.NET Test Automation Recipes

A Problem-Solution Approach

James D. McCaffrey

Apress®

.NET Test Automation Recipes: A Problem-Solution Approach

ISBN-13: 978-1-59059-663-0

ISBN-10: 1-59059-663-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Hassell
Technical Reviewer: Josh Kelling
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Elizabeth Seymour
Copy Edit Manager: Nicole LeClerc
Copy Editor: Julie McNamee
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Lynn L'Heureux
Proofreader: Elizabeth Berry
Indexer: Becky Hornak
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

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, e-mail `info@apress.com`, or visit `http://www.apress.com`.

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

CHAPTER 8

Web Services Testing

8.0 Introduction

The techniques in this chapter show you how to test ASP.NET Web services. You can think of a Web service as a collection of methods that resides on a server machine, which can be called by a client machine over a network. Web services often expose data from a SQL database. For example, suppose you own a company that sells books. You want your book data available to other companies' Web sites to expand your reach. However, you don't want to allow direct access to your databases. One solution to this problem is for you to create an ASP.NET Web service that exposes your book data in a simple, standardized, and secure way. Figure 8-1 shows a demonstration Web application. Users can query a data store of book information. What is not obvious from the figure is that the data displayed on the Web application comes from an ASP.NET Web service, rather than directly from a SQL database.

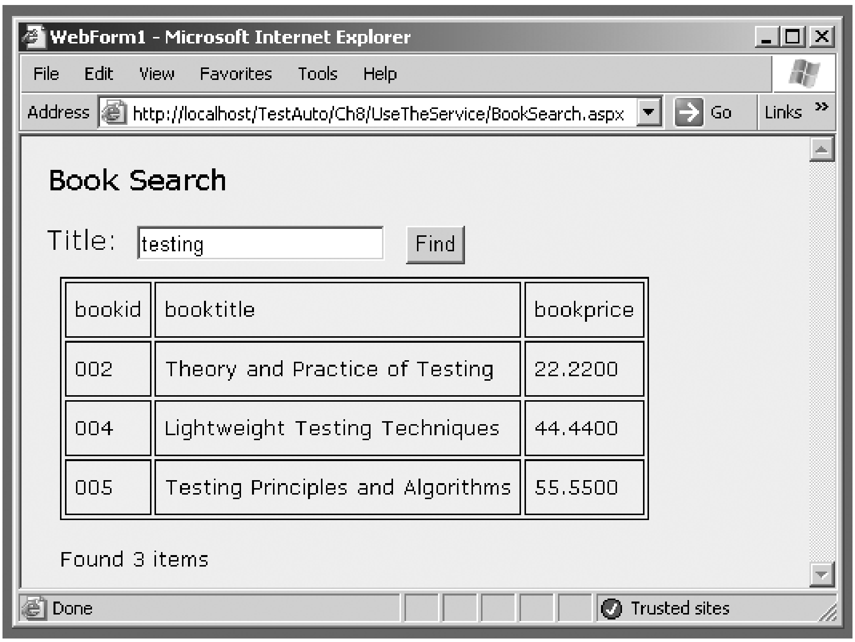

Figure 8-1. *Web application using an ASP.NET Web service*

Behind the scenes, there is an ASP.NET Web service in action. This Web service accepts requests for data from the Web application `BookSearch.aspx`, pulls the appropriate data from a backend SQL database, and returns the data to the Web application where it is displayed. The Web service has two methods. The first is `GetTitles()`, which accepts a target string as an input argument and returns a `DataSet` of book information where the book titles contain the target string. This is the method being called by the Web application in Figure 8-1. The second Web method is `CountTitles()`, which also accepts a target string but just returns the number of titles that contain the string. The terms Web service and Web method are often used interchangeably.

Testing the methods of a Web service is conceptually similar to the API testing described in Chapter 1—you pass input arguments to the method under test, fetch a return value, and compare the actual return value with an expected value. The main difference is that because Web methods reside on a remote computer and are called over a network, they may be called in several different ways. The fundamental communication protocol for Web services is SOAP (Simple Object Access Protocol). As you'll see, SOAP is nothing more than a particular form of XML. Because of this, Web services are sometimes called XML web services. Although Web services are transport protocol-independent, in practice, Web services are almost always used in conjunction with HTTP. So when a typical Web service request is made, the request is encapsulated in a SOAP/XML packet. That packet is in turn encapsulated in an HTTP packet. The HTTP request packet is then sent via TCP/IP. The TCP/IP packet is finally sent between two network sockets as raw bytes. To hide all this complexity, Visual Studio .NET can call Web methods and receive the return values in a way called a Web service proxy mechanism. Therefore, there are four fundamental ways to call a Web method in an ASP.NET Web service. Listed in order from highest level of abstraction and easiest to code, to lowest level of abstraction, following are the ways to call a Web method:

- Using a Proxy Mechanism (Section 8.1)
- Using HTTP (Section 8.3)
- Using TCP (Section 8.4)
- Using Sockets (Section 8.2)

The techniques in this chapter demonstrate each of these four techniques. Figure 8-2 shows such a run.

Test case #001 in the test run in Figure 8-2 corresponds to the user input and response in Figure 8-1. Each test case is run twice: first by sending test case input and receiving a return value at a high level of abstraction using the proxy mechanism, and second by sending and receiving at a lower level of abstraction using the TCP mechanism. The idea behind testing a system in two different ways is validation. If you test your system in two different ways using the same test case data, you should get the same test results. If you don't get identical results, then the two test approaches are not testing the same thing, and you need to investigate. Notice that test case `002` produces a pass result when calling the `GetTitles()` method with input *and* via TCP, but a fail result when calling via the proxy mechanism. (Test case `002` contains a deliberately faulty expected value to demonstrate the idea of validation.) Validation is closely related to verification. We often say that verification asks if the SUT works correctly, whereas validation asks if we are testing correctly. However, the two terms are often used interchangeably.

