14

Case Study: A Java Filesystem Web Service

So far in this book, we've seen how Web Services provide a powerful way to expose business functionality over the Internet. Now, let's take a look at a practical implementation of a unique Web Service using the Apache SOAP Toolkit and Java. This case study describes the tools and the object model of the Apache SOAP Toolkit, and then demonstrates an implementation of a functional Filesystem Web Service. All the required source code for this chapter is available from the Wrox web site at: http://www.wrox.com/.

The Filesystem Web Service

The Filesystem Web Service implements a virtual filesystem over HTTP. Using SOAP Messaging with Attachments to wrap binary and text files, distributed applications can manipulate files and directories with location transparency and security. The benefits to this Web Service are:

- Secure remote storage and retrieval of files without investing in quickly outdated hardware
- Virtual archive and automated/unattended backups
- Access to data anywhere in the world
- Virtually unlimited file storage
- Cross-platform interoperability
- Operating system and location transparency

Some target users for this Web Service may be:

- ❑ Wireless application providers and wireless device users who have no local storage
- ❑ Organizations that want to provide a personal backup solution to their users but cannot afford to install zip/tape drives in every computer
- ❑ Internet/Application Service Providers (ISPs, ASPs) who want to provide paid services to their customers
- ❑ Intranet users who require self-service backup/restore capabilities
- ❑ People who travel, and need access to their files on the road
- ❑ Individuals with no local backup devices
- ❑ Users who need data archived in a secure manner
- ❑ Individuals who require additional disk space due to small local hard drives
- ❑ Telecommuters
- ❑ Organizations that require remote backup/disaster recovery

With such a broad range of possible applications in mind, one concern for a Web Service such as this is security.

Why Do We Need Security for Web Services?

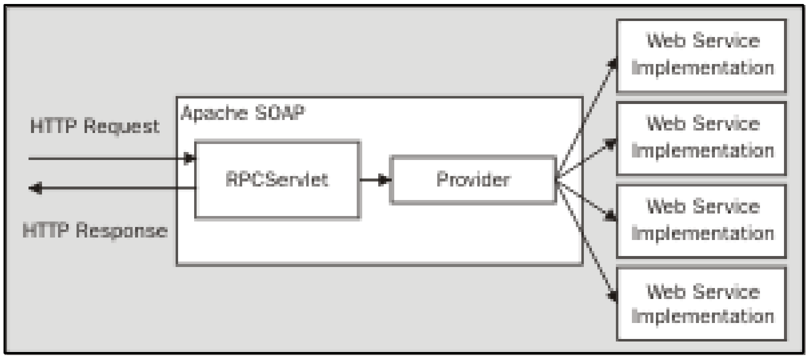
When developing web services, specifically those that transact sensitive data, you should verify that only authorized individuals utilize your web service. It makes sense that a Web Services protocol, like SOAP should provide a method for verifying the identity of its user.

Upon inspection of the SOAP specification, you will notice there is no mention of security. Why create a protocol that intentionally omits one of the prime facets of distributed computing? SOAP is actually incomplete by design. Because the SOAP specification doesn't currently address issues related to security, we utilize the built-in security features of the underlying transport protocols that are used to deliver SOAP messages. Since most Web Services rely on the HTTP protocol, we can use any of its various security implementations. Web servers, firewalls, and the like can use the information provided by HTTP to validate usernames and passwords (authentication) and grant or deny their access to resources based on the caller's identity (authorization). With SOAP on top of HTTP, we can pass username and password information in the HTTP header. The HTTP 1.1 Specification (http://www.ietf.org/rfc/rfc2616.txt) defines several challenge response authentication mechanisms, which can be used by a server to challenge a client request and by a client to provide authentication information. The general framework for access authentication, and the specification of **basic** and **digest** authentication, are defined in *"HTTP Authentication: Basic and Digest Access Authentication*" (http://www.ietf.org/rfc/rfc2617.txt).

Ultimately, a future SOAP specification should provide support for many of the available standard transport layer security mechanisms, such as basic, digest, and cryptographic messages over SSL (Secure Socket Layers, see http://home.netscape.com/eng/ssl3/index.html). The server invoking the Web Service should perform authentication and authorization, as opposed to relying on the web server or other hardware. In this fashion, we will have a robust framework through which we expose our services. These issues are especially important if you deal with sensitive information, or wish to charge a fee for your Web Services.

Apache SOAP and the Pluggable Provider

The Apache SOAP project implements a SOAP Web Services framework using Java. The bridge between the SOAP engine and the service being invoked is called the **provider**. The following diagram shows how messages flow through Apache SOAP, and where the pluggable provider fits in:


HTTP Request
HTTP Response
Apache SOAP
RPCServlet
Provider
Web Service Implementation
Web Service Implementation
Web Service Implementation
Web Service Implementation

The provider is responsible for:

- ❑ Locating, loading, and invoking the service
- ❑ Converting the result from the service into a SOAP envelope
- ❑ Implementing specialized functionality outside the scope of the SOAP specification

Here, our specialized functionality will be security. By default the SOAP engine will use the `RPCJavaProvider` class as the provider for RPC services, and the `MsgJavaProvider` class for Message services. `RPCJavaProvider` simply loads the appropriate class, invokes the desired method and converts any result into a SOAP envelope. Our pluggable provider will use the Java Database Connectivity (JDBC) interface to authenticate and authorize users against the SOAP service before invoking it. The pluggable provider demonstrated in this chapter will implement authentication and authorization through the use of a JDBC interface to a database of user profiles and Web Services. The user profiles contain username/password pairs for authentication, and a mapping of usernames to Web Services for authorization. The use of a SQL datasource will be a familiar paradigm for Java developers who already make use of database-driven security.

Setting Up The Server

In Chapter 10, we learned how to configure Apache SOAP and Tomcat. The examples in this chapter run unmodified on Apache Tomcat (available from http://jakarta.apache.org/tomcat/index.html). Remember to install the Xerces-J XML parser (http://xml.apache.org/xerces-j/). You will also need JavaMail (available from http://java.sun.com/products/javamail/; remember to specify the location of' `mail.jar` on your classpath) and the JavaBeans Activation Framework (available from http://java.sun.com/products/javabeans/glasgow/jaf.html; you'll need to set `activation.jar` on your classpath). These two packages are necessary to handle the file attachment and MIME-encoding mechanisms of this example. To compile the servlet classes, you will need the Java 2 SDK Enterprise Edition (see http://java.sun.com/j2ee/index.html). Some samples that come with Apache SOAP, such as the Calculator sample, require the Bean Scripting Framework (from http://oss.software.ibm.com/developerworks/projects/bsf) and JavaScript (which is available with Mozilla Rhino from http://www.mozilla.org/rhino/). If you would like to try these as well, place `bsf.jar` and `js.jar` in your classpath. Don't forget to add `soap.jar` to the classpath as well. Ensure that the `.jar` files are not only referenced in the classpath environment variable, which allows them to be utilized at compilation time, but that copies of them are placed in Tomcat's `/lib` folder, so that they are on the server's classpath as well. In a nutshell, here's how to hook it all up:

Some systems require you to set the `TOMCAT_HOME` (or `CATALINA_HOME` for Tomcat 4.0) environment variable, which points to the directory containing Tomcat.

Set Tomcat's classpath with `xerces.jar` at the front. In `bin/tomcat.bat`, find the `:setClasspath` section, and modify it to look like this:

```
   :setClasspath
   set CP=<path to Xerces-J>\lib\xerces.jar;%TOMCAT_HOME%\classes
```

On Unix systems, find the `export CLASSPATH` statement in `bin/tomcat.sh`, and add this line right before it:

```
   CLASSPATH=<path to Xerces-J>/lib/xerces.jar:${CLASSPATH}
```

Next, make sure Tomcat is running on port 8080. You can do this by verifying that `conf/server.xml` contains a section like this:

```
   <Connector className="org.apache.tomcat.service.PoolTcpConnector">
       <Parameter name="handler"
           value="org.apache.tomcat.service.http.HttpConnectionHandler"/>
       <Parameter name="port" value="8080"/>
   </Connector>
```

Finally, add a server context for Apache SOAP. Inside the `<ContextManager>` tag, add the following section:

```
   <Context path="/soap"
       docBase="c:/soap-2_2/webapps/soap"
       crossContext="true"
       debug="0"
       reloadable="true"
       trusted="false" >
   </Context>
```

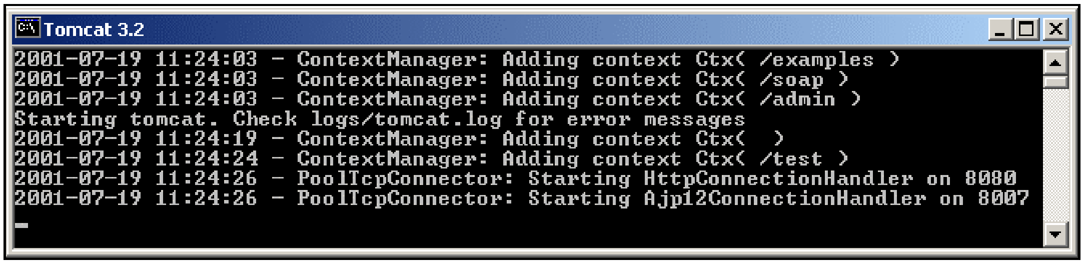
The `<Context>` section maps to a path within Tomcat in which a web application exists.

