6

Web Services

When XML use began to take off just after the year 2000, businesses, developers, and others looked for new ways to use it. The promise of separating content and presentation had been met, but how could this capability be capitalized on? The answer came in the form of web services.

Web services provide a way to exchange data between applications and servers. To facilitate this communication, web services use the Internet to send messages composed of XML data back and forth between a *consumer* (the application that uses the data) and a *provider* (the server that contains the data). This is not unlike traditional distributed computing models, such as CORBA, DCOM, and RMI, where method calls are executed over a network connection. The major difference with web services is that the data being transmitted is XML text instead of a binary format.

The promise behind web services is that of having software components available, on demand, to any application in the world. Whether that be a web application or a traditional desktop application, it is possible to use the same service to perform the same task.

Related Technologies

Web services aren't a single technology or platform; in fact, they are a mixture of several protocols, languages, and formats. And although several different platforms have incorporated web services into their current offerings, there are still some basic parts that remain consistent.

SOAP

SOAP is a combination of an XML-based language and any number of common protocols for transmitting this data. The SOAP specification describes an intricate language with numerous elements and attributes, intended to describe most types of data. This information can be transported over any number of protocols, but is most commonly sent over HTTP along with other web traffic.

Originally an acronym for Simple Object Access Protocol, the specification now simply goes by SOAP.

There are two main ways of using SOAP, the *remote procedure call* (*RPC*) style and the *document* style.

RPC-Style SOAP

The RPC style of SOAP treats the web service as though it were an object containing one or more methods (in much the same way you would use a local class to establish communication with a database). A request is made to the service detailing the method name to call and the parameters, if any, to pass. The method is executed on the server, and an XML response is dispatched containing the return value, if any, or an error message if something went awry. Imagine a web service that provides simple arithmetic operations: addition, subtraction, multiplication, and division. Each method takes two numbers and returns a result. An RPC-style request for the add operation would look something like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <soap:Body>
    <w:add xmlns:w="http://www.wrox.com/services/math">
      <w:op1>4.5</w:op1>
      <w:op2>5.4</w:op2>
    </w:add>
  </soap:Body>
</soap:Envelope>
```

Whenever you are dealing with non-trivial XML documents, for example documents that are to be shared across businesses and applications, namespaces come into play. Namespaces are especially important in SOAP because these documents need to be produced and read by different systems. The SOAP namespace, specified in this example as `http://schemas.xmlsoap.org/soap/envelope/`, is for version 1.1, and can vary depending on which version you are using. The version 1.2 namespace is `http://www.w3.org/2003/05/soap-envelope`.

The `<w:add/>` element specifies the name of the method to call (`add`) and contains the other namespace in the example, `http://www.wrox.com/services/math`. This namespace is specific to the service that is being called and can be defined by the developer. The `soap:encodingStyle` attribute points to a URI indicating how the data is encoded in the request. There are a variety of other encoding styles available as well, such as the type system employed in XML schemas.

An optional `<soap:Header/>` element can be used to contain additional information, such as security credentials. If used, this element comes immediately before `<soap:Body/>`.

If the request to add two numbers executed successfully, the response message would look like this:

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

```
  <soap:Body>
    <w:addResponse xmlns:w="http://www.wrox.com/services/math">
      <w:addResult>9.9</w:addResult>
    </w:addResponse>
  </soap:Body>
</soap:Envelope>
```

As you can see, the format is similar to the initial request. The standard way of supplying the result is to create an element with the name of the method followed by the word "Response." In this case, the element is `<w:addResponse/>`, and has the same namespace as the `<w:add/>` element in the request. The actual result is returned in the `<w:addResult/>` element. Note that the web service developer can define all these element names.

Were there a problem processing the SOAP request on the server, assuming the request actually reached that far, then a `<soap:Fault>` element will be returned. For instance, if the first operand in the example had been wrongly entered as a letter instead of the number, you might receive the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Client</faultcode>
      <faultstring>Server was unable to read request.
                   Input string was not in a correct format.
                   There is an error in XML document (4, 13).
      </faultstring>
      <detail/>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

The `<soap:Fault>` element, of which there can be only one, gives a clue as to the problem encountered. The most telling information is that contained in `<faultcode/>`. There are a limited number of options for this value, of which the two common ones are `soap:Server` and `soap:Client`. A `soap:Server` fault code could indicate a problem such as the server being unable to connect to a database. In this case, resending the message may well succeed. If `soap:Client` is specified, it often means that the message is incorrectly formatted and will not succeed without some modification.

A more human-readable error message is stored in the `<faultstring/>` element, which contains application-specific error details. If a secondary system, such as a database, is the primary cause of a web service error, information pertaining to this error may be returned in an optional `<faultactor/>` element (not shown in the previous example).

Document-Style SOAP

The document style of SOAP relies on XML schemas to designate the format of the request and response. This style seems to be gaining in popularity and some predict that it will eventually all but replace the RPC style. For a lucid explanation of why people are shying away from RPC with SOAP encoding, see `http://msdn.microsoft.com/library/en-us/dnsoap/html/argsoape.asp`.

A document-style request may not look that different from an RPC-style request. For example, the RPC request example from the previous section could be a valid document-style request by simply removing the `soap:encodingStyle` attribute. The difference is that an RPC request always follows the same style, with the method name in a containing element around its parameters; the document-style request has no such constraints. Here's an example that is completely different from an RPC request:

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <w:add xmlns:w="http://www.wrox.com/services/math" op1="4.5" op2="5.4" />
  </soap:Body>
</soap:Envelope>
```

Note that the highlighted line contains the method name (`add`) and two operands (`op1` and `op2`) in a single element. This construct is not possible using RPC-style requests. The document style has this flexibility because there is an accompanying XML schema. A web service can use this XML schema to validate the structure of the request; the service is then free to use the information in the request appropriately. Responses follow the same basic rules as requests: they can be very similar to RPC style or completely different, again based on an XML schema.

> **Web services created using Visual Studio .NET are, by default, in document style (although this can be changed by applying various attributes to the underlying code).**

At this point, you might be wondering where the XML schema is kept and how it is made available to both the client and the service. The answer to those questions lies in yet another abbreviation: WSDL.