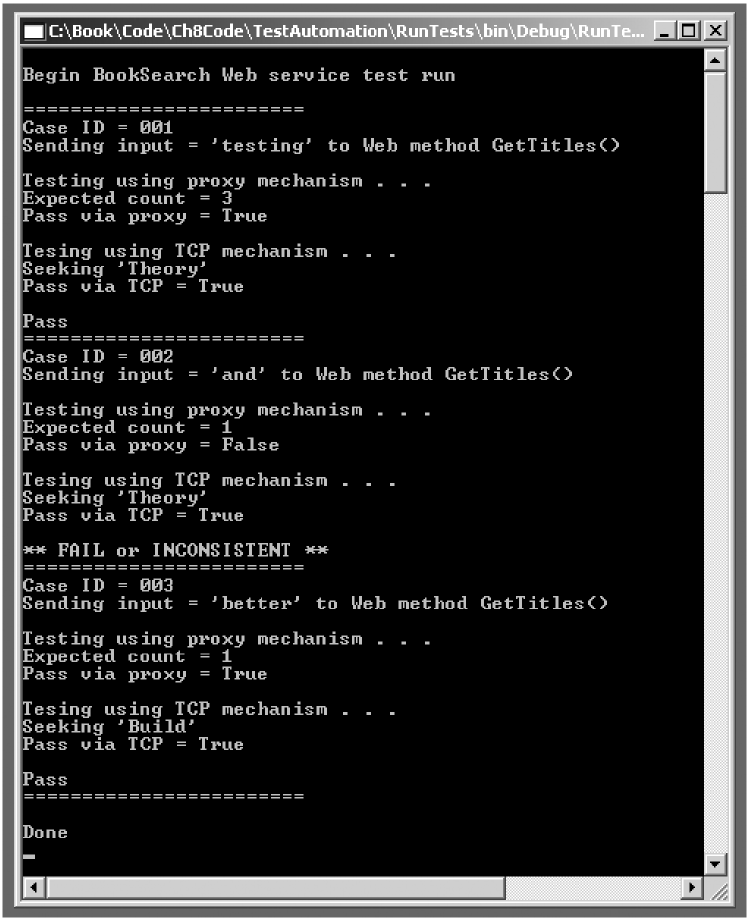

Figure 8-2. *Web service test run with validation*

Many of the techniques in this chapter make reference to the Web service, which supplies the data to the Web application shown previously in Figure 8-1. The Web service is based on a SQL database. The key SQL statements that create and populate the database are

```
create database dbBooks
go

use dbBooks
go

create table tblBooks
(
bookid char(3) primary key,
booktitle varchar(50) not null,
bookprice money not null
)
```

```
go

insert into tblBooks values('001','First Test Automation Principles',11.11)
insert into tblBooks values('002','Theory and Practice of Testing',22.22)
insert into tblBooks values('003','Build Better Software through Automation',33.33)
insert into tblBooks values('004','Lightweight Testing Techniques',44.44)
insert into tblBooks values('005','Testing Principles and Algorithms',55.55)
go

exec sp_addlogin 'webServiceLogin', 'secret'
go

-- grant execute permissions to webServiceLogin here
```

The database dbBooks contains a single table, tblBooks, which has three columns: bookid, booktitle, and bookprice. The table is populated with five dummy records. A SQL login named webServiceLogin is associated with the database. Two stored procedures are contained in the database to access the data:

```
create procedure usp_GetTitles
 @filter varchar(50)
as
 select * from tblBooks where booktitle like '%' + @filter + '%'
go

create procedure usp_CountTitles
 @filter varchar(50)
as
 declare @result int
 select @result = count(*) from tblBooks where booktitle like '%' + @filter + '%'
 return @result
go
```

Stored procedure usp_GetTitles() accepts a string filter and returns a SQL rowset of the rows that have the filter contained in the booktitle column. Stored procedure usp_CountTitles() is similar except that it just returns the number of rows in the rowset rather than the rowset itself.

The Web service under test is named BookSearch. The service has two Web methods. The first method is named GetTitles() and is defined as

```
[WebMethod]
public DataSet GetTitles(string filter)
{
  try
  {
    string connStr =
"Server=(local);Database=dbBooks;UID=webServiceLogin;PWD=secret";
    SqlConnection sc = new SqlConnection(connStr);
    SqlCommand cmd = new SqlCommand("usp_GetTitles", sc);
    cmd.CommandType = CommandType.StoredProcedure;
```

```
    cmd.Parameters.Add("@filter", SqlDbType.VarChar, 50);
    cmd.Parameters["@filter"].Direction = ParameterDirection.Input;
    cmd.Parameters["@filter"].Value = filter;
    SqlDataAdapter sda = new SqlDataAdapter(cmd);
    DataSet ds = new DataSet();
    sda.Fill(ds);
    sc.Close();
    return ds;
  }
  catch
  {
    return null;
  }
} // GetTitles
```

The GetTitles() method calls the usp_GetTitles() stored procedure to populate a DataSet object, which is returned by the method. Similarly, the CountTitles() Web method calls the usp_CountTitles() stored procedure:

```
[WebMethod]
public int CountTitles(string filter)
{
  try
  {
    string connString =
 "Server=(local);Database=dbBooks;UID=webServiceLogin;PWD=secret";
    SqlConnection sc = new SqlConnection(connString);
    SqlCommand cmd = new SqlCommand("usp_CountTitles", sc);
    cmd.CommandType = CommandType.StoredProcedure;
    SqlParameter p1 = cmd.Parameters.Add("ret_val", SqlDbType.Int, 4);
    p1.Direction = ParameterDirection.ReturnValue;
    SqlParameter p2 = cmd.Parameters.Add("@filter", SqlDbType.VarChar, 50);
    p2.Direction = ParameterDirection.Input;
    p2.Value = filter;
    sc.Open();
    cmd.ExecuteNonQuery();
    int result = (int)cmd.Parameters["ret_val"].Value;
    sc.Close();
    return result;
  }
  catch
  {
    return -1;
  }
} // CountTitles()
```

Except for the [WebMethod] attribute, nothing distinguishes these Web methods from ordinary methods; the .NET environment takes care of all the details for you. These are the two methods we want to test. Now, although not absolutely necessary to write test automation code, it helps to see the key code from the Web application that calls the Web service:

```
private void Button1_Click(object sender, System.EventArgs e)
{
  try
  {
    TheWebReference.BookSearch bs = new TheWebReference.BookSearch();
    string filter = TextBox1.Text.Trim();
    DataSet ds = bs.GetTitles(filter);
    DataGrid1.DataSource = ds;
    DataGrid1.DataBind();
    Label3.Text = "Found " + ds.Tables["Table"].Rows.Count + " items";
  }
  catch(Exception ex)
  {
    Label3.Text = ex.Message;
  }
}
```

This code illustrates the proxy mechanism. Calling a Web method of a Web service follows the same pattern as calling an ordinary method. When you test the Web service using a proxy mechanism, the test automation code will look very much like the preceding application code.

When a Web service accesses a database using stored procedures, the stored procedures are parts of the SUT. Techniques to test stored procedure are presented in Chapter 9. The techniques in this chapter demonstrate how to call and test a Web method with a single test case. To construct a complete test harness, you can use one of the harness patterns described in Chapter 4. The complete test harness that produced the test run shown in Figure 8-2 is presented in Section 8.7.

8.1 Testing a Web Method Using the Proxy Mechanism

Problem

You want to test a Web method in a Web service by calling the method using the proxy mechanism.