The attributes in the above element are described below:

Attribute	**Description**
`path`	The prefix of an HTTP request, instructing Tomcat which application context to use. This attribute always begins with a forward slash ("/").
`docBase`	Points to a directory which will be the root of this web application.
`reloadable`	During development, setting this value to true enables Tomcat to reload changes to your source code. This is time consuming, and reportedly error-prone. In a production environment, this should be set to false.

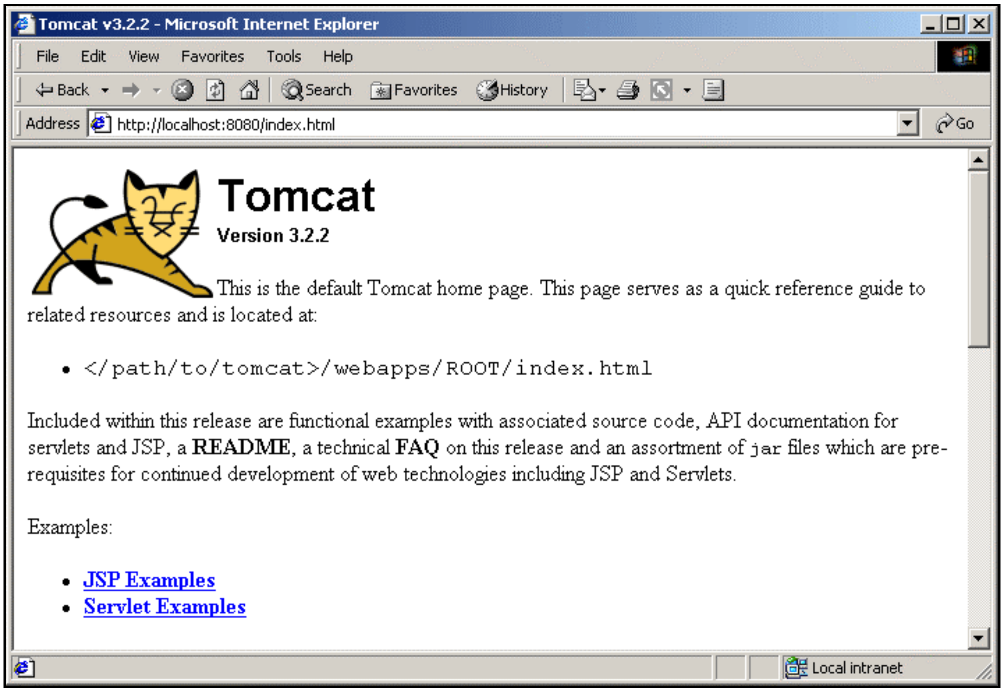
Attribute	Description
`trusted`	Enables access to Tomcat's internal objects. Most applications will set this to false.
`debug`	A value of "`0`" suppresses all console output. A value of "`9`" implies verbose mode.

If you set up everything correctly, running `bin/tomcat start` should display this console:



```
2001-07-19 11:24:03 - ContextManager: Adding context Ctx( /examples )
2001-07-19 11:24:03 - ContextManager: Adding context Ctx( /soap )
2001-07-19 11:24:03 - ContextManager: Adding context Ctx( /admin )
Starting tomcat. Check logs/tomcat.log for error messages
2001-07-19 11:24:19 - ContextManager: Adding context Ctx(  )
2001-07-19 11:24:24 - ContextManager: Adding context Ctx( /test )
2001-07-19 11:24:26 - PoolTcpConnector: Starting HttpConnectionHandler on 8080
2001-07-19 11:24:26 - PoolTcpConnector: Starting Ajp12ConnectionHandler on 8007
_
```

Point your browser to http://localhost:8080/ to verify that everything is up and running. You should see the following page:


Tomcat v3.2.2 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8080/index.html
Go
Local intranet

Tomcat

Version 3.2.2

This is the default Tomcat home page. This page serves as a quick reference guide to related resources and is located at:

- `</path/to/tomcat>/webapps/ROOT/index.html`

Included within this release are functional examples with associated source code, API documentation for servlets and JSP, a **README**, a technical **FAQ** on this release and an assortment of `jar` files which are pre-requisites for continued development of web technologies including JSP and Servlets.

Examples:

- JSP Examples
- Servlet Examples

To stop Tomcat, run `bin/shutdown` , or just kill the command window.

Setting Up The Database

For authentication and authorization, we will use a SQL database. Specifically, I've chosen MySQL (freely available from http://sourceforge.net/projects/mysql/), although any SQL database will work. MySQL, aside from being free, is very fast, and runs on both Windows and Linux. There is an excellent article on About.com entitled *How to install and configure MySQL for Windows*. The URL for this is http://perl.about.com/library/weekly/aa111400a.htm. For this example, when you are setting up MySQL, please set both the username and password to `mysql`.

This type of database-driven authentication should be familiar if you've ever set up any type of security on a web site. Our database schema is relatively simple: a table to store our principals and credentials (usernames and passwords), a table to store our Web Service identifiers (URNs), and a table to map the principals and Web Services. We use a many-to-many relationship between principals and Web Services. We want to relate many users to many Web Services. In order to decrease the duplication of data, we'll create a table consisting of nothing but primary keys from the two other tables, called `principal_webservice_map`. Here we adhere to the Fourth Normal Form (4NF), sometimes referred to as Boyce-Codd Normal Form (BCNF). This form is often overlooked, but it's important when dealing with many-to-many relations. In a nutshell, this means that any given relation may not contain more than one multivalued attribute. This concept is beyond the scope of this chapter, so I leave it as an exercise to the reader to discover its implications. Most of these rules were defined in a paper by E.F. Codd entitled *Further Normalization of the Data Base Relational Model* (this can be found in the book *Data base systems*, R. Rustin, ed., Prentice-Hall, 1972). Mathematicians later developed other normalization rules, such as 4NF, 5NF, etc. There is a good tutorial on database schema normalization at:

http://www.phpbuilder.com/columns/barry20000731.php3.

For the data definition language (DDL) in our example, we will use MySQL's syntax. This DDL can easily be ported to any SQL database with minor changes. To create our database using MySQL, place the file `FileSystem.ddl` in the `bin` folder of your MySQL directory (by default this is `C:\mysql\bin`). Change the current directory to the `bin` folder of your MySQL directory, and start the server with this command:

```
>mysqld
```

Then, execute the following command from the command prompt to create the database using the DDL:

```
>mysql < FileSystem.ddl
```

The following command will display that the new database structure was created successfully:

```
>mysqlshow filesystem
```

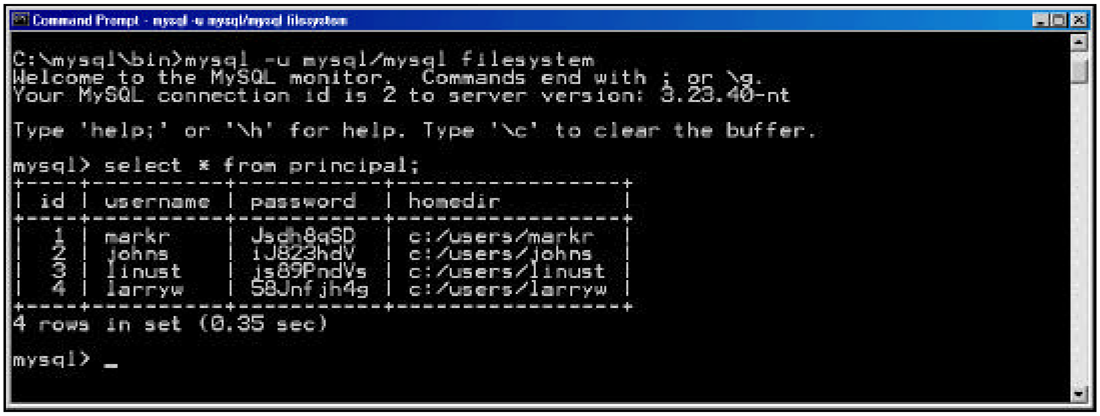
Whenever you restart MySQL, navigate to the MySQL `bin` folder in the command prompt, start the server, and enter the command:

```
>mysql -u mysql/mysql filesystem
```