WSDL

Web Services Description Language (WSDL) is another XML-based language that was created to describe the usage of a particular web service, or rather, how a particular service could be called. The resulting specification describes an incredibly dense language, designed to be extremely flexible and allow for as much re-use as possible; it is the sort of document that is manually constructed only by the most ardent enthusiast. Typically, a software tool is used for the initial WSDL file creation and then hand tweaked, as necessary.

The following is a WSDL file describing a sample math service with a single `add` method (which you will be building later):

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions
                 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
                 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                 xmlns:s="http://www.w3.org/2001/XMLSchema"
                 xmlns:tns="http://www.wrox.com/services/math"
                 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                 targetNamespace="http://www.wrox.com/services/math">
  <wsdl:types>
    <s:schema elementFormDefault="qualified"
              targetNamespace="http://www.wrox.com/services/math">
```

```
      <s:element name="add">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="op1" type="s:float" />
            <s:element minOccurs="1" maxOccurs="1" name="op2" type="s:float" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="addResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="addResult"
                       type="s:float" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </wsdl:types>
  <wsdl:message name="addSoapIn">
    <wsdl:part name="parameters" element="tns:add" />
  </wsdl:message>
  <wsdl:message name="addSoapOut">
    <wsdl:part name="parameters" element="tns:addResponse" />
  </wsdl:message>
  <wsdl:portType name="MathSoap">
    <wsdl:operation name="add">
      <wsdl:documentation>
       Returns the sum of two floats as a float
      </wsdl:documentation>
      <wsdl:input message="tns:addSoapIn" />
      <wsdl:output message="tns:addSoapOut" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="MathSoap" type="tns:MathSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                  style="document" />
    <wsdl:operation name="add">
      <soap:operation soapAction="http://www.wrox.com/services/math/add"
                      style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="Math">
    <wsdl:documentation>
      Contains a number of simple arithmetical functions
    </wsdl:documentation>
    <wsdl:port name="MathSoap" binding="tns:MathSoap">
      <soap:address location="http://localhost/Math/Math.asmx" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Remember that this WSDL is describing only a basic service that adds two numbers; for simplicity, the three other methods you will be implementing have been removed. Although this WSDL file is long and complex, you should understand what its various sections mean.

The document element, `<wsdl:definitions/>`, encompasses the content and allows the declaration of various namespaces. The next element, `<wsdl:types/>`, contains the XML schema used by the service. Inside of this element is `<s:schema/>`, which describes the format of all elements that can appear in the `<soap:Body/>` of either the request or the response.

The first element described in the schema is `<add/>`. Since the `<s:schema/>` element has `elementFormDefault` set to `qualified`, `<add/>` is assumed to be in the namespace designated by the `targetNamespace` attribute, `http://www.wrox.com/services/math`. The `<add/>` element is then declared to contain a sequence of two other elements, `<op1/>` and `<op2/>`. Both of these elements have `minOccurs` and `maxOccurs` set to `1`, which means that they must both appear once and once only. They also both have a `type` attribute of `s:float`, which is one of the built-in XML schema types.

> *You can find a complete list of XML schema data types at* `www.w3.org/TR/xmlschema-0/#CreatDt`. *If your service needs more complicated types than these, you can construct complex types by aggregating and restricting these base types.*

Next in the schema is another `<s:element/>`, this one describing `<addResponse/>`. This element is defined to have one child element, `<addResult/>`, which contains the result of the operation (also defined as type `s:float`). This is the last entry in the included XML schema.

Back in the main body of the WSDL file is a short section describing two `<wsdl:message/>` elements: `addSoapIn` and `addSoapOut`. Each of these elements has a `<wsdl:part/>` element that specifies the `element` in the XML schema to use. These both refer to the two elements `add` and `addResponse`, respectively. This section states the format of each message.

The following section, `<wsdl:portType/>`, is used to group the `<wsdl:message>` elements into operations. An *operation* is considered to be a single unit of work and, therefore, comprises of a `<wsdl:input>` and normally a `<wsdl:output>` and an optional `<wsdl:fault>` element. The preceding example has one `<wsdl:portType>` and describes a `<wsdl:operation/>` named `add`. The message attributes on its `<wsdl:input>` and `<wsdl:output>` children refer back to the `<wsdl:message>` elements previously defined. There is also a `<wsdl:documentation/>` element containing a user-friendly description of the method. (You will learn where this information comes from later in the chapter.)

After the port type is a `<wsdl:binding/>` block. A *binding* pairs an operation with a protocol used to communicate with the service. There are three bindings described in the WSDL specification, SOAP, HTTP GET/POST, and MIME.

> **This chapter concentrates on the SOAP binding. The HTTP GET/POST binding deals with how URLs are constructed (for GET requests) or how the form data is encoded (for POST requests). The MIME binding allows parts of the message, normally the output, to be expressed in different mime types. This means that one part of the response could be in XML, whereas a second part could be in HTML.**
>
> **You can read more about these alternative bindings at** `www.w3.org/TR/2002/WD-wsdl12-bindings-20020709/`**.**

First, the `name` of the binding is set to `MathSoap`, and the `type` points to the `MathSoap` port type defined in the `<wsdl:portType>` section. Second, the `<soap:binding/>` element uses the `transport` attribute to specify that the service operates over HTTP. The `<wsdl:operation/>` element simply defines the `name` of the method, `add`. The `<soap:operation/>` element contains the `soapAction` that needs to be incorporated into the header of the HTTP request as well as the `style` attribute, which specifies that the SOAP message will be a document type rather than an RPC type.

> *The main difference between a document-style message and an RPC-style message is that the document style sends the message as elements within the* `<soap:body>` *that can have whatever structure the sender and receiver agree on using the embedded schema as a guide. An RPC-style message, however, has an element named after the method being called. This in turn will have one element for each parameter the method accepts.*

The `<soap:operation/>` element has two children, `<wsdl:input>` and `<wsdl:output>`, which are used to further describe the request and response format. In this case, the `<soap:body>` specifies a `use` attribute of `literal`. In practical terms, this is the only option with document-style services; with RPC-style services the choice extends to *encoded*, in which case the `<soap:body>` would further specify exactly which encoding type was to be used for the parameter types.