Design

Using Visual Studio .NET, add a Web Reference to your test automation harness that points to the Web service under test. This creates a proxy for the Web service that gives the Web service the appearance of being a local class. You can then instantiate an object that represents the Web service, and call the Web methods belonging to the service.

Solution

```
try
{
  string input = "the";
  int expectedCount = 1;
  TheWebReference.BookSearch bs = new TheWebReference.BookSearch();
  DataSet ds = new DataSet();

  Console.WriteLine("Calling Web Method GetTitles() with 'the'");
  ds = bs.GetTitles(input);
  if (ds == null)
    Console.WriteLine("Web Method GetTitles() returned null");
  else
  {
    int actualCount = ds.Tables["Table"].Rows.Count;
    Console.WriteLine("Web Method GetTitles() returned " + actualCount + " rows");
    if (actualCount == expectedCount)
      Console.WriteLine("Pass");
    else
      Console.WriteLine("*FAIL*");
  }

  Console.WriteLine("Done");
  Console.ReadLine();
}
catch(Exception ex)
{
  Console.WriteLine("Fatal error: " + ex.Message);
  Console.ReadLine();
}
```

This code assumes there is a Web service named `BookSearch` that contains a Web method named `GetTitles()`. The `GetTitles()` method accepts a target string as an input parameter and returns a `DataSet` object containing book information (ID, title, price) of the books that have the target string in their titles. When the Web Reference was added to the harness code, the reference name was changed from the default `localhost` to the slightly more descriptive `TheWebReference`. This name is then used as a namespace alias. The Web service name, `BookSearch`, acts as a proxy and is instantiated just as any local class would be, so you can call the `GetTitles()` method like an ordinary instance method. Notice that the fact that `GetTitles()` is a Web method rather than a regular method is almost completely transparent to the calling program.

Comments

Of the four main ways to test an ASP.NET Web service (by proxy mechanism, HTTP, TCP, sockets), using the Visual Studio proxy mechanism is by far the simplest. You call the Web method under test just as an application would. This situation is analogous to API testing where your test harness calls the API method under test just like an application would. Using the proxy mechanism is the most basic way to call a Web service and should always be a part of your test automation effort.

In this example, determining the correct return value from the `GetTitles()` method is more difficult than calling the method. Because `GetTitles()` returns a `DataSet` object, a complete expected value would be another `DataSet` object. In cases where the Web method under test returns a scalar value, such as a single `int` value for example, determining a pass/fail result is easy. For example, to test the `CountTitles()` method:

```
Console.WriteLine("Testing CountTitles() via poxy mechanism");
TheWebReference.BookSearch bs = new TheWebReference.BookSearch();
string input = "testing";
int expected = 3;

int actual = bs.CountTitles(input);

if (actual == expected)
  Console.WriteLine("Pass");
else
  Console.WriteLine("*FAIL*");
```

In the preceding solution, after calling `GetTitles()`, you compare the actual number of rows in the returned `DataSet` object with an expected number of rows. But this only checks for the correct number of rows and does not check whether the correct row data has been returned. Additional techniques to deal with complex return types, such as `DataSet` objects, are presented in Chapter 11.

8.2 Testing a Web Method Using Sockets

Problem

You want to test a Web method in a Web service by calling the method using sockets.

Design

First, construct a SOAP message to send to the Web method. Second, instantiate a `Socket` object and connect to the remote server that hosts the Web service. Third, construct a header that contains HTTP information. Fourth, send the header plus SOAP message using the `Socket.Send()` method. Fifth, receive the SOAP response using `Socket.Receive()` in a `while` loop. Sixth, analyze the SOAP response for an expected value(s).

Solution

Here is an example that sends the string "testing" to Web method `GetTitles()` and checks the response:

```
Console.WriteLine("Calling Web Method GetTitles() using sockets");
string input = "testing";

string soapMessage = "<?xml version=\"1.0\" encoding=\"utf-8\"?>";
soapMessage += "<soap:Envelope xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-
               instance\"";
soapMessage += " xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"";
soapMessage += " xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">";
soapMessage += "<soap:Body>";
soapMessage += "<GetTitles xmlns=\"http://tempuri.org/\">";
soapMessage += "<filter>" + input + "</filter>";
soapMessage += "</GetTitles>";
soapMessage += "</soap:Body>";
soapMessage += "</soap:Envelope>";

Console.WriteLine("SOAP message is: \n");
Console.WriteLine(soapMessage);

string host = "localhost";
string webService = "/TestAuto/Ch8/TheWebService/BookSearch.asmx";
string webMethod = "GetTitles";

IPHostEntry iphe = Dns.Resolve(host);
IPAddress[] addList = iphe.AddressList; // addList[0] == 127.0.0.1
EndPoint ep = new IPEndPoint(addList[0], 80); // ep = 127.0.0.1:80
Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
                           ProtocolType.Tcp);
socket.Connect(ep);
if (socket.Connected)
  Console.WriteLine("\nConnected to " + ep.ToString());
else
  Console.WriteLine("\nError: socket not connected");

string header = "POST " + webService + " HTTP/1.1\r\n";
header += "Host: " + host + "\r\n";
header += "Content-Type: text/xml; charset=utf-8\r\n";
header += "Content-Length: " + soapMessage.Length.ToString() + "\r\n";
header += "Connection: close\r\n";
header += "SOAPAction: \"http://tempuri.org/" + webMethod + "\"\r\n\r\n";
Console.Write("Header is: \n" + header);
```

```
string sendAsString = header + soapMessage;
byte[] sendAsBytes = Encoding.ASCII.GetBytes(sendAsString);
int numBytesSent = socket.Send(sendAsBytes, sendAsBytes.Length,
                               SocketFlags.None);
Console.WriteLine("Sending = " + numBytesSent + " bytes\n");

byte[] receiveBufferAsBytes = new byte[512];
string receiveAsString = "";
string entireReceive = "";
int numBytesReceived = 0;

while ((numBytesReceived = socket.Receive(receiveBufferAsBytes, 512,
        SocketFlags.None)) > 0 )
{
  receiveAsString = Encoding.ASCII.GetString(receiveBufferAsBytes, 0,
                                             numBytesReceived);
  entireReceive += receiveAsString;
}

Console.WriteLine("\nThe SOAP response is " + entireReceive);

Console.WriteLine("\nDetermining pass/fail");

if ( entireReceive.IndexOf("002") >= 0 &&
     entireReceive.IndexOf("004") >= 0 &&
     entireReceive.IndexOf("005") >= 0 )
  Console.WriteLine("\nPass");
else
  Console.WriteLine("\nFail");
```

Each of the six steps when using sockets to call a Web method could be considered a separate problem-solution, but because the steps are so completely interrelated, it's easier to understand them when presented together.