Where **mysql/mysql** are your username and password. This ensures that the `filesystem` database is accessible from the database server. To check that it is accessible, enter:

```
>select * from principal;
```

at the `mysql` prompt, which will display the `principal` table of the `filesystem` database:



Command Prompt - mysql -u mysql/mysql filesystem

```
C:\mysql\bin>mysql -u mysql/mysql filesystem
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 3.23.40-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> select * from principal;
+----+----------+-----------+-----------------+
| id | username | password  | homedir         |
+----+----------+-----------+-----------------+
|  1 | markr    | Jsdh8qSD  | c:/users/markr  |
|  2 | johns    | iJ823hdV  | c:/users/johns  |
|  3 | linust   | js89PndVs | c:/users/linust |
|  4 | larryw   | 58Jnfjh4g | c:/users/larryw |
+----+----------+-----------+-----------------+
4 rows in set (0.35 sec)

mysql> _
```

Let's walk through the DDL for some further explanation of the schema (`FileSystem.ddl` in the code download for this book):

The first line creates an empty database structure, simply called "filesystem", assuming one does not already exist with the same name:

```
CREATE DATABASE IF NOT EXISTS filesystem;
```

Under Unix, database names are case-sensitive (unlike SQL keywords), so you must always refer to your database as `filesystem`, not as `Filesystem`, `FILESYSTEM`, or some other variant. This is also true for table names. Under Windows, this restriction does not apply, although you must refer to databases and tables using the same case throughout a given query.

Creating a database does not select it for use; you must do that explicitly. To make `filesystem` the current database, we `USE` the database:

```
USE filesystem;
```

The hardest part of creating a database is choosing a sound structure. What tables and columns best suit our application? Our users, or **principals**, require a table of their own:

```
'CREATE TABLE principal (
  id int(11) NOT NULL auto_increment,
  username varchar(32) NOT NULL default '',
  password varchar(32) default '',
  homedir varchar(255) NOT NULL default '',
  PRIMARY KEY (id),
  UNIQUE KEY username (username)
) TYPE=MyISAM;
```

We need some way to index and uniquely identify the principals. To that end, we create a **primary key** called `id`, which we set to `auto_increment` (also called `identity` on many systems). We also create two fields for `username` and `password`, which are self-explanatory, and a third called `homedir`, which will contain the physical directory on the web server which will host this user's files. Users also need to be unique, so we assign a `UNIQUE KEY` to the `username` field. Finally, we tell MySQL to use the `MyISAM` database type. This is the default, but you may choose from other database types, such as `BerkeleyDB`, or `InnoDB`. These other types support transactions with locking mechanisms, which are not necessary here, since our application is mostly read-only.

Next, let's create a few users for our Web Service by performing an `INSERT` into the table we just created:

```
INSERT INTO principal VALUES (1,'markr',''Jsdh8qSD'','c:/users/markr');
INSERT INTO principal VALUES (2,'johns','iJ823hdV','c:/users/johns');
INSERT INTO principal VALUES (3,'linust','js89PndVs','c:/users/linust');
INSERT INTO principal VALUES (4,'larryw','58Jnfjh4g','c:/users/larryw');
```

The passwords can be anything you like. Here, we've just used dummy data. Next, we need to create a table to hold references to our deployed Web Services. Since Apache SOAP uses a URI to uniquely identify Web Services, it makes sense for us to use the same identifiers in our database. In the `webservice` table, we create two columns: `id` and `uri`. This pair is also unique, so we create a **composite** primary key consisting of both columns. You will notice below that the `PRIMARY KEY` clause contains both columns as parameters. Again, we use the default `MyISAM` database type:

```
CREATE TABLE webservice (
  id int(11) NOT NULL auto_increment,
  uri varchar(64) NOT NULL default '',
  PRIMARY KEY  (id,uri)
) TYPE=MyISAM;
```

Next, we insert the URI for our Web Service, `urn:filesystem`:

```
INSERT INTO webservice VALUES (1,'urn:filesystem');
```

We can also insert some of the Web Services that come with Apache SOAP:

```
INSERT INTO webservice VALUES (2,'urn:AddressFetcher');
INSERT INTO webservice VALUES (3,'urn:AddressFetcher2');
INSERT INTO webservice VALUES (4,'urn:xml-soap-demo-calculator');
INSERT INTO webservice VALUES (5,'urn:sum-COM');
INSERT INTO webservice VALUES (6,'urn:adder-COM');
INSERT INTO webservice VALUES (7,'urn:po-processor');
INSERT INTO webservice VALUES (8,'urn:mimetestprocessor');
INSERT INTO webservice VALUES (9,'urn:mimetest');
INSERT INTO webservice VALUES (10,'urn:xmltoday-delayed-quotes');
INSERT INTO webservice VALUES (11,'urn:ejbhello');
INSERT INTO webservice VALUES (12,'urn:soap-unauthorized');
```

Finally, we need to create a relationship between our principals and our Web Services. Since we have a many-to-many relationship (many principals can access many Web Services), we need a **map** table (sometimes called a **join** table) to link the two. Each record in the table is the unique intersection of a principal and a Web Service. Here, the columns are foreign keys to the two other tables, represented by `p_id` and `ws_id` (principal id, and Web Service id, respectively). This pair is also a primary key to the `principal_webservice_map` table, as shown below:

```
CREATE TABLE principal_webservice_map (
  p_id int(11) NOT NULL default '0',
  ws_id int(11) NOT NULL default '0',
  PRIMARY KEY  (p_id,ws_id)
) TYPE=MyISAM;
```

The insertion of a record into this table logically defines permissions. That is, the Web Services to which a particular principal has access. For example, let's grant user `markr` (principal id: `1`) access to Web Service `urn:filesystem` (webservice id: `1`):

```
INSERT INTO principal_webservice_map VALUES (1,1);
```

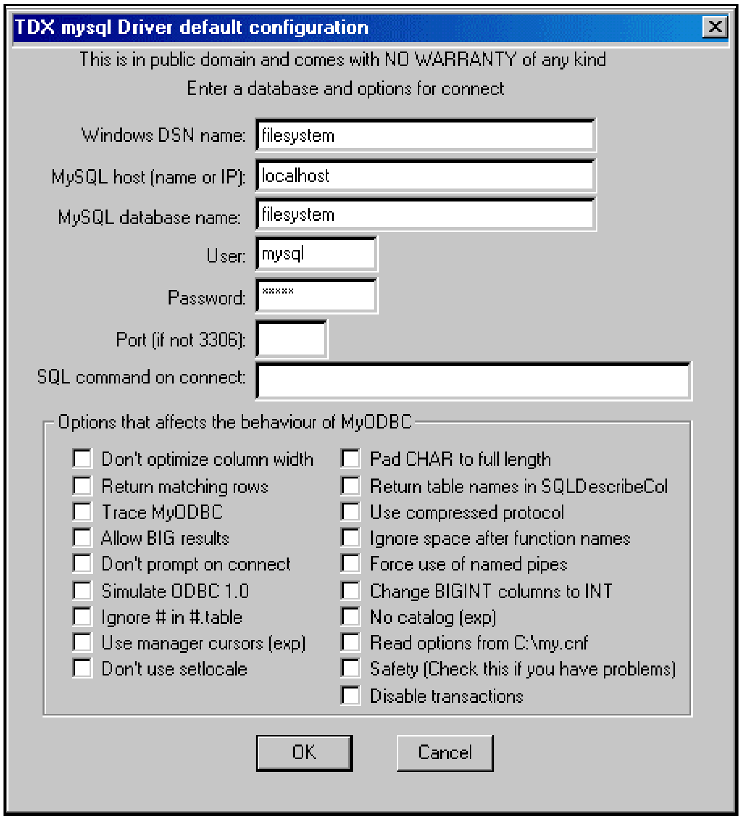
Let's go ahead and grant this user access to all other Web Services as well:

```
INSERT INTO principal_webservice_map VALUES (1,2);
INSERT INTO principal_webservice_map VALUES (1,3);
INSERT INTO principal_webservice_map VALUES (1,4);
INSERT INTO principal_webservice_map VALUES (1,5);
INSERT INTO principal_webservice_map VALUES (1,6);
INSERT INTO principal_webservice_map VALUES (1,7);
INSERT INTO principal_webservice_map VALUES (1,8);
INSERT INTO principal_webservice_map VALUES (1,9);
INSERT INTO principal_webservice_map VALUES (1,10);
INSERT INTO principal_webservice_map VALUES (1,11);
```