The final part of the document, `<wsdl:service/>`, deals with how to call the service from a client. It also contains a human-readable description of the service and the `<wsdl:port/>` element, which references the `MathSoap` binding in the last document section. Perhaps the most important element in this section is `<soap:address/>`, which contains the crucial `location` attribute containing the URL needed to access the service.

```
<wsdl:service name="Math">
  <wsdl:documentation>
    Contains a number of simple arithmetical functions
  </wsdl:documentation>
  <wsdl:port name="MathSoap" binding="tns:MathSoap">
    <soap:address location="http://localhost/Math/Math.asmx" />
  </wsdl:port>
</wsdl:service>
```

Looking at a complete WSDL file may be a bit daunting for new developers, but the good news is that you will probably never have to hand code one yourself. In fact, the example file in this section was created automatically by the .NET web service, which examines the underlying code and generates the necessary XML.

> *XML schemas are a vast topic and a full discussion is outside the scope of this book. If you'd like to learn more about XML schemas, consider picking up Beginning XML, 3rd Edition (Wiley Publishing, ISBN 0-7645-7077-3), or for a web tutorial, visit* `www.w3schools.com/schema/default.asp`.

REST

Representational State Transfer, often abbreviated as REST, describes a way of using the existing HTTP protocol to transmit data. Although used mostly for web services, REST can be used for any type of HTTP-based request and response systems as well. In regard to web services, REST enables you to call a given URL in a specific format to return data (which will also be in a specific format). This data may contain further information on how to retrieve even more data. For the web service usage, the data will be returned as XML.

For example, suppose Wrox would like to provide a way for others to retrieve a list of all authors. REST-style web services use simple URLs to access data; the Wrox Book service could use this URL to retrieve the list of authors:

```
http://www.wrox.com/services/authors/
```

This service may return an XML representation of the known authors along with information on how to access details about each one, such as:

```
<?xml version="1.0" encoding="utf-8" ?>
<authors xmlns:xlink="http://www.w3.org/1999/xlink"
         xmlns="http://www.wrox.com/services/authors-books"
         xlink:href="http://www.wrox.com/services/authors/">
  <author forenames="Michael" surname="Kay"
    xlink:href="http://www.wrox.com/services/authors/kaym"
    id="kaym"/>
  <author forenames="Joe" surname="Fawcett"
    xlink:href="http://www.wrox.com/services/authors/fawcettj"
    id="fawcettj"/>
  <author forenames="Jeremy" surname="McPeak"
    xlink:href="http://www.wrox.com/services/authors/mcpeakj"
    id="mcpeakj"/>
  <author forename="Nicholas" surname="Zakas"
    xlink:href="http://www.wrox.com/services/authors/zakasn"
    id="zakasn"/>
  <!--
    More authors
  -->
</authors>
```

There are a couple of things to note about this XML. First, a default namespace of `http://www.wrox.com/services/authors-books` is declared so that any un-prefixed elements, such as `<authors/>`, are assumed to belong to this namespace. This means that the `<authors/>` element can be differentiated from another element with a similar name but from a different namespace. The namespace URI, `http://www.wrox.com/services/authors-books`, is used simply as a unique string; there is no guarantee that an actual resource is available at that location. The key is that it is a *Uniform Resource Identifier* (URI), which is simply an identifier, not a *Uniform Resource Locator* (URL), which would indicate that a resource is available at a specific location.

Second, note the use of the `href` attribute from the `http://www.w3.org/1999/xlink` namespace. Although not essential, many REST-style services have now standardized on this notation for what, in HTML, would be a standard hyperlink.

> *XLink is a way of linking documents that goes way beyond the straightforward hyperlinks of HTML. It provides capabilities to specify a two-way dependency so that documents can be accessible from each other as well as indicating how a link should be activated — for example, by hand, automatically, or after a preset time. Its cousin, XPointer, is concerned with specifying sections within a document and arose from the need for a more powerful notation than the simple named links within HTML pages.*
>
> *Although they have both reached recommendation status at the W3C, they are still not widely used. For more information, visit* `www.w3.org/XML/Linking`.

If used in a web site or web application, the XML returned from the REST service would be transformed, either client-side or server-side, to a more user-friendly format (most likely HTML) — perhaps something like this:

```
<html>
  <head>
    <title>Wrox Authors</title>
  </head>
  <body>
    <a href="http://www.wrox.com/services/authors/kaym">Michael Kay</a>
    <a href="http://www.wrox.com/services/authors/fawcettj">Joe Fawcett</a>
    <a href="http://www.wrox.com/services/authors/mcpeakj">Jeremy McPeak</a>
    <a href="http://www.wrox.com/services/authors/zakasn">Nicholas Zakas</a>
  </body>
</html>
```

The user could then retrieve individual author data by following one of the links, which may return XML similar to this:

```
<?xml version="1.0" encoding="utf-8" ?>
<author xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns="http://www.wrox.com/services/authors-books"
        xlink:href="http://www.wrox.com/services/authors/fawcettj"
        id="fawcettj" forenames="Joe" surname="Fawcett">
  <books>
    <book
      xlink:href="http://www.wrox.com/services/books/0764570773"
      isbn="0764570773" title="Beginning XML"/>
    <book
      xlink:href="http://www.wrox.com/services/books/0471777781"
      isbn="0471777781" title="Professional Ajax"/>
  </books>
</author>
```

Again, you see that the elements are in the `http://www.wrox.com/services/authors-books` namespace and the `xlink:href` attribute is a way to extract further information. An HTML representation of this data may look like this:

```
<html>
  <head>
    <title>Author Details</title>
  </head>
  <body>
    <p>Details for
    <a href="http://www.wrox.com/services/authors/fawcettj">Joe Fawcett</a></p>
    <p>Books</p>
    <a href="http://www.wrox.com/services/books/0764570773">Beginning XML</a>
    <a href="http://www.wrox.com/services/books/0471777781">Professional Ajax</a>
  </body>
</html>
```