Comments

Of the four main ways to test an ASP.NET Web service (by proxy mechanism, HTTP, TCP, sockets), using sockets operates at the lowest level of abstraction. This gives you the most flexibility but requires the most code.

The first step is to construct a SOAP message. You must construct the SOAP message before constructing the HTTP header string because the header string requires the length (in bytes) of the SOAP message. Constructing the appropriate SOAP message is easier than you might expect. You can get a template of the SOAP message from Visual Studio .NET by loading up the Web service as a project. Next you instruct Visual Studio to run the Web service by pressing F5. Because a Web service is a type of library and not an executable, the service cannot run. Instead, Visual Studio launches a utility application that gives you a template for the SOAP message to send. For example, instructing the `BookSearch` service to run and selecting the `GetTitles()` method produces a Web page that contains this template information:

```
The following is a sample SOAP request and response. The placeholders shown need
to be replaced with actual values.

POST /TestAuto/Ch8/TheWebService/BookSearch.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/GetTitles"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetTitles xmlns="http://tempuri.org/">
      <filter>string</filter>
    </GetTitles>
  </soap:Body>
</soap:Envelope>
```

The lower part of the template is the SOAP message. You can build up the message by appending short strings together as demonstrated in the preceding solution, or you can simply assign the entire SOAP message as one long string. Notice that SOAP is nothing more than a particular type of XML. In XML, you can use either single quotes or double quotes, so replacing the double quote characters in the template with single quote characters often improves readability. If you want to retain the double quote characters, be sure to escape them using the \" sequence as demonstrated in the "Solution" part of this technique. In the preceding template, `<filter>` corresponds to the input parameter for method `GetTitles()`, and the string placeholder represents the value of the parameter. You have to be careful when constructing the SOAP message because the syntax is very brittle, meaning that it usually only takes one wrong character in the message (a missing blank space for example) to generate a general internal server error message.

The second step when calling a Web method using sockets is to instantiate a `Socket` object and connect to the remote server that hosts the Web service. The `Socket` class is housed in the `System.Net.Sockets` namespace. You can instantiate a `Socket` object with a single statement:

```
Socket socket = new Socket(AddressFamily.InterNetwork,
                           SocketType.Stream, ProtocolType.Tcp);
```

`Socket` objects implement the Berkeley sockets interface, which is an abstraction mechanism for sending and receiving network data. The first argument to the `Socket()` constructor specifies the addressing scheme that the instance of the `Socket` class will use. The `InterNetwork` value specifies ordinary IP version 4. Some of the other schemes include the following:

- `InterNetworkV6`: Address for IP version 6.
- `NetBios`: NetBios address.
- `Unix`: Unix local to host address.
- `Sna`: IBM SNA (Systems Network Architecture) address.

The second argument to the Socket() constructor specifies one of six types of sockets you can create. Using SocketType.Stream means the socket is a TCP socket. The other common socket type is Dgram, which is used for UDP (User Datagram Protocol) sockets. The third argument to the Socket constructor represents the network protocol that the Socket object will use for communication. The type ProtocolType.Tcp is most common but others such as ProtocolType.Udp are available too. The great flexibility you have when instantiating a Socket object points out some of the situations in which you want to call a Web method using sockets instead of the much easier proxy mechanism—for example, using sockets lets you call a UDP-specific Web service. After you have created a Socket object, you can connect to the server that houses the Web service:

```
IPHostEntry iphe = Dns.Resolve(host);
IPAddress[] addList = iphe.AddressList; // addList[0] == 127.0.0.1
EndPoint ep = new IPEndPoint(addList[0], 80); // ep = 127.0.0.1:80

socket.Connect(ep);
if (socket.Connected)
  Console.WriteLine("\nConnected to " + ep.ToString());
else
  Console.WriteLine("\nError: socket not connected");
```

The Socket.Connect() method accepts an EndPoint object. You can think of an EndPoint as an IP address plus a port number. You can specify the IP address directly like this:

```
IPAddress ipa = IPAddress.Parse("127.0.0.1");
```

You can also get the IP address by calling the Dns.Resolve() method that returns a list of IP addresses that map to the host input argument. After you have the IP address, you can create an EndPoint object and then pass that object to the Socket.Connect() method.

The third step when calling a Web method using sockets is to construct a header that contains HTTP information. You can get this information from the Visual Studio-generated template in the first step described earlier in this "Comments" section. You have to be careful with two minor issues, however. The first issue is that HTTP headers are terminated by a \r\n sequence instead of the \n sequence. Additionally the last line of the HTTP header is indicated by a double set of \r\n sequences. The second issue is the SOAPAction header. This header essentially tells the Web server that the HTTP request has SOAP content. This header has been deprecated in the SOAP 1.2 specification in favor of a new "Content-Type:application/soap+xml" header, but for now most servers are expecting a SOAPAction header.

The fourth step when calling a Web method using sockets is to send the header plus SOAP message using the Socket.Send() method:

```
string sendAsString = header + soapMessage;
byte[] sendAsBytes = Encoding.ASCII.GetBytes(sendAsString);
int numBytesSent = socket.Send(sendAsBytes, sendAsBytes.Length,
                               SocketFlags.None);
Console.WriteLine("Sending header + body = " + numBytesSent + " bytes\n");
```

This part of the process is straightforward. You take the data to send as a string, convert to a byte array using the GetBytes() method (located in the System.Text namespace), and then call Socket.Send() passing in the byte array, its length, and a SocketFlags value. The SocketFlags enumeration is rarely needed. For example, `SocketFlags.DontRoute` means to send the bytes without using any routing tables. The `Socket.Send()` method returns the number of bytes sent, which is useful for diagnosing troubles with your test automation. The `Socket.Send()` method and its counterpart `Socket.Receive()` are synchronous methods. You can send and receive asynchronously using `Socket.BeginSend()` and `Socket.BeginReceive()`.

The fifth step when calling a Web method using sockets is to receive the SOAP response using `Socket.Receive()` inside a `while` loop:

```
byte[] receiveBufferAsBytes = new byte[512];
string receiveAsString = "";
string entireReceive = "";
int numBytesReceived = 0;

while ((numBytesReceived = socket.Receive(receiveBufferAsBytes, 512,
        SocketFlags.None)) > 0 )
{
  receiveAsString = Encoding.ASCII.GetString(receiveBufferAsBytes, 0,
                                             numBytesReceived);
  entireReceive += receiveAsString;
}
```

This code snippet uses a classic stream-reading technique. You declare a byte array buffer to hold chunks of the return data—there is no way to predict how big the return will be. The size of 512 bytes used here is arbitrary. The `Socket.Receive()` method reads bytes from the associated socket, stores those bytes, and returns the actual number of bytes read. If the number of bytes read is 0, then the return bytes have been used up, and you exit the loop. After each block of 512 bytes is received, it is stored as an ASCII string using the `GetString()` method. A second string accumulates the entire received data by appending as each new block arrives.

