it by using the HTTP commands GET, POST, PUT, and DELETE. This so-called REST (Representational State Transfer) approach is frequently compared and contrasted to SOAP, and we are going to look at the pros and cons of the two approaches. (Somewhat surprisingly, very few changes in code will be needed to re-implement DBService as a REST application.)