And if, per chance, the user feels like following the link for *Professional Ajax*, he or she may receive the following XML in response:

```
<?xml version="1.0" encoding="utf-8" ?>
<book xmlns:xlink="http://www.w3.org/1999/xlink"
      xmlns="http://www.wrox.com/services/authors-books"
      xlink:href="http://www.wrox.com/services/books/0471777781"
      isbn="0471777781">
  <genre>Web Programming</genre>
  <title>Professional AJAX</title>
  <description>How to take advantage of asynchronous JavaScript
   and XML to give your web pages a rich UI.</description>
  <authors>
    <author forenames="Nicholas" surname="Zakas"
      xlink:href="http://www.wrox.com/services/authors/zakasn"
      id="zakasn" />
    <author forenames="Jeremy" surname="McPeak"
      xlink:href="http://www.wrox.com/services/authors/mcpeakj"
      id="mcpeakj" />
    <author forenames="Joe" surname="Fawcett"
      xlink:href="http://www.wrox.com/services/authors/fawcettj"
      id="fawcettj" />
  </authors>
</book>
```

REST-style services are fairly straightforward and follow a repeating pattern. For example, you may get a complete list of authors by using `http://www.wrox.com/services/authors/`, whereas a slight modification, adding an author ID at the end, may retrieve information about a single author, perhaps from `http://www.wrox.com/services/authors/fawcettj`.

The service can be implemented in any number of ways. It could be through static web pages or, more likely, some sort of server-side processing such as ASP, JSP, or PHP, that fetch data from a database and return the appropriately constructed XML. In this case the URL would be mapped by the server into an application-specific way of retrieving the data, possibly by invoking a stored procedure on the database in question.

> *You can read more about REST-style services (also called RESTful services) at* `www.networkworld.com/ee/2003/eerest.html`.

The .NET Connection

By most accounts, Microsoft spearheaded the web services movement with the introduction of SOAP. When Microsoft presented SOAP to IBM as a way of transporting data, IBM quickly came on board, helping to develop what later became WSDL. With the combined force of Microsoft and IBM, many more big companies jumped on board, such as Oracle, Sun, and HP. The standards were established and the beginning of the web service era was on the horizon, but there was a catch: there were no tools to facilitate the creation of web services. That's where .NET came in.

Microsoft released the .NET Framework in 2000 with the aim of providing a platform-independent development framework to compete with Java. Since Microsoft started nearly from scratch with the .NET initiative, they built in strong support for XML, as well as the creation and consumption of web services using SOAP and WSDL. Using .NET, there are simple ways to provide a web services wrapper around existing applications as well as exposing most .NET classes using web services.

When developing web services, you can decide how much interaction with SOAP and WSDL is necessary. There are tools to shield developers from the underlying structure, but you can also change fine details if necessary. The 2005 version of the .NET Framework makes even more use of XML and web services.

Design Decisions

Although the .NET Framework makes web service development easier, it is by no means the only way to create them. Just like any other programming task, there are several design and development decisions that must be made. Remember, web services provide a platform-independent way of requesting and receiving data, so the service consumer doesn't need (or in many cases want) information about how it is implemented. Unfortunately, there are some things to be aware of when interoperability is a concern:

- **Not all platforms support the same data types.** For example, many services return an ADO.NET dataset. A system without .NET will be unlikely to understand this data form. Similarly, arrays can be problematic because they can be represented in any number of ways.
- **Some services are more tolerant of missing or extra headers in the request.** This problem is allied to consumers that do not send all the correct headers, which can create problems, especially when it comes to securing a service.

In an effort to overcome these and other related issues, the Web Services Interoperability Organization was formed. You can find its aims, findings, and conformance recommendations at `www.ws-i.org/`.

When creating a web service, your first decision is which platform to use. If you choose Windows, you'll almost certainly use IIS as your web server. You can use ASP.NET to create your web services, or ASP for older versions of IIS (though this is more difficult). The examples in this chapter use ASP.NET.

If you are using UNIX or Linux, you will likely be using JSP or PHP, both of which have open source web servers available. Using these, you need to program in Java or PHP, respectively, to create web services.

> *The Axis project (*`http://ws.apache.org/axis/`*) has development tools for both Java and C++.*
>
> *For PHP there are also plenty of options, including PhpXMLRPC (*`http://phpxmlrpc.sourceforge.net/`*) and Pear SOAP (*`http://pear.php.net/package/SOAP`*).*

After you've chosen your language, you'll need to decide who will have access to your service. Will your application be the only one calling it, or will your service be accessible publicly? If the latter, you will need to take into account the interoperability issues discussed previously; if the former, you can take advantage of some of the specialized features provided by the client or the server.

With a web service created, the next step is to *consume* it. Any application that calls a web service is considered a consumer. Typically, consuming a web service follows a distinct pattern: create a request, send the request, and act on the response received. The exact method for taking these steps is up to the functionality that is accessible by the consumer.

Creating a Windows Web Service

Now it's time to move away from the specifications and theories to create a simple web service. The web service described in this section uses document-style SOAP requests and responses to implement the Math service described in the WSDL file earlier in this chapter. Note that this process uses free tools available from Microsoft, which involve a little more work than if you were to use, say, Visual Studio .NET. However, the extra work you'll do in this section will aid in your understanding of web services and could pay handsome dividends later in your development career should anything go wrong with an auto-generated service.

System Requirements

To create this service, you will need three minimum requirements:

- A Windows machine running IIS 5 or greater. This comes as standard on all XP Professional machines and on all servers from Windows 2000 onwards.
- The .NET Framework must be installed on the machine running IIS. You will also need the .NET Software Development Kit (SDK) on the machine you are developing on. For the purposes of this example, it is assumed that you are developing on the machine that is running IIS. (You can download both the .NET Framework and the SDK from `http://msdn.microsoft.com/netframework/downloads/updates/default.aspx`.)
- A text editor to write the code. This can simply be Notepad, which is standard on all Windows machines and is more than adequate for the purposes of this example (although for serious development an editor that supports syntax highlighting is preferable).

Configuring IIS

The first task is to create a home for your service. Go to Start➪Administrative Tools and select Internet Information Services. (Alternatively, enter **%SystemRoot%\System32\inetsrv\iis.msc** in the Start➪Run box and click the OK button.) Expand the tree on the left to show the Default Web Site node, and then right-click and choose New➪Virtual Directory, as shown in Figure 6-1. This will bring up the Virtual Directory Creation Wizard, where you choose the name of the web service folder as seen by the client (see Figure 6-2).