We can also define permissions for other users:

```
INSERT INTO principal_webservice_map VALUES (2,1);
INSERT INTO principal_webservice_map VALUES (2,4);
INSERT INTO principal_webservice_map VALUES (2,7);
INSERT INTO principal_webservice_map VALUES (2,11);
INSERT INTO principal_webservice_map VALUES (3,1);
INSERT INTO principal_webservice_map VALUES (3,5);
INSERT INTO principal_webservice_map VALUES (3,9);
INSERT INTO principal_webservice_map VALUES (4,1);
INSERT INTO principal_webservice_map VALUES (4,2);
INSERT INTO principal_webservice_map VALUES (4,4);
INSERT INTO principal_webservice_map VALUES (4,7);
INSERT INTO principal_webservice_map VALUES (4,11);
```

The next step in setting up our database is to configure our JDBC datasource. We can use either a native driver or the JDBC-ODBC bridge driver. In this case, we use the bridge driver (`sun.jdbc.odbc.JdbcOdbcDriver`) to make our provider a little more portable. Ideally, we want to use a native, or Type 4, driver. This is a pure Java driver that uses a native protocol to convert JDBC calls into the database server network protocol. Using this type of driver, the application can make direct calls from a Java client to the database. Type 4 drivers, such as the MySQL JDBC Driver, are typically offered by the database vendor. The native JDBC driver for MySQL is available from http://www.mysql.com/downloads/api-jdbc.html. Because the driver is written purely in Java, it requires no configuration on the client machine other than telling the application where to find the driver. When using a Type 4 driver, just remember to add it to the classpath of your server. In Windows 2000's Control Panel, I've created a System DSN called `filesystem` using the MySQL ODBC driver (which is available from http://www.mysql.com/downloads/api-myodbc.html). The following screenshot shows the configuration options for the ODBC driver on Windows:


TDX mysql Driver default configuration

This is in public domain and comes with NO WARRANTY of any kind

Enter a database and options for connect

Windows DSN name:	filesystem
MySQL host (name or IP):	localhost
MySQL database name:	filesystem
User:	mysql
Password:	*****
Port (if not 3306):	
SQL command on connect:	

Options that affects the behaviour of MyODBC

☐ Don't optimize column width	☐ Pad CHAR to full length
☐ Return matching rows	☐ Return table names in SQLDescribeCol
☐ Trace MyODBC	☐ Use compressed protocol
☐ Allow BIG results	☐ Ignore space after function names
☐ Don't prompt on connect	☐ Force use of named pipes
☐ Simulate ODBC 1.0	☐ Change BIGINT columns to INT
☐ Ignore # in #.table	☐ No catalog (exp)
☐ Use manager cursors (exp)	☐ Read options from C:\my.cnf
☐ Don't use setlocale	☐ Safety (Check this if you have problems)
	☐ Disable transactions

Make sure that you set both the username and password to mysql.That's all there is to setting up the SQL database. Let's move on to writing our pluggable provider's implementation.

Writing a Pluggable Provider

Since the default Java provider does not implement security in any way, we need to create our own pluggable provider. To do this, we implement the `org.apache.soap.util.Provider` interface, as all Apache SOAP providers do. This interface is as follows:

```
package org.apache.soap.util;

import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.server.*;

public interface Provider {
    public void locate(DeploymentDescriptor dd,
                       Envelope             env,
                       Call                 call,
                       String               methodName,
                          String              targetObjectURI,
                          SOAPContext         reqContext)
                   throws SOAPException;
      public void invoke(SOAPContext req, SOAPContext res) throws SOAPException;
  }
```

The `locate()` method will be called to allow the provider to verify that the service exists and is available to process the request. If an error occurs this method should throw a `SOAPException`. After a successful call to `locate()` the SOAP engine will then call `invoke()` to actually call the service. You may notice that the invoke method does not have any parameters explicitly pertaining to the Web Service; they are extracted using the SOAP envelope passed to the `RPCRouterServlet` via HTTP.

The `invoke()` method is also responsible for converting any response from the service into a SOAP envelope and placing it in the `res` parameter (a `SOAPContext`). This response is then sent back to the client via HTTP.

Now, let's walk through the implementation of `FilesystemProvider`, our pluggable provider:

```
  import org.apache.soap.rpc.* ;
  import org.apache.soap.server.* ;
  import org.apache.soap.server.http.* ;
  import org.apache.soap.util.* ;

  public class FilesystemProvider implements Provider
  {
```

Variables are declared, followed by the `locate()` method. Here, we need to grab the HTTP Basic Authentication information from the headers. This can be obtained from the `HttpServletRequest` object:

```
    public void locate( DeploymentDescriptor dd,
      Envelope env,
      Call call,
      String methodName,
      String targetObjectURI,
      SOAPContext reqContext)
    throws SOAPException {

      HttpServlet servlet = (HttpServlet)reqContext.getProperty(
      Constants.BAG_HTTPSERVLET );
      HttpSession session = (HttpSession)reqContext.getProperty(
      Constants.BAG_HTTPSESSION );
      HttpServletRequest req =
  (HttpServletRequest)reqContext.getProperty(Constants.BAG_HTTPSERVLETREQUEST);
```

HTTP encodes the authentication header using Base64 encoding. This string is in the form:

```
  Authorization: Basic username:password
```

Where `username:password` has been encoded in Base64. Now, let's extract the authentication information:

```
   String authorization = req.getHeader("authorization");
   authorization = authorization.substring(authorization.indexOf(" "));
   byte[] bytes = Base64.decode(authorization);
   String decoded = new String(bytes);
   int i = decoded.indexOf(":");
   String username = decoded.substring(0,i);
   String password = decoded.substring(i+1,decoded.length());
```

Next, we need to connect to our database so we can perform the authentication. I've created a separate data access class called `FilesystemProviderDAO` to manage the database connection and abstract out the authentication and authorization queries. We'll use the `isAuthenticated()` and `isAuthorized()` methods of this class to evaluate the caller's credentials. Our first query validates the user's password. If this function returns true, we perform our second query, which evaluates his authorization to the Web Service requested (`FilesystemProviderDAO.java`):

```
   public boolean isAuthenticated(String username, String password) {
     try  {

       // Create our Statement object
       Statement stmt = con.createStatement();

       // Execute the dynamic query
       ResultSet rs = stmt.executeQuery("SELECT password FROM " +
         "principal WHERE username = '" + username + "'");

       // Get the first (and hopefully only) record:
       if (rs.next()) {
         // Validate the password
         if(rs.getString("password").equals(password)) {
           return true;
         }
       }
       return false;
     }
     catch (SQLException e)  {
       return false;
     }
     }
```

Here, we dynamically create the query string by concatenating the `username` and `password` parameters with the `SELECT` statement. While this works, it's not very efficient under load. The next method, `isAuthorized()`, demonstrates the use of a **prepared statement** to perform its query.

```
   public boolean isAuthorized(String username, String uri) {
       try  {
       // Create a Prepared Statement object

       PreparedStatement pstmt =
         con.prepareStatement("SELECT p.username FROM " +
         "principal_webservice_map pwm, " +
         "principal p, webservice ws " +
         "WHERE pwm.p_id = p.id " +
         "AND pwm.ws_id = ws.id " +
         "AND p.username = ? AND ws.uri = ?");
```

Notice the question marks – these are placeholders for the parameters to the query. Unlike the previous example, we do not dynamically build a query string for prepared statements. We set parameters like this:

```
       pstmt.setString(1,username);
       pstmt.setString(2,uri);
```

Next, we execute the query, and compare the password field as before:

```
       ResultSet rs = pstmt.executeQuery();
       if (rs.next()) {
         if(rs.getString("username").equals(username)) {
           return true;
         }
       }
       return false;
     }
     catch (SQLException e)  {
       System.err.println("SQLException: " + e.getMessage());
       return false;
     }
     }
```

Usually, using a `PreparedStatement` is preferable, since many database servers can pre-compile or cache this type of query, improving performance. The database query can be done either way; I just wanted to demonstrate both techniques. If this function returns `true`, we can grant the caller access to the Web Service!