At this point, you have the entire SOAP response stored into a string variable. If you are just calling a Web method using sockets, then you are done. But if you are testing the Web method, then you must perform the sixth step in the process, which is to examine the received data for an expected value. This is not so easy to do. In the preceding solution, you check the received string for the presence of three substrings, which are the IDs of the three expected books that have "testing" in their titles:

```
if ( entireReceive.IndexOf("002") >= 0 &&
     entireReceive.IndexOf("004") >= 0 &&
     entireReceive.IndexOf("005") >= 0 )
  Console.WriteLine("\nPass");
else
  Console.WriteLine("\nFail");
```

This approach does not absolutely guarantee that the SOAP response is exactly correct. Because the actual return value when calling a Web method using sockets is a SOAP string, which in turn is a kind of XML, a complete expected value would be another SOAP/XML string. Comparing two XML fragments or documents is rather subtle and is discussed in Chapter 12.

This technique reads the entire SOAP response into a string variable. This string could be very long. An alternative approach is to process each 512-byte string as it arrives. However, this approach is tricky because the target string you are searching for could be chopped by the buffering process. For example, if you were searching for the string "002", the response could conceivably break in the middle of the string with "00" coming as the last two characters of one receive block and "2" coming as the first character of the next receive block.

8.3 Testing a Web Method Using HTTP

Problem

You want to test a Web method in a Web service by calling the method using HTTP.

Design

Create an HTTPWebRequest object that points to the Web method, use the GetResponse() method to send name-value pairs that correspond to parameter-argument pairs, and then fetch the response using the GetResponseStream() method.

Solution

This example sends the string "testing" to Web method GetTitles():

```
Console.WriteLine("Calling Web Method GetTitles() using HTTP");

string input = "testing";
string postData = "filter=" + input;
byte[] buffer = Encoding.ASCII.GetBytes(postData);
string uri =
 "http://localhost/TestAuto/Ch8/TheWebService/BookSearch.asmx/GetTitles";

HttpWebRequest req = (HttpWebRequest)WebRequest.Create(uri);
req.Method = "POST";
req.ContentType = "application/x-www-form-urlencoded";
req.ContentLength = buffer.Length;
req.Headers.Add("SOAPAction: \"http://tempuri.org/GetTitles\"");
req.Timeout = 5000;

Stream reqst = req.GetRequestStream(); // add form data to request stream
reqst.Write(buffer, 0, buffer.Length);

reqst.Flush();
reqst.Close();

HttpWebResponse res = (HttpWebResponse)req.GetResponse();
Stream resst = res.GetResponseStream();
StreamReader sr = new StreamReader(resst);

string response = sr.ReadToEnd();

Console.WriteLine("HTTP response is " + response);
```

```
Console.WriteLine("\nDetermining pass/fail");

if ( response.IndexOf("002") >= 0 &&
     response.IndexOf("004") >= 0 &&
     response.IndexOf("005") >= 0 )
  Console.WriteLine("\nPass");
else
  Console.WriteLine("\nFail");

sr.Close();
resst.Close();
```

Because ASP.NET Web services operate over HTTP, you can use the `HttpWebRequest` class to post data directly to Web methods. Web methods expect data in name-value pairs such as

```
filter=testing
```

where the name part is the Web method parameter name, and the value part is the parameter value. The `HttpWebRequest.GetResponse()` method returns an `HttpWebResponse` object, which, in turn, has a `GetResponseStream()` method that can be used to read the response as string data.

Comments

Of the four main ways to test an ASP.NET Web service, using HTTP operates at the middle level of abstraction. The technique provides a nice compromise between simplicity and flexibility. Just as you can generate a SOAP request template as described in Section 8.2, you can also generate an HTTP request template by instructing Visual Studio to run the Web service.

Depending on the particular configuration of the server that hosts the Web service, you may or may not need to add the special `SOAPAction` header:

```
req.Headers.Add("SOAPAction: \"http://tempuri.org/GetTitles\"");
```

In practical terms, it's often easier to first try the request with the header because unrecognized headers are typically ignored by the server. The pattern to post HTTP data is fairly simple: first create a string of name-value pairs that correspond to the Web method's parameter-argument pairs, and then convert the post string to bytes. Next, create an `HttpWebRequest` object using the factory mechanism with the `WebRequest.Create()` method (as opposed to instantiation using the `new` keyword), and then specify values for the `Method`, `ContentType`, and `ContentLength` properties. Send the HTTP request by writing the `POST` data into a `Stream` object (as opposed to using an explicit `Write()` method of some sort), and fetch the response using a `StreamReader` object. The process is best understood by examining concrete examples like the solution in this section, rather than by general principles.

As discussed in Section 8.2, determining a pass/fail result is harder than calling the Web method under test. One good strategy is to structure your underlying database test bed as much as possible so that the data is easily and uniquely identifiable. This is not always feasible, however, and, in such situations, you must sometimes simply rely on manual testing to supplement your test automation. The essence of calling a Web method using HTTP is programmatically posting data to a Web server; see Chapter 5 for additional techniques.

8.4 Testing a Web Method Using TCP

Problem

You want to test a Web method in a Web service by calling the method using TCP.

Design

First, instantiate a `TcpClient` object and connect to the remote server that hosts the Web service. Second, construct a SOAP message to send to the Web method. Third, construct a header that contains HTTP information. Fourth, instantiate a `NetworkStream` object associated with the `TcpClient` object and send the header plus SOAP message using the `NetworkStream.Write()` method. Fifth, receive the SOAP response using a `NetworkStream.Read()` method in a `while` loop. Sixth, analyze the SOAP response for an expected value(s).