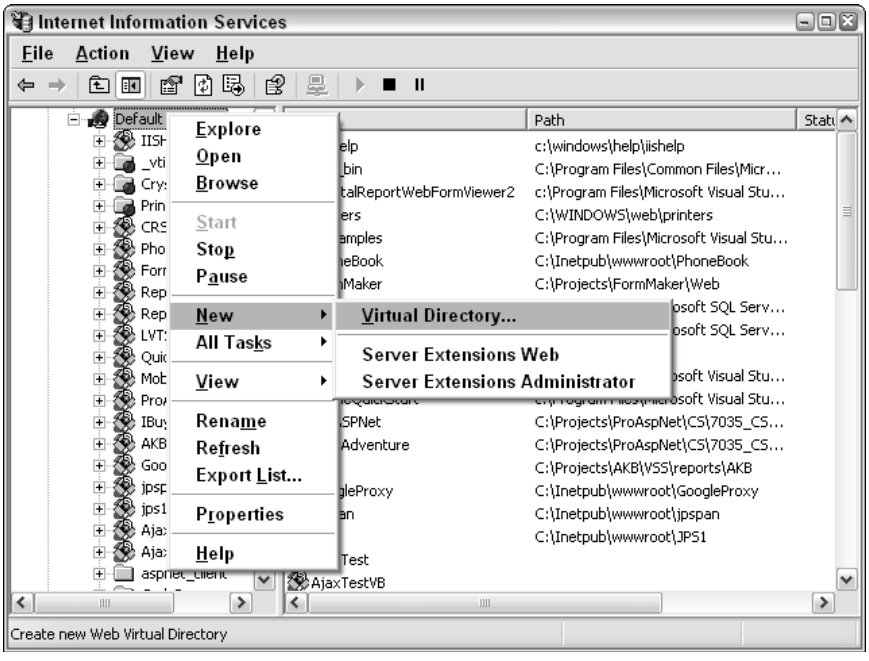

Figure 6-1

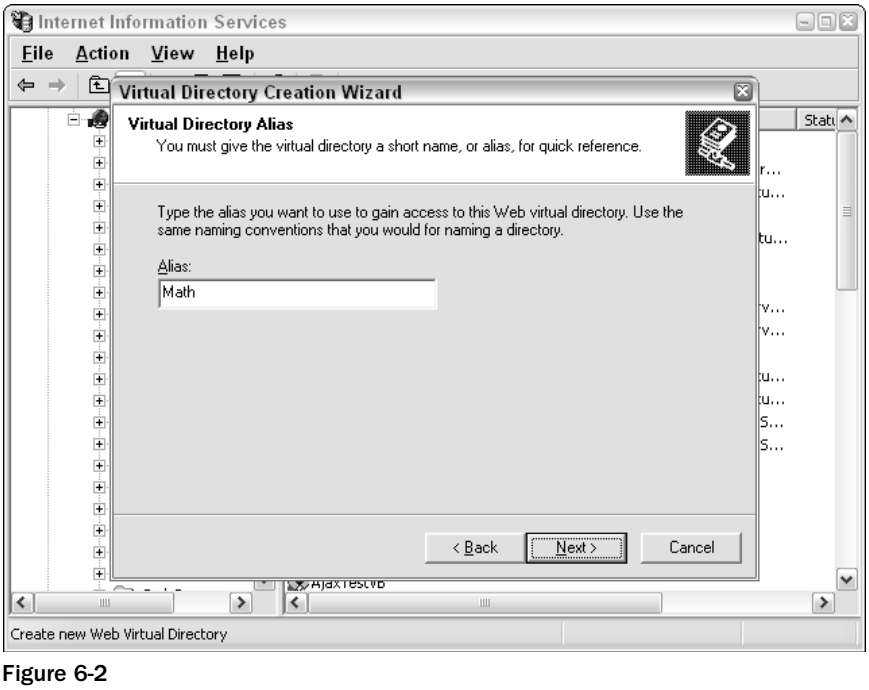

Figure 6-2

Name the folder **Math**, and then click Next. On the next screen, browse to the standard IIS directory of `C:\InetPub\wwwroot`. Create a new folder, also named **Math**, directly below this folder. Accept the defaults for the remaining screens of the wizard. When this is done, use Windows Explorer to create a new folder underneath Math named **bin**, which will hold the DLL once the service is built. Your folder hierarchy should now look like Figure 6-3.

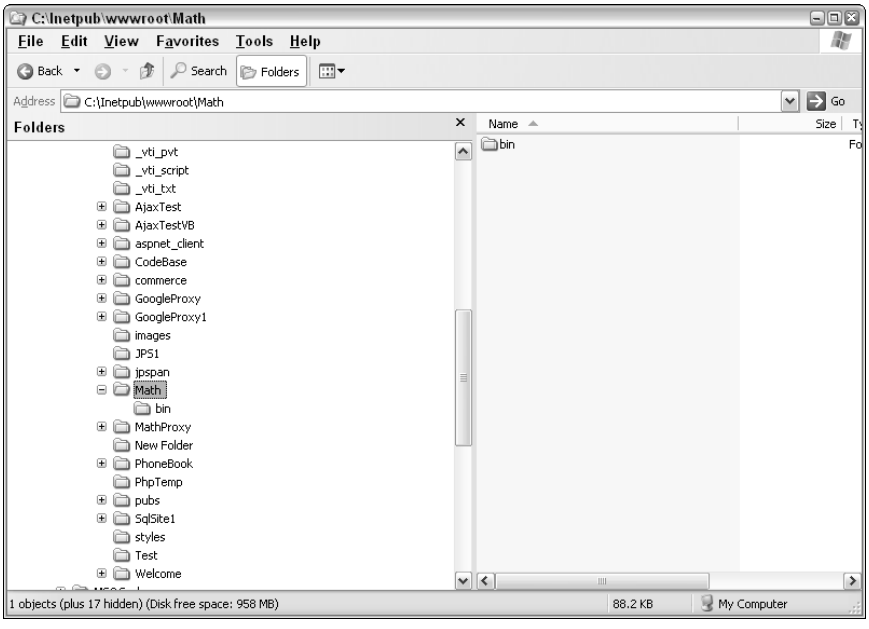

Figure 6-3

Coding the Web Service

The web service you are creating is quite simple. Its name is *Math*, and it implements the four basic arithmetic operations: addition, subtraction, multiplication, and division. These four operations each accept two parameters, defined as floats, and return a float as the result. The class itself will be coded in C#, and the web service will be published in ASP.NET.