Next, the `FilesystemProvider` class resolves the Web Service, which is referred to as the **target object**. Apache SOAP's Service Manager performs this lookup.

```
   ServletConfig  config  = servlet.getServletConfig();
   ServletContext context = config.getServletContext();
   ServiceManager serviceManager =
       ServerHTTPUtils.getServiceManagerFromContext(context);
```

Did we perform a call on a valid method name? Let's check. If not, we need to throw an exception in the form of a SOAP Fault:

```
   if (!RPCRouter.validCall (dd, call)) {
     throw new SOAPException (Constants.FAULT_CODE_SERVER,
       "Method '" + call.getMethodName () + "' is not supported."); }
```

Now that we've successfully located the Web Service, let's set a reference to it as our target object:

```
   Object targetObject = ServerHTTPUtils.getTargetObject(serviceManager,
       dd, targetObjectURI, servlet, session, context);
```

Once our target object has been referenced, we can invoke the service:

```
   public void invoke(SOAPContext reqContext, SOAPContext resContext)
   throws SOAPException {
```

First, we perform the actual call on the target object:

```
       Response resp = RPCRouter.invoke(dd, call, targetObject, resContext);
```

Next, we build our SOAP Envelope object. This will contain our response:

```
       Envelope env = resp.buildEnvelope();
```

We need an `IOWriter` object with which to construct our XML declarations. A `StringWriter` is passed to the `marshall()` method to accomplish this:

```
       StringWriter  sw = new StringWriter();
       env.marshall(sw, call.getSOAPMappingRegistry(), resContext);
```

Our last bit of logic sets the root part of our SOAP response to the `StringWriter` object we just created, using our standard UTF-8 encoding:

```
   resContext.setRootPart(sw.toString(),
           Constants.HEADERVAL_CONTENT_TYPE_UTF8);
   }
```

Here, the `Provider` class invokes the Web Service as identified by the target object and builds a SOAP Response envelope to encapsulate our return value, encoded in SOAP's XML Envelope format.

Writing the Filesystem Web Service

Coding the actual Web Service is the simple part of this exercise. Here, we will write a Web Service that exposes several methods, exposing some of the functionality of the `java.io.File` class as a Web Service. Since we are using the HTTP Header information in the SOAP context for authentication information, we need to have access to that information in our Web Service implementation. You will notice each public method has a parameter of type `SOAPContext`. These method signatures do not match those expected by the `FilesystemProxy` class (shown later in this chapter). If a service's public method with a matching signature is not found, a second search is done by our `FilesystemProvider` for a method with an initial parameter of type `SOAPContext`.

You should be forewarned that using the SOAP context information in your Web Service implementation will bind you to Apache SOAP. This is one published issue in the Apache SOAP release notes, and you are advised to use this technique carefully.

Having access to the incoming `SOAPContext` provides you with access to the following objects via the `SOAPContext.getProperty()` method:

- `HttpServlet`, using the key `org.apache.soap.Constants.BAG_HTTPSERVLET`
- `HttpSession`, using the key `org.apache.soap.Constants.BAG_HTTPSESSION`
- `HttpServletRequest`, using the key `org.apache.soap.Constants.BAG_HTTPSERVLETREQUEST`
- `HttpServletResponse`, using the key `org.apache.soap.Constants.BAG_HTTPSERVLETRESPONSE`

If the original SOAP request was in the SOAP Attachments form, then you can reference the MIME-encoded attachments using the `getBodyPart()` method.

Using SOAP Attachments

In this section, you will see how SOAP Attachments are used to send and receive file attachments as part of the `createNewFile()` and `getFile()` methods in our Web Service. Apache SOAP allows data to be passed along with the XML message without having to embed the data in the XML itself. Since our SOAP message take the form of a MIME multipart message (this is defined in the specification – see http://www.w3.org/TR/SOAP-attachments), entities such as files can be embedded inside.

Below is our Web Service implementation. (This is the file `Filesystem.java` from the code download):

```
package com.markrichman.filesystem;

import java.io.*;
import java.sql.*;
import java.util.Date;
import java.util.Enumeration;
import javax.activation.*;
import javax.mail.internet.*;
import javax.servlet.* ;
import javax.servlet.http.* ;
import org.apache.soap.encoding.soapenc.Base64;
import org.apache.soap.rpc.SOAPContext;
import org.apache.soap.util.mime.*;
import org.apache.soap.util.xml.*;

/**
 * @author Mark A. Richman
 * @version 1.0
 */
public class Filesystem {
```

We will go through each of the methods, explaining how the whole attachment mechanism works. Many of the method names will remind you of those from `java.io.File`. This is intentional, as our Web Service should behave as transparently as possible from a developer's point of view. In fact, most of these methods simply wrap the `File` class methods entirely.

The `copyTo()` method simply copies a file to a new location:

```
   public void copyTo(SOAPContext ctx, String sourcePath, String destPath) throws
   Exception {
           File fnew = new File(getHomedir(getUsername(ctx)) + "/" + destPath);
           File fold = new File(getHomedir(getUsername(ctx)) + "/" + sourcePath);

           InputStream in = null;
           OutputStream out = null;
           try {
               in = new FileInputStream(fold);
               out = new FileOutputStream(fnew);
               while (true) {
                   int data = in.read();
                   if (data == -1) {
                       break;
                   }
                   out.write(data);
               }
               in.close();
               out.close();
               }
           finally {
               if (in != null) {
                   in.close();
                   }
               if (out != null) {
                   out.close();
                   }
           }
       }
```

Note the two methods below, which are used throughout this class:

- `getHomedir` – Gets a user's home directory from the database via the `FilesystemDAO` class
- `getUsername` – Gets the username from the SOAP Context object

`createNewFile` creates a new file, relative to the user's home directory (see the DDL above):

```
   public void createNewFile(SOAPContext ctx, String filePath) throws Exception {

           File fnew = new File(getHomedir(getUsername(ctx)) + "/" + filePath);

           if(!fnew.createNewFile()) {
               throw new Exception("File already exists: " + filePath);
           }
```

Next, we must create two objects: a `MimeBodyPart` object to get at the SOAP attachment, and a `DataHandler` object to get an `InputStream` for reading. The JavaMail API provides this functionality for us:

```
            MimeBodyPart mbp;
            DataHandler dh;
            Object o;
            InputStream is;
            FileOutputStream fos = new FileOutputStream(fnew);

            try {
                mbp = ctx.getBodyPart(1);
                dh = mbp.getDataHandler();
                is = dh.getInputStream();
```

Now, we simply write out the file on the server:

```
                int c;
                while ((c = is.read()) != -1)
                    fos.write(c);
            }
            catch(Exception e) {
                throw new Exception(e.getMessage());
            }

            return;
        }
```

The `delete`, method deletes a file:

```
    public boolean delete(SOAPContext ctx, String filePath) throws Exception {
            File fnew = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
            return fnew.delete();
        }
```

We can also check the existence of a file on the server using the `exists`, method:

```
    public boolean exists(SOAPContext ctx, String filePath) throws Exception {
        File f = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
        return f.exists();
    }
```

This method uses the JavaMail API in the reverse way of the `createFile`, method. Here, we send a file back to the client using the `DataHandler`:

```
    public DataHandler getFile(SOAPContext ctx, String filePath) throws Exception {
        File f = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
        if(!f.exists())
            throw new Exception("File not found.");

        try {
            DataSource ds = new ByteArrayDataSource(new File(f), null);
            DataHandler dh = new DataHandler(ds);
            return dh;
        }
        catch(Exception e) {
            System.err.println(e.getMessage());
        }
    }
```

`length`, gets a file's length in bytes:

```
    public long length(SOAPContext ctx, String filePath) throws Exception {
        File f = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
        return f.length();
    }
```