Solution

This example sends the string "`testing`" to Web method `GetTitles()` via TCP:

```
Console.WriteLine("Calling Web Method GetTitles() using TCP");
string input = "testing";

//TcpClient client = new TcpClient("127.0.0.1", 80);
TcpClient client = new TcpClient(AddressFamily.InterNetwork);
client.Connect("127.0.0.1", 80);

string soapMessage = "<?xml version=\"1.0\" encoding=\"utf-8\"?>";
soapMessage += "<soap:Envelope xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-
 instance\"";
soapMessage += " xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"";
soapMessage += " xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">";
soapMessage += "<soap:Body>";
soapMessage += "<GetTitles xmlns=\"http://tempuri.org/\">";
soapMessage += "<filter>" + input + "</filter>";
soapMessage += "</GetTitles>";
soapMessage += "</soap:Body>";
soapMessage += "</soap:Envelope>";

string webService = "/TestAuto/Ch8/TheWebService/BookSearch.asmx";
string host = "localhost";
string webMethod = "GetTitles";

string header = "POST " + webService + " HTTP/1.1\r\n";
header += "Host: " + host + "\r\n";
header += "Content-Type: text/xml; charset=utf-8\r\n";
header += "Content-Length: " + soapMessage.Length.ToString() + "\r\n";
header += "Connection: close\r\n";
header += "SOAPAction: \"http://tempuri.org/" + webMethod + "\"\r\n\r\n";
```

```
Console.Write("Header is: \n" + header);

string requestAsString = header + soapMessage;
byte[] requestAsBytes = Encoding.ASCII.GetBytes(requestAsString);

NetworkStream stream = client.GetStream();
stream.Write(requestAsBytes, 0, requestAsBytes.Length);

byte[] responseBufferAsBytes = new byte[512];
string responseAsString = "";
string entireResponse = "";
int numBytesReceived = 0;

while ((numBytesReceived = stream.Read(responseBufferAsBytes, 0, 512)) > 0)
{
  responseAsString = Encoding.ASCII.GetString(responseBufferAsBytes, 0,
                                              numBytesReceived);
  entireResponse += responseAsString;
}

Console.WriteLine(entireResponse);

if ( entireResponse.IndexOf("002") >= 0 &&
     entireResponse.IndexOf("004") >= 0 &&
     entireResponse.IndexOf("005") >= 0 )
  Console.WriteLine("\nPass");
else
  Console.WriteLine("\nFail");
```

Calling a Web method using TCP is very similar to calling a Web method using sockets. This makes sense because the `TcpClient` class was designed to act as a friendly wrapper around the `Socket` class. The two techniques are so similar that it's difficult to choose between them. There are two ways to view the question of which technique to use. One argument is that because using `TcpClient` is slightly cleaner than using the `Socket` class, you should use `TcpClient` when calling a Web service over TCP, and you should use `Socket` only when calling a Web service that is implemented using a non-TCP protocol. In practical terms, because ASP.NET Web services use HTTP, which, in turn, uses TCP, the first argument simplifies to "always use the `TcpClient` class to call Web methods at a low level." The second argument about which technique to use is that because using `TcpClient` and `Socket` are so similar, you might just as well use the `Socket` class because it's more flexible. So the second argument simplifies to "always use the `Socket` class to call Web methods at a low level." Ultimately having two different methods in your skill set is better than having just one to choose from. However, because using `TcpClient` is so similar to using the `Socket` class, when you are testing a Web method by calling in two different ways as a means to validate your test automation, you should use one or the other technique but not both.

Comments

Of the four main ways to test an ASP.NET Web service, using TCP operates at a low level of abstraction, one just barely above using sockets. There are six discrete steps to perform when testing a Web method using the `TcpClient` class. These six steps could be considered separate problem/solution pairs but because the steps are so dependent on each other, it's easier to understand them when presented together. The first step is to instantiate a `TcpClient` object and connect to the server that hosts the Web service under test:

```
TcpClient client = new TcpClient(AddressFamily.InterNetwork);
client.Connect("127.0.0.1", 80);
```

See Section 8.2 for a discussion of the `AddressFamily` enumeration. After the `TcpClient` object has been created, you can connect by passing the server IP address and port number. A minor variation you can employ is to pass the sever IP address and port number to an overloaded version of the constructor and omit the `AddressFamily` specification because `InterNetwork` is the default value for the `Connect()` method:

```
TcpClient client = new TcpClient("127.0.0.1", 80);
```

The second step to call a Web method using TCP is to construct a SOAP message to send to the Web method:

```
string soapMessage = "<?xml version=\"1.0\" encoding=\"utf-8\"?>";
soapMessage += "<soap:Envelope xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-
 instance\"";
soapMessage += " xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"";
soapMessage += " xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">";
soapMessage += "<soap:Body>";
soapMessage += "<GetTitles xmlns=\"http://tempuri.org/\">";
soapMessage += "<filter>" + input + "</filter>";
soapMessage += "</GetTitles>";
soapMessage += "</soap:Body>";
soapMessage += "</soap:Envelope>";
```

As described in Section 8.2, you can get a SOAP message template from Visual Studio. The third step is to construct a header that contains HTTP information:

```
string webService = "/TestAuto/Ch8/TheWebService/BookSearch.asmx";
string host = "localhost";
string webMethod = "GetTitles";

string header = "POST " + webService + " HTTP/1.1\r\n";
header += "Host: " + host + "\r\n";
header += "Content-Type: text/xml; charset=utf-8\r\n";
header += "Content-Length: " + soapMessage.Length.ToString() + "\r\n";
header += "Connection: close\r\n";
header += "SOAPAction: \"http://tempuri.org/" + webMethod + "\"\r\n\r\n";
Console.Write("Header is: \n" + header);
```

Again, you can get this information from Visual Studio. The fourth step is to instantiate a NetworkStream object associated with the TcpClient object and send the header plus SOAP message using the NetworkStream.Write() method:

```
string requestAsString = header + soapMessage;
byte[] requestAsBytes = Encoding.ASCII.GetBytes(requestAsString);

NetworkStream stream = client.GetStream();
stream.Write(requestAsBytes, 0, requestAsBytes.Length);
```

This step differs most from using the Socket class. After converting the HTTP header and SOAP message into a byte array using the GetBytes() method, you create a NetworkStream object using TcpClient.GetStream() and then send the data over the network using NetworkStream.Write()—very clean and easy. The fifth step to call a Web method using TCP is to receive the SOAP response using a NetworkStream.Read() method inside a while loop:

```
byte[] responseBufferAsBytes = new byte[512];
string responseAsString = "";
string entireResponse = "";
int numBytesReceived = 0;

while ((numBytesReceived = stream.Read(responseBufferAsBytes, 0, 512)) > 0)
{
  responseAsString = Encoding.ASCII.GetString(responseBufferAsBytes, 0,
                                              numBytesReceived);
  entireResponse += responseAsString;
}
```

This step follows almost the same buffered reading pattern as the one described in Section 8.2. You may find it instructive to compare the two code blocks side by side. After you understand the general pattern, you'll find it useful in a wide range of test automation and development scenarios. The sixth and final step to test a Web method using TCP is to examine the SOAP response for some sort of an expected value:

```
Console.WriteLine(entireResponse);

if ( entireResponse.IndexOf("002") >= 0 &&
     entireResponse.IndexOf("004") >= 0 &&
     entireResponse.IndexOf("005") >= 0 )
  Console.WriteLine("\nPass");
else
  Console.WriteLine("\nFail");
```

As discussed in Sections 8.2 and 8.3, determining a pass or fail result is not easy when the return value is as complex as it is here.