Create a new file in your favorite text editor and add the following three lines:

```
using System;
using System.Web;
using System.Web.Services;
```

This code doesn't add any extra functionality, but it does save you from having to type fully qualified class names. Since you will be using several classes from these namespaces, it saves space to reference them here.

Next, create a namespace called *Wrox.Services* and a class called *Math* that inherits from `System.Web.Services.WebService`:

```
namespace Wrox.Services
{
  [WebService (Description = "Contains a number of simple arithmetical functions",
               Namespace = "http://www.wrox.com/services/math")]

  public class Math : System.Web.Services.WebService
  {
    //class code here
  }

}
```

The `namespace` keyword is used to in a similar way as namespaces in XML; it means the full name of the `Math` class is `Wrox.Services.Math`. Immediately inside the namespace definition is an attribute called `WebService`, which marks the class on the following line as a web service. Doing this enables extra functionality for the class, such as generating a WSDL file. You will also notice that a `Description` parameter is included (and will also appear in the WSDL file).

Then comes the class name, `Math`, which inherits from the base class of `System.Web.Services.WebService`. Inheriting from this class means that you don't need to worry about any specific code for writing web services; the base class handles all of this. You can simply focus on writing the methods that will be published as part of the web service.

Defining a method to be used in a web service is as easy as writing a regular method and tagging it with the special `WebMethod` attribute:

```
[WebMethod(Description = "Returns the sum of two floats as a float")]
public float add(float op1, float op2)
{
  return op1 + op2;
}
```

Once again, the code is very simple. (What could be simpler than an addition operation?) Any method that has a `WebMethod` attribute preceding it is considered part of the web service. The `Description` parameter will become part of the generated WSDL file. Although you can write as many methods as you'd like, here is the complete code for this example, including the four arithmetic methods:

```
using System;
using System.Web;
using System.Web.Services;

namespace Wrox.Services
{
  [WebService (Description = "Contains a number of simple arithmetical functions",
               Namespace = "http://www.wrox.com/services/math")]
    public class Math : System.Web.Services.WebService
    {

    [WebMethod(Description = "Returns the sum of two floats as a float")]
    public float add(float op1, float op2)
    {
      return op1 + op2;
```

```
    }

    [WebMethod(Description = "Returns the difference of two floats as a float")]
    public float subtract(float op1, float op2)
    {
      return op1 - op2;
    }

    [WebMethod(Description = "Returns the product of two floats as a float")]
    public float multiply(float op1, float op2)
    {
      return op1 * op2;
    }

    [WebMethod(Description = "Returns the quotient of two floats as a float")]
    public float divide(float op1, float op2)
    {
      return op1 / op2;
    }
  }
}
```

Save this file in the Math directory and name it `Math.asmx.cs`.

Create another text file and enter the following line:

```
<%@WebService Language="c#" Codebehind="Math.asmx.cs" Class="Wrox.Services.Math" %>
```

This is the ASP.NET file that uses the `Math` class you just created. The `@WebService` directive tells the page to act like a web service. The meaning of the other attributes should be fairly obvious: `Language` specifies the language of the code to use; `Codebehind` specifies the name of the file that the code exists in; and `Class` specifies the fully qualified name of the class to use. Save this file in Math directory as well, with the name `Math.asmx`.

Creating the Assembly

After you have created these two files, you can proceed with the next stage: compiling the source code into an assembly that will be housed in a DLL. To do this, you can use the C# compiler that comes with the .NET SDK. This will be in your Windows directory below the `Microsoft.Net\Framework\<version number>` folder (for example, `C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\`).

The easiest way to compile and debug the code is to create a batch file. Create another text file and enter the following (it should all be on one line despite the formatting of the book):

```
C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\csc.exe /r:System.dll
/r:System.Web.dll
/r:System.Web.Services.dll /t:library /out:bin\Math.dll  Math.asmx.cs
```

Remember to modify the path to `csc.exe` if necessary and save it to the Math folder as `MakeService.bat`.

> **If you are using Notepad, be careful using the Save As dialog box. Make sure that the File Type box is set to All Files, or else enclose the file name in double quotes. Otherwise, Notepad adds a `.txt` extension to your file.**

Next, you need to compile the DLL. Open a command prompt from Start➪Run and type **cmd**. Now navigate to the Math folder by typing **cd \inetpub\wwwroot\Math**. Finally, run the following batch file:

```
C:\inetpub\wwwroot\Math\MakeService.bat
```

If all is well, you should be greeted with the compiler displaying some version and copyright information, and then a blank line. This is good news and indicates that the compilation was successful. (If there were any errors, they will be outputted to the console. Check the lines indicated and correct any syntax or spelling mistakes.)

Assuming the DLL has compiled, you are ready to test the service. One of the joys of .NET web services is that a whole test harness is created automatically for you. Open your web browser and navigate to `http://localhost/Math/math.asmx`. You should soon see a page similar to the one displayed in Figure 6-4.