`getParent`, gets a file's parent directory:

```
    public String getParent(SOAPContext ctx, String filePath) throws Exception {
        File f = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
        return f.getParent();
    }
```

`isDirectory`, asks if a file descriptor is a directory.

```
    public boolean isDirectory(SOAPContext ctx, String filePath) throws Exception {
        File f = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
        return f.isDirectory();
    }
```

`isFile`, tests to see whether the file is a **normal** file. A file is normal if:

- It is not a directory
- It satisfies other operating system-dependent criteria (i.e. symbolic links on Unix)

Any non-directory file created by a Java application is guaranteed to be a normal file.

```
    public boolean isFile(SOAPContext ctx, String filePath) throws Exception {
        File f = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
        return f.isFile();
    }
```

`lastModified,` returns the time that the file denoted by the filePath was last modified:

```
    public long lastModified(SOAPContext ctx, String filePath) throws Exception {
        File f = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
        return f.lastModified();
    }
```

`list` returns an array of strings naming the files and directories in the directory, assuming the filename we give represents a directory. This again simply encapsulates the functionality of `java.io.File`:

```
public String[] list(SOAPContext ctx, String filePath) throws Exception {
    File f = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
    return f.list();
}
```

`mkdir` creates a directory on the server, given a parent directory specified by `filePath`:

```
public boolean mkdir(SOAPContext ctx, String filePath) throws Exception {
    File f = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
    return f.mkdir();
}
```

`mkdirs` creates the directory named by the `filePath`, including any necessary parent directories that do not already exist:

```
public boolean mkdirs(SOAPContext ctx, String filePath) throws Exception {
    File f = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
    return f.mkdirs();
}
```

`renameTo` renames a file:

```
public boolean renameTo(SOAPContext ctx, String filePath, String newName) throws
Exception {
        File f = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
        return f.renameTo(new File(getUsername(ctx) + "/" + newName));
}
```

`setLastModified` sets the last-modified time of a file or directory:

```
public boolean setLastModified(SOAPContext ctx, String filePath, long time) throws
Exception {
    File f = new File(getHomedir(getUsername(ctx)) + "/" + filePath);
    return f.setLastModified(time);
}
```

Here is our `getUsername()` method. It's important to understand how the HTTP header information is extracted, so we'll go into some detail here. We are particularly interested in the `Authorization` field in the HTTP headers. We can access this simply by calling `getHeader()` on the request object:

```
protected String getUsername(SOAPContext ctx) {
        HttpServletRequest req =
(HttpServletRequest)ctx.getProperty("HttpServletRequest");
        String authorization = req.getHeader("Authorization");
```

For HTTP Basic Authorization, the authorization header is of the form:

```
Authorization: Basic {Base64-encoded username & password}
```

Where the username and password look like "`username:password`" prior to encoding with the Base64 algorithm. Using some fancy substring manipulation, we get the encoded username and password string:

```
        authorization = authorization.substring(authorization.indexOf(" ")); //
   strip "Basic "
```

Next, decode it using `Base64.decode()`:

```
            byte[] bytes = Base64.decode(authorization);
            String decoded = new String(bytes);
```

The username is what we're after here. It's on the left side of the colon:

```
            int i = decoded.indexOf(":");
            String username = decoded.substring(0,i);
            return username;
        }
```

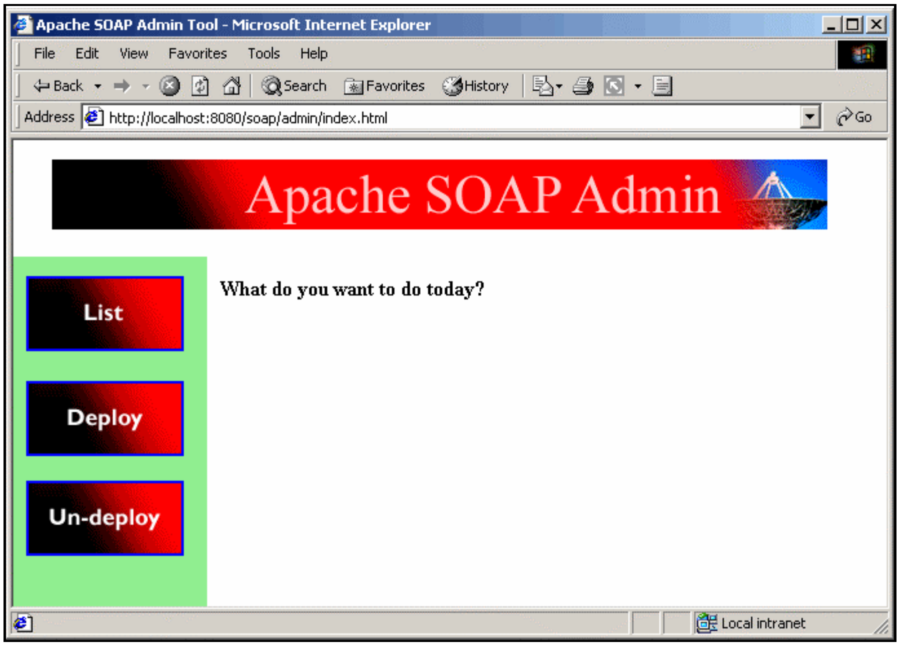
`getHomedir` gets the user's home directory via the `FilesystemDAO` data access object:

```
        /**
        * Gets the user's home directory via the FilesystemDAO
        * data access object.
        */
        protected String getHomedir(String username) throws Exception {
            FilesystemDAO dao = new FilesystemDAO();
            try {
                return dao.getHomedir(username);
            }
            catch (Exception e) {
                throw new Exception(e.getMessage());
            }

        }

    }
```

Once compiled, you can add `Filesystem.class` to your classpath, either under the `com/markrichman/filesystem/` path, or in a `.jar` file containing this package structure. We'll cover packaging a bit later.

Deploying Web Services Using Pluggable Providers

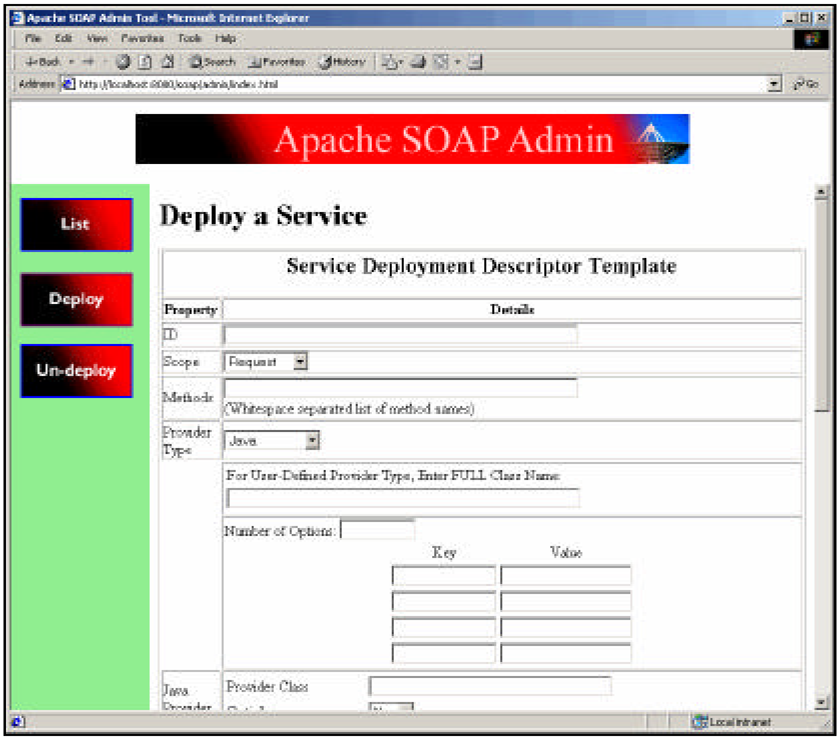
Web Services can be deployed via the command line or through the Apache SOAP Web Admin. Make sure your web server is running, and launch http://localhost:8080/soap/ from your browser. Click Run the admin client when you see the welcome screen. You should then see the following screen:


Apache SOAP Admin Tool - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Go
Apache SOAP Admin
List
Deploy
Un-deploy
Local intranet

http://localhost:8080/soap/admin/index.html

What do you want to do today?