8.5 Using an In-Memory Test Case Data Store

Problem

You want to use an in-memory test case data store rather than use external storage such as a text file or SQL table.

Design

Create a class-scope `ArrayList` object and use the `Add()` method to insert test case data. Iterate through the `ArrayList` object using either a `foreach` or `for` loop.

Solution

```
class Class1
{
  static ArrayList testcases = new ArrayList();

  static void Main(string[] args)
  {
    try
    {
      Console.WriteLine("\nBegin test run\n");

      testcases.Add("001:GetTitles:testing:3:Theory");
      testcases.Add("002:GetTitles:and:1:Theory");
      testcases.Add("003:GetTitles:better:1:Build");
      testcases.Add("004:GetTitles:Algorithms:1:Algorithms");

      foreach (string testcase in testcases) // main test loop
      {
        string[] tokens = testcase.Split(':');
        string id = tokens[0];
        string method = tokens[1];
        string input = tokens[2];
        int expectedCount = int.Parse(tokens[3]);
        string hint = tokens[4];

        // call method under test here
        // compare actual result to expected result here
        // display or store test result here
      }
    }
    catch(Exception ex)
    {
      Console.WriteLine("Fatal error: " + ex.Message);
    }
  }
}
```

As a general rule of thumb, in lightweight test automation situations, external test case storage (in the form of text files, XML files, SQL databases, and so on) is preferable to internal storage. External test case data can be shared among different test harnesses and is easier to edit. But using an in-memory test case data store has certain advantages; keeping the test case data embedded within the test harness makes maintenance somewhat easier. The simplest approach to in-memory test case storage is to use an `ArrayList` object. In-memory test case storage is particularly appropriate when your number of test cases is relatively small (generally, under 100) or if you want to distribute the harness as a single, stand-alone executable to several people for use as DRT (Developer Regression Test) or BVT (Build Verification Test) regression purposes.

Comments

One alternative to using an `ArrayList` object for in-memory test case storage is to use an `array` object. Using an array, the preceding solution becomes

```
class Class1
{
  static string[] testcases =
    new string[] { "001:GetTitles:testing:3:Theory",
                   "002:GetTitles:and:1:Theory",
                   "003:GetTitles:better:1:Build"
                 };

  static void Main(string[] args)
  {
    try
    {
      Console.WriteLine("\nBegin test run\n");

      for (int i = 0; i < testcases.Length; ++i) // main test loop
      {
        string[] tokens = testcases[i].Split(':');
        string id = tokens[0];
        string method = tokens[1];
        string input = tokens[2];
        int expectedCount = int.Parse(tokens[3]);
        string hint = tokens[4];

        // call method under test here
        // compare actual result to expected result here
        // display or store test result here
      }
    }
    catch(Exception ex)
    {
      Console.WriteLine("Fatal error: " + ex.Message);
    }
  }
}
```

This approach has an older, pre-.NET feel but otherwise is virtually equivalent to using an `ArrayList` object. In theory, using an `array` object provides better performance than using an `ArrayList`, but this would only be a factor with a very large number of test cases, which means using an in-memory data store is probably not a good idea anyway.

A second minor variation to using an `ArrayList` object for in-memory test case data storage is to use a `Queue` object. The solution using a `Queue` looks like this:

```
class Class1
{
  static Queue testcases = new Queue();

  static void Main(string[] args)
  {
    try
    {
      testcases.Enqueue("001:GetTitles:testing:3:Theory");
      testcases.Enqueue("002:GetTitles:and:1:Theory");
      testcases.Enqueue("003:GetTitles:better:1:Build");

      while (testcases.Count > 0)
      {
        string testcase = (string)testcases.Dequeue();
        string[] tokens = testcase.Split(':');
        string id = tokens[0];
        string method = tokens[1];
        string input = tokens[2];
        int expectedCount = int.Parse(tokens[3]);
        string hint = tokens[4];

        // call method under test here
        // compare actual result to expected result here
        // display or store test result here
      }
    }
    catch(Exception ex)
    {
      Console.WriteLine("Fatal error: " + ex.Message);
    }
  }
}
```

There are few technical reasons to choose one of these data store objects (`ArrayList`, `array`, `Queue`) over another. Using an `ArrayList` or `array` object allows random access to any test case because you can fetch a particular test case by index value. Using an `ArrayList` or `Queue` object gives you the possibility of adding test case data programmatically via the `ArrayList.Add()` or `Queue.Enqueue()` methods. However, your choice will most often be based on personal coding style preference.

8.6 Working with an In-Memory Test Results Data Store

Problem

You want to save your test results to an in-memory data store in such a way that you can easily determine if a particular test case passed or not before running a new test case.

Design

If your test cases have dependencies, where running one case depends on the result of a previous case, then consider storing test results into a `Hashtable` object. For general processing of test results, using an `ArrayList` object is usually the best choice.

Solution

To insert a test result into a `Hashtable`, use

```
TestResult tr = null;
if (actualResult == expectedResult)
{
  tr = new TestResult(id, "Pass");
  testResults.Add(id, tr);
}
else
{
  tr = new TestResult(id, "FAIL");
  testResults.Add(id, tr);
}
```

where

```
static Hashtable testResults = new Hashtable();

class TestResult
{
  public string id;
  public string pf; // "pass" or "fail"
  public TestResult(string id, string pf)
  {
    this.id = id; this.pf = pf;
  }
}
```

The result of each test case is stored into a `Hashtable`. Now suppose that all test cases are dependent upon test case `002` passing. You can write code like this snippet:

```
string mustPass = "002";

TestResult tr = testResults[mustPass] as TestResult;

if (tr.pf == "Pass")
{
  Console.WriteLine("The dependency passed so I can run case " + id);
  // run test and store result
}
else
{
  Console.WriteLine("The dependency failed so I will skip case " + id);
  continue;
}
```

You will sometimes have test cases that have dependencies on other test cases, meaning whether or not a test case runs is contingent on whether a previous test case passes (or fails). In situations like this, you should store test results to an in-memory data store. This data store must be searched before executing each test case. Even for a moderate number of test cases, you need a data structure that can be searched as quickly as possible. The `Hashtable` object is designed for just such situations.

Comments

Working with test case data that has dependencies on the results of other test cases is simple in principle but can be tricky in practice. A `Hashtable` object accepts a key, which you can think of as an ID, and an object to store. For test case results, the test case ID is a natural choice as a key. Because a `Hashtable` stores objects, a good design approach is to create a lightweight class to hold test result information so you can call the `Hashtable.Add()` method passing in a `TestResult` object as the value to be added.