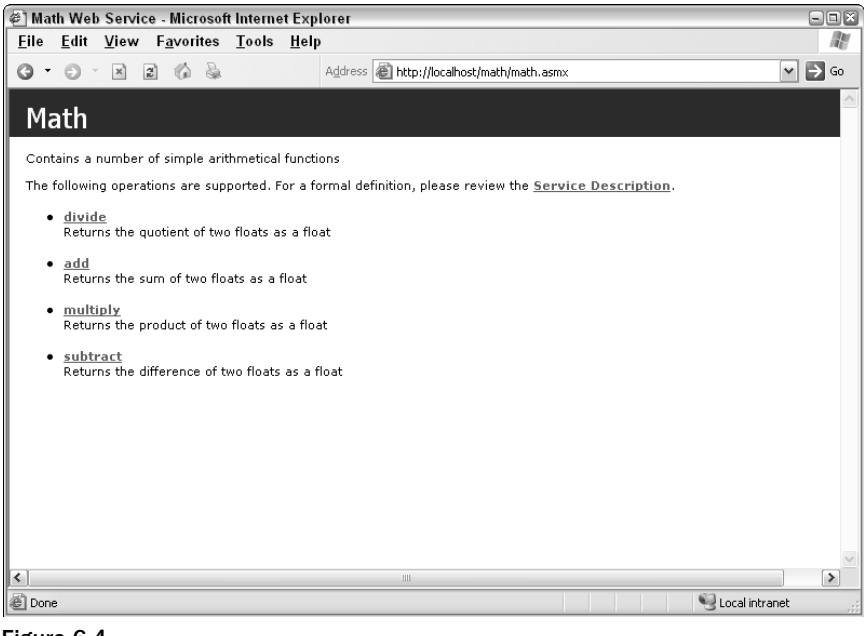

Figure 6-4

You have the choice to try any of the four methods or you can view the generated WSDL file by clicking the Service Description link. This reveals a WSDL file similar to the example earlier in the chapter, but this will have entries for all four methods.

> **Another way of viewing the WSDL file is to add `?WSDL` to the web service URL, such as `http://localhost/Math/math.asmx?WSDL`.**

Since you have probably had enough of the `add` method, try the `divide` method. Clicking the link from the previous screen should display the page in Figure 6-5.

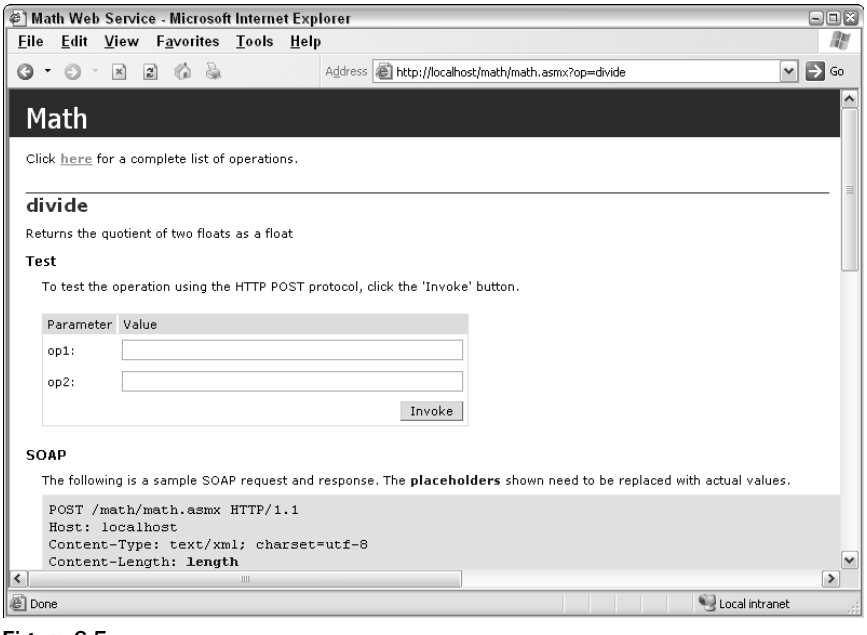

Figure 6-5

Below the divide heading is the description you used with the `WebMethod` attribute, and below that is a small test form. If you enter two numbers, such as 22 and 7, you'll receive a response such as the one displayed in Figure 6-6.

In a production environment, you can now remove the `Math.asmx.cs` file, as it is no longer needed. The `Math.asmx` simply passes on all requests directly to the DLL.

This test harness does not use SOAP for the request and response. Instead, it passes the two operands as a POST request; the details of how to do this are shown at the bottom of the divide page in Figure 6-5. Now that you have a web service defined, it's time to use Ajax to call it.

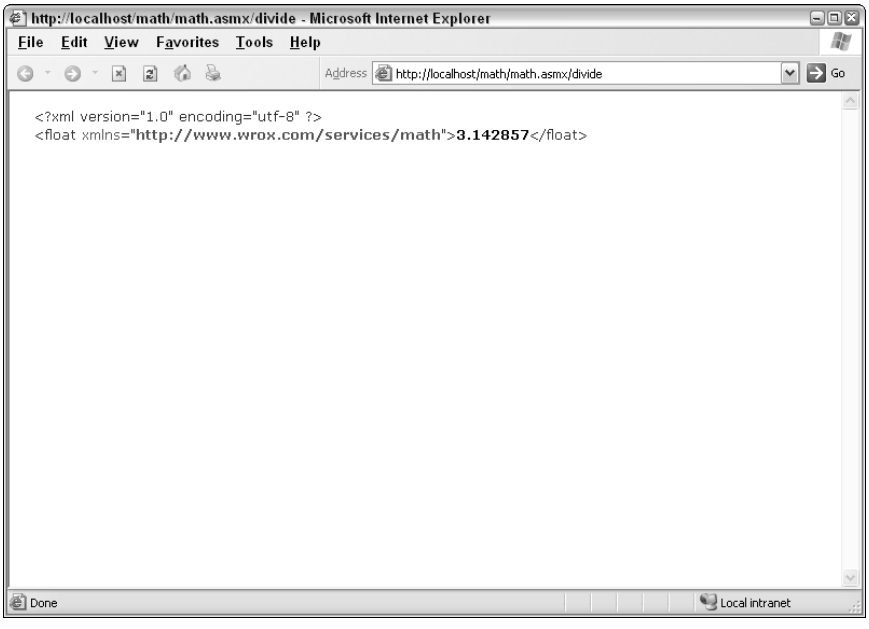

Figure 6-6

Web Services and Ajax

Now that you have a basic understanding of web services, and have created your own, you're probably wondering what this has to do with Ajax. Quite simply, web services are another avenue for Ajax applications to retrieve information. In the last chapter, you learned how to retrieve and use RSS and Atom feeds to display information to the user, which is very similar to using web services. The main difference is that by using web services, you are able to pass information to the server that can be manipulated and sent back; you aren't simply pulling information.

You can use JavaScript to consume a web service from within a web page so long as your users have a modern browser. Internet Explorer 5.0 and higher, as well as the Mozilla family of browsers (including Firefox), all have some functionality that allows web services to be consumed.