This screen shows the three options that allow you to list, deploy, and un-deploy your Web Services. Since we have no Web Services deployed yet, go ahead and click Deploy. You will be presented with the following screen:


Apache SOAP Admin Tool - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Go
Apache SOAP Admin
List
Deploy
Un-deploy
Local intranet

http://localhost:8080/soap/admin/index.html

Deploy a Service

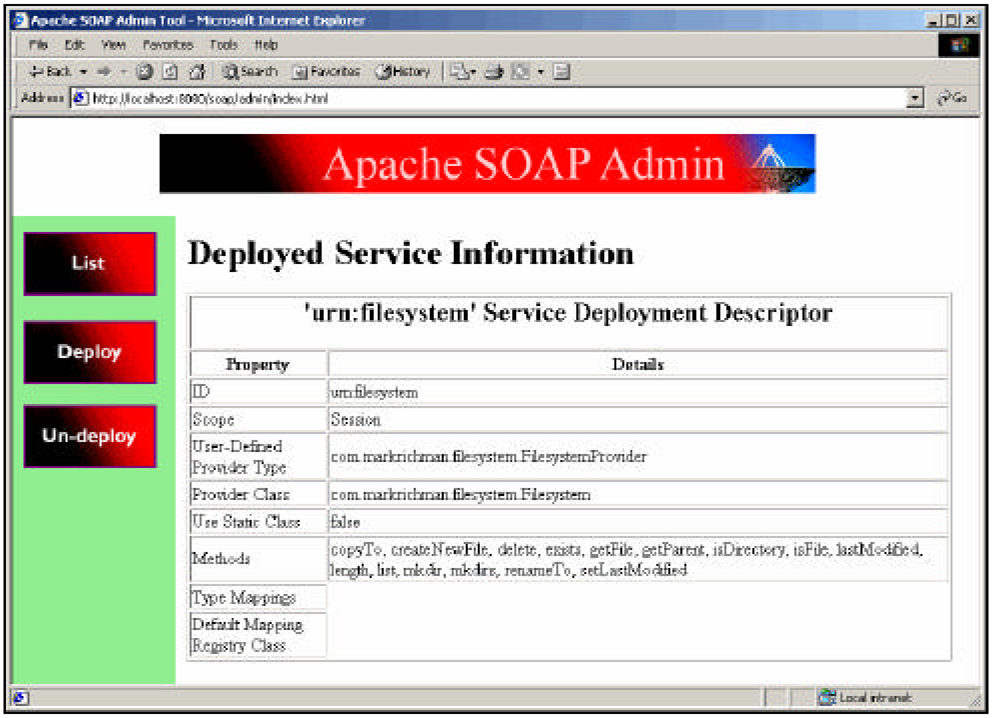
Service Deployment Descriptor Template

Property	Details
ID	
Scope	Request
Methods	(Whitespace separated list of method names)
Provider Type	Java
	For User-Defined Provider Type, Enter FULL Class Name
	Number of Options: Key Value
Java Provider	Provider Class

Let's look at each property in the deployment screen:

Property	Description
ID	A URN that uniquely identifies the service to clients. It must be unique among the deployed services, and be encoded as a URI. We commonly use the format: "`urn:UniqueServiceID`". It corresponds to the target object ID, in the terminology of the SOAP specification. I chose `urn:filesystem`, but you can really pick any locally unique string you like. See http://www.ietf.org/rfc/rfc2141.txt for a full description of the URN syntax.
Scope	Defines the lifetime of the object serving the invocation request. This corresponds to the scope attribute of the `<jsp:useBean>` tag in JSP. This tag can have one of the following values: ❑ Request – a new object is created for each request, and is available for the complete duration of the request. ❑ Session – a single instance of the object is created and available for the complete duration of the session. ❑ Application – a single instance of the object is created and available for the complete duration of the application. That is, until the server is shut down.
Method list	Defines the names of the method that can be invoked on this service object. We have the following methods: `copyTo`, `createNewFile`, `delete`, `exists`, `getFile`, `getParent`, `isDirectory`, `isFile`, `lastModified`, `length`, `list`, `mkdir`, `mkdirs`, `renameTo`, and `setLastModified`.
Provider type	Indicates whether the service is implemented using Java or a scripting language. We obviously are using Java, and our provider's full class name is: `com.markrichman.filesystem.FilesystemProvider`.
Provider class (for Java services)	Fully specified class name of the target object servicing the request. Our pluggable provider is called `com.markrichman.filesystem.Filesystem`.

We can skip the rest of the options for the purposes of this chapter. Once you've entered the fields, scroll down and click the `Deploy` button at the bottom of the window. You should see a screen that indicates that the service has been deployed. If you click on the `List` button, you'll see the URN of your Web Service is listed. Click on its link and you should see the deployed service information screen:


Apache SOAP Admin Tool - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8080/soap/admin/index.html
Go
Apache SOAP Admin
List
Deploy
Un-deploy
Local intranet

Deployed Service Information

'urn:filesystem' Service Deployment Descriptor

Property	Details
ID	urn:filesystem
Scope	Session
User-Defined Provider Type	com.markrichman.filesystem.FilesystemProvider
Provider Class	com.markrichman.filesystem.Filesystem
Use Static Class	false
Methods	copyTo, createNewFile, delete, exists, getFile, getParent, isDirectory, isFile, lastModified, length, list, mkdir, mkdirs, renameTo, setLastModified
Type Mappings	
Default Mapping Registry Class	

Alternatively, we can deploy the Web Service from the command prompt. To do this, we use a deployment descriptor in XML format. The pluggable provider is simply placed as the full classname in the deployment descriptor's type attribute (`DeploymentDescriptor.xml`):

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:filesystem">
  <isd:provider type="com.markrichman.filesystem.FilesystemProvider"
                scope="Session"
                methods="copyTo createNewFile delete exists getFile getParent
isDirectory isFile lastModified length list mkdir mkdirs renameTo
setLastModified">
    <isd:java class="com.markrichman.filesystem.Filesystem"/>
  </isd:provider>
</isd:service>
```

Use the deployment descriptor above to fully describe the service. To deploy the Web Service from the command line:

```
java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/soap/servlet/rpcrouter deploy DeploymentDescriptor.xml
```

The `ServiceManagerClient` is SOAP client provided with the distribution that communicates with the `ServiceManager`. Your Web Service will be configured and registered by the Service Manager automatically.

Writing the Filesystem Proxy Class

Invoking a Web Service using the Apache SOAP client API is relatively straightforward. Here, we have created a client proxy class called `FilesystemProxy` to help abstract out a lot of the SOAP internals. You will use this class from your client applications to interact with the Filesystem Web Service. Apache SOAP Web Services are invoked using the `Call` object (`org.apache.soap.rpc.Call`). This object is configured with the endpoint's URL, the target object's URI, method name, and any parameters. In this example, the URL we connect to will be `http://localhost:8080/soap/servlet/rpcrouter`. Remember earlier in this chapter, we configured the server to look for SOAP services on port 8080, with context `/soap.` This URL is bound to `org.apache.soap.server.http.RPCRouterServlet` on the server (see /`webapps/soap/WEB-INF/web.xml`). We also construct a `SOAPHTTPConnection` object to capture the HTTP basic authentication information. The URI of the method call element is used as the object ID on the remote side. Since our Filesystem service takes no parameters, our `params` object is empty (you will notice, however, that the Filesystem Web Service captures the `SOAPContext` parameter transparently). We can now call the `invoke()` method of our `Call` object, which returns a `Response` object. If any exceptions were thrown on the server, the `Fault` object will contain those details. The `resp` object will encapsulate the SOAP response. Calling the `Response.getReturnValue().getValue()` will display the return value of an object. All other types must be cast from object to their expected types. The following code is available from the download as `FilesystemProxy.java`. With the Java Activation Framework and JavaMail on your classpath, you can compile the code with the following command:

```
javac com/markrichman/filesystem/FilesystemProxy.java
```

Again, let's walk through this class's implementation. As the majority of the functions in this section are fairly repetitive, I will just display the most interesting ones:

```
package com.markrichman.filesystem;

import java.io.*;
import java.net.URL;
import java.util.Vector;
import javax.activation.*;
import javax.mail.internet.*;
import org.apache.soap.Constants;
import org.apache.soap.rpc.Call;
import org.apache.soap.Fault;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;
import org.apache.soap.transport.http.SOAPHTTPConnection;
import org.apache.soap.util.mime.*;

public class FilesystemProxy {
```

The `createNewFile()` method sets up our `Call` object as usual. Notice the `attach()` method. This is responsible for the MIME functionality used in the SOAP with Attachments specification.