If the number of test case dependencies is very small, you can hard-code the dependency logic into your test automation harness. If the number of dependencies is large, however, you should first rethink your entire test harness design and see if you can simplify. Then, if the dependencies are unavoidable, you'll want to store the dependencies as part of the test case input data. This pushes you to store test case data in a lightweight class such as

```
class TestCase
{
  public string id;
  public string input;
  public string expected;
  public ArrayList dependencies;

  public TestCase(string id, string input, string expected,
                  ArrayList dependencies)
  {
    // constructor code here
  }
}
```

Then you can structure your test harness (in pseudo-code) like this:

```
loop thru test case collection
  fetch a TestCase object
  bool shouldRun = true;
  loop thru each dependency
  {
    pull dependency result from Hashtable
    if (dependency case failed)
      shouldRun = false;
      break;
  }

  if (shouldRun == true)
  {
    run test;
    store test result;
  }
  else
  {
    skip test;
  }
end loop
```

When using a `Hashtable` object for an in-memory test results data store, at some point you have to either display the results or save them to external storage. There are several ways to do this. One approach is to maintain two different data structures—one `Hashtable` to determine test case dependencies and one `ArrayList` to hold test case results for display/external storage. This is simple but inefficient. A second approach is to keep test results in a `Hashtable` and then after the main test-processing loop has finished, traverse the `Hashtable` and save to external storage. For example:

```
Hashtable testResults = new Hashtable();

// run harness, store all results into testResults

FileStream fs = new FileStream("TestResults.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);

foreach (DictionaryEntry de in testResults)
{
  TestResult t = de.Value as TestResult;
  sw.WriteLine(t.id + " " + t.pf);
}
sw.Close();
fs.Close();
```

Because each element in a `Hashtable` object is a `DictionaryEntry` object, you can iterate through a `Hashtable` in this nonobvious way.

8.7 Example Program: WebServiceTest

This program combines several of the techniques in this chapter to create a lightweight test automation harness to test a Web service (see Listing 8-1). When run, the output will be as shown in Figure 8-2 in the introduction to this chapter. The test harness uses an in-memory test case data store. Test case `002` has a deliberate error to demonstrate the concept of validation. The expected count for case `002` should be `2`, not `1` as coded. Each test case is used to call the Web method under test, `GetTitles()`, twice. The first call is via the simple proxy mechanism. The second test call is via the low-level TCP technique.

Listing 8-1. *Program WebServiceTest*

```
using System;
using System.Collections;
using System.Data;
using System.Net.Sockets;
using System.Text;

namespace RunTests
{
  class Class1
  {
    static ArrayList testcases = new ArrayList();

    [STAThread]
    static void Main(string[] args)
    {
      try
      {
        Console.WriteLine("\nBegin BookSearch Web service test run\n");

        testcases.Add("001:GetTitles:testing:3:Theory");
        testcases.Add("002:GetTitles:and:1:Theory"); // error
        testcases.Add("003:GetTitles:better:1:Build");
        // other test cases go here

        foreach (string testcase in testcases)
        {
          string[] tokens = testcase.Split(':');
          string id = tokens[0];
          string method = tokens[1];
          string input = tokens[2];
          int expectedCount = int.Parse(tokens[3]);
          string hint = tokens[4];
```

```
        Console.WriteLine("=======================");
        Console.WriteLine("Case ID = " + id);
        Console.WriteLine("Sending input = '" + input + "' to Web
                      method GetTitles()");

        Console.WriteLine("\nTesting using proxy mechanism . . . ");
        BookReference.BookSearch bs = new BookReference.BookSearch();
        DataSet ds = bs.GetTitles(input);
        Console.WriteLine("Expected count = " + expectedCount);

        bool proxyPass;
        if (ds.Tables["Table"].Rows.Count == expectedCount &&
ds.Tables["Table"].Rows[0]["booktitle"].ToString().IndexOf(hint) >= 0)
          proxyPass = true;
        else
          proxyPass = false;
        Console.WriteLine("Pass via proxy = " + proxyPass);

        Console.WriteLine("\nTesing using TCP mechanism . . . ");
        TcpClient client = new TcpClient(AddressFamily.InterNetwork);
        client.Connect("127.0.0.1", 80);

        string soapMessage = "<?xml version=\"1.0\" encoding=\"utf-8\"?>";
        soapMessage +=
"<soap:Envelope xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"";
        soapMessage += " xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"";
        soapMessage +=
"xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">";
        soapMessage += "<soap:Body>";
        soapMessage += "<GetTitles xmlns=\"http://tempuri.org/\">";
        soapMessage += "<filter>" + input + "</filter>";
        soapMessage += "</GetTitles>";
        soapMessage += "</soap:Body>";
        soapMessage += "</soap:Envelope>";
        // Console.WriteLine("SOAP message is " + soapMessage);

        string webService = "/TestAuto/Ch8/TheWebService/BookSearch.asmx";
        string host = "localhost";
        string webMethod = "GetTitles";

        string header = "POST " + webService + " HTTP/1.1\r\n";
        header += "Host: " + host + "\r\n";
        header += "Content-Type: text/xml; charset=utf-8\r\n";
        header += "Content-Length: " + soapMessage.Length.ToString() + "\r\n";
        header += "Connection: close\r\n";
        header += "SOAPAction: \"http://tempuri.org/" + webMethod + "\"\r\n\r\n";
        //Console.Write("Header is: \n" + header);
```

```
          string requestAsString = header + soapMessage;
          byte[] requestAsBytes = Encoding.ASCII.GetBytes(requestAsString);

          NetworkStream ns = client.GetStream();
          ns.Write(requestAsBytes, 0, requestAsBytes.Length);

          byte[] responseBufferAsBytes = new byte[512];
          string responseAsString = "";
          string entireResponse = "";
          int numBytesReceived = 0;

          while ((numBytesReceived = ns.Read(responseBufferAsBytes, 0, 512)) > 0)
          {
            responseAsString = Encoding.ASCII.GetString(responseBufferAsBytes, 0,
                                                        numBytesReceived);
            entireResponse += responseAsString;
          }

          //Console.WriteLine(entireResponse);
          Console.WriteLine("Seeking '" + hint + "'");
          bool tcpPass;
          if (entireResponse.IndexOf(hint) >= 0)
            tcpPass = true;
          else
            tcpPass = false;
          Console.WriteLine("Pass via TCP = " + tcpPass);

          if (proxyPass == true && tcpPass == true)
            Console.WriteLine("\nPass");
          else
            Console.WriteLine("\n** FAIL or INCONSISTENT **");

        } // main test loop

        Console.WriteLine("=======================");
        Console.WriteLine("\nDone");
        Console.ReadLine();
      }
      catch(Exception ex)
      {
        Console.WriteLine("Fatal error: " + ex.Message);
        Console.ReadLine();
      }

    } // Main()
  } // Class1
} // ns
```