```
  public boolean createNewFile(String filePath, File file) throws Exception {
        Call call = new Call ();
        call.setTargetObjectURI(targetObjectURI);
        call.setMethodName("createNewFile");
        SOAPHTTPConnection hc = new SOAPHTTPConnection();
        hc.setUserName(username);
        hc.setPassword(password);
        call.setSOAPTransport(hc);
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
        Vector params = new Vector ();
        params.addElement (new Parameter("filePath", String.class, filePath, null));
        call.setParams(params);

        System.out.println("Attaching file: " + file.getName());
        attach(file,call);   // MIME Attachment

        Response resp = call.invoke (new URL(url), "");

        if (resp.generatedFault()) {
           Fault fault = resp.getFault ();
           System.out.println ("Ouch, the call failed: ");
           System.out.println ("  Fault Code   = " + fault.getFaultCode ());
           System.out.println ("  Fault java.lang.String = " + fault.getFaultString
  ());
           throw new Exception(fault.getFaultString());
        }
        else {
           return true;
        }
     }
```

Here is the code to actually perform the download. We get the file using the DataHandler. We use a protected utility function called `detach()` to assist us:

```
     public void getFile(String filePath, String local) throws Exception {
        Call call = new Call ();
        call.setTargetObjectURI(targetObjectURI);
        call.setMethodName("getFile");
        SOAPHTTPConnection hc = new SOAPHTTPConnection();
        hc.setUserName(username);
        hc.setPassword(password);
        call.setSOAPTransport(hc);
        Vector params = new Vector ();
        params.addElement (new Parameter("filePath", String.class, filePath, null));
        call.setParams (params);

        Response resp = call.invoke (new URL(url), "");

        if (resp.generatedFault ()) {
            Fault fault = resp.getFault ();
            System.out.println ("Ouch, the call failed: ");
            System.out.println ("  Fault Code   = " + fault.getFaultCode ());
            System.out.println ("  Fault java.lang.String = " + fault.getFaultString
   ());
            throw new Exception(fault.getFaultString());
         }
         else {
            Parameter result = resp.getReturnValue ();
            System.out.println ( result.getValue() );
            File f = new File(local);
            detach(resp,f);
         }
      }
```

Another interesting method – here we pass back a string array. Apache SOAP handles this for us automatically. List all the files in a directory, returned as a string array:

```
      public String[] list(String filePath) throws Exception {
         Call call = new Call ();
         call.setTargetObjectURI(targetObjectURI);
         call.setMethodName("length");
         SOAPHTTPConnection hc = new SOAPHTTPConnection();
         hc.setUserName(username);
         hc.setPassword(password);
         call.setSOAPTransport(hc);
         Vector params = new Vector ();
         params.addElement (new Parameter("filePath", String.class, filePath, null));
         call.setParams (params);

         Response resp = call.invoke (new URL(url), "");

         if (resp.generatedFault ()) {
            Fault fault = resp.getFault ();
            System.out.println ("Ouch, the call failed: ");
            System.out.println ("  Fault Code   = " + fault.getFaultCode ());
            System.out.println ("  Fault java.lang.String = " + fault.getFaultString
   ());
            throw new Exception(fault.getFaultString());
         }
         else {
            Parameter result = resp.getReturnValue ();
            System.out.println ( result.getValue() );
            String[] s = (String[])result.getValue();
            return s;
         }
      }
```

Here is our utility function for attaching a file to our `Call` object. This should look very familiar, however we add one line of code to add the `MimeBodyPart` to the `Call` object. The SOAP framework will marshall the binary attachment for us:

```
      protected void attach(File file, Call call) {
         try {
            DataSource ds = new ByteArrayDataSource(file,null);
            DataHandler dh = new DataHandler(ds);
            MimeBodyPart part = new MimeBodyPart();
            part.setDataHandler(dh) ;
            call.addBodyPart(part);
         }
         catch(Exception e) {
            System.err.println(e.getMessage());
         }

      }
```

Now, we can detach the MIME attachment. We simply get at the `MimeBodyPart`; since we only have one attachment in this instance, we pass `getBodyPart()` a parameter of `1`. The attachment is saved to a local file, specified by `f`.

```
      protected void detach(Response resp, File f) {
         // Write the data
         try {
            MimeBodyPart mbp;
            DataHandler dh;
            Object o;
            InputStream is;
            FileOutputStream fos = new FileOutputStream(f);

            mbp = (MimeBodyPart)resp.getBodyPart(1);
            dh = mbp.getDataHandler();
            is = dh.getInputStream();

            int c;
            while ((c = is.read()) != -1)
               fos.write(c);
         }
         catch(Exception e) {
            System.err.println(e.getMessage());
         }
   }
   }
```

Writing the Filesystem Client Class

The `Filesystem` client class is relatively simple. It simply instantiates the `FilesystemProxy` class, and exercises a few of its functions. Feel free to modify this implementation to test on your own. You'll want to specify a file on your local system, represented here as `a.gif.` The following code is available in the code download as `FilesystemClient.java`:

```
package com.markrichman.filesystem;

public class FilesystemClient {

    public static void main (String[] args) {
        try {
            FilesystemProxy fs = new FilesystemProxy();
            fs.setUsername("markr");
            fs.setPassword("Jsdh8qSD");

            for(int i=0;i<10;i++) {
                if(fs.exists(i+"a.gif"))
                    continue;
                fs.createNewFile(i+"a.gif", new java.io.File("a.gif"));
                System.out.println("Last Modified: " +
fs.lastModified(i+"a.gif"));
```

We specify the new last-modified time, measured in millis econds since the epoch (00:00:00 GMT, January 1, 1970):

```
                fs.setLastModified(i+"a.gif", 153055769);
                System.out.println("New Last Modified: " +
fs.lastModified(i+"a.gif"));
                System.out.println(fs.length(i+"a.gif"));
                fs.copyTo(i+"a.gif", i+"xa.gif");
                fs.delete(i+"a.gif");
                fs.delete(i+"xa.gif");
            }
        }
        catch(Exception e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Packaging the .jar File

I prefer to use a `.jar` file, so I can package my Web Service, pluggable provider, and database classes all together. We create our `.jar` file as follows:

```
jar cvf Filesystem.jar com/markrichman/filesystem/*.class
```

The resulting `.jar` file will be structured like this:

```
META-INF/MANIFEST.MF
com/markrichman/filesystem/Filesystem.class
com/markrichman/filesystem/FilesystemClient.class
com/markrichman/filesystem/FilesystemDAO.class
com/markrichman/filesystem/FilesystemProvider.class
com/markrichman/filesystem/FilesystemProxy.class
```

Dropping this `.jar` file into Tomcat's `/lib` folder automatically adds it to the classpath on startup.

Trying It Out

There are a few minor prerequisites before you fire up the client. You need to create a folder called `C:\users`. This is the `homedir` value in the `principal` table specified in `Filesystem.ddl`. The test file `a.gif` will be uploaded and saved there by the web service. The `a.gif` file is a test file - you can alter `FilesystemClient.java` to point to whichever file you like on your hard drive.

To execute the `FilesystemClient`, run the following at the command prompt:

```
>java com.markrichman.filesystem.FilesystemClient
```

If all goes well, the client will be authenticated against the `Filesystem` Web Service. You should also see some informational text in the web server's console. Now, test out what happens when you use the wrong password, or delete the reference in the `principal_webservice_map` table in the database. There is a Web Service called `urn:soap-unauthorized` in the database for which no user has permissions. Try using this as the parameter for `call.setTargetObjectURI()`. Bad username/password combinations generate the fault string Bad password, and authorization failures generate a User not authorized fault. I'll leave it as an exercise to the reader to explore the strength of this mechanism.

Summary

I hope you enjoyed this tour of the features and facilities offered by the Apache SOAP Toolkit. The intention throughout this case study has been to provide a solid starting point for you to explore the areas of Web Service development that are of interest to you. Hopefully, this chapter has inspired you to develop unique Web Services of your own with Apache SOAP and Java.

For further work, here are some suggestions on how to build on this case study:

- Implement the data access in LDAP, as opposed to JDBC.
- Create a wireless device application that performs the client and proxy functionality. One application here is virtual local storage for handheld devices with no local storage of their own.
- Implement an offsite backup solution via SOAP and HTTP.
- Build upon this service to add versioning for your own configuration management system.