Creating the Test Harness

First, you'll need a test harness to test the various approaches to calling web services from the browser. This test harness is fairly simple: there is a list box to select one of the four arithmetic operations to execute, a text box for each of the two operands, and a button to invoke the service. These controls are disabled until the page has fully loaded. Below those controls is another text box to display any valid result as well as two text areas to display the request and response data:

```
<html>
  <head>
    <title>Web Service Test Harness</title>
    <script type="text/javascript">

      var SERVICE_URL = "http://localhost/Math/Math.asmx";
      var SOAP_ACTION_BASE = "http://www.wrox.com/services/math";

      function setUIEnabled(bEnabled)
      {
        var oButton = document.getElementById("cmdRequest");
        oButton.disabled = !bEnabled;
        var oList = document.getElementById("lstMethods");
        oList.disabled = !bEnabled
      }

      function performOperation()
      {
        var oList = document.getElementById("lstMethods");
        var sMethod = oList.options[oList.selectedIndex].value;
        var sOp1 = document.getElementById("txtOp1").value;
        var sOp2 = document.getElementById("txtOp2").value;

        //Clear the message panes
        document.getElementById("txtRequest").value = "";
        document.getElementById("txtResponse").value = "";
        document.getElementById("txtResult").value = "";
        performSpecificOperation(sMethod, sOp1, sOp2);
      }
    </script>
  </head>
  <body onload="setUIEnabled(true)">
    Operation: <select id="lstMethods" style="width: 200px" disabled="disabled">
      <option value="add" selected="selected">Add</option>
      <option value="subtract">Subtract</option>
      <option value="multiply">Multiply</option>
      <option value="divide">Divide</option>
    </select>
    <br/><br/>
    Operand 1: <input type="text" id="txtOp1" size="10"/><br/>
    Operand 2: <input type="text" id="txtOp2" size="10"/><br/><br/>
    <input type="button" id="cmdRequest"
       value="Perform Operation"
       onclick="performOperation();" disabled="disabled"/>
    <br/><br/>
    Result: <input type="text" size="20" id="txtResult">
    <br/>
    <textarea rows="30" cols="60" id="txtRequest"></textarea>
    <textarea rows="30" cols="60" id="txtResponse"></textarea>
  </body>
</html>
```

The `setUIEnabled()` function is used to enable and disable the user interface of the test harness. This ensures that only one request is sent at a time. There are also two constants defined, `SERVICE_URL` and `SOAP_ACTION_BASE`, that contain the URL for the web service and the SOAP action header required to

call it, respectively. The button calls a function named `performOperation()`, which gathers the relevant data and clears the text boxes before calling `performSpecificOperation()`. This method must be defined by the particular test being used to execute the web service call (which will be included using a JavaScript file). Depending on your personal browser preferences, the page resembles Figure 6-7.

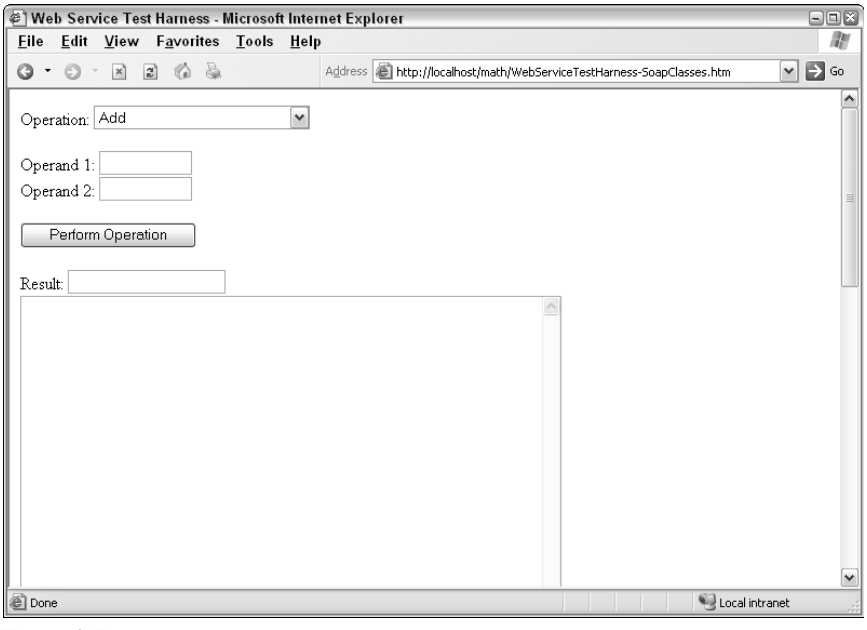

Figure 6-7

The Internet Explorer Approach

In an early effort to get developers excited about web services, Microsoft developed and released a web service behavior for Internet Explorer. *Behaviors* enable you to redefine the functionality, properties, and methods of an existing HTML element or to create an entirely new one. The advantage of behaviors is the encapsulation of functionality into a single file, ending with the `.htc` extension. Although the promise of behaviors never lived up to the hype, the web service behavior is a solid component that can be very useful to web developers. You can download the behavior from `http://msdn.microsoft.com/library/default.asp?url=/workshop/author/webservice/webservice.asp`.

> *This file and the others associated with this chapter are included in the code download for Professional Ajax, at www.wrox.com.*

You can add behaviors to elements in a number of ways. The most straightforward is to use the element's CSS `style` property. To add the web service behavior to a `<div/>`, the code would be:

```
<div id="divServiceHandler" style="behavior: url(webservice.htc);"></div>
```

This assumes that the behavior is in the same folder on the server as the web page.

To show how to use this behavior, create a new version of the test harness and insert the highlighted line directly below the `<body/>` element, as follows:

```
<body onload="setUIEnabled(true);">
<div id="divServiceHandler" style="behavior: url(webservice.htc);"></div>
Operation: <select id="lstMethods" style="width: 200px" name="lstMethods"
                  disabled="disabled">
```

The next step is to define the `performSpecificOperation()` method specific to using the web service behavior. This method accepts three arguments: the method name and the two operands. The code is as follows:

```
var iCallId = 0;
function performSpecificOperation(sMethod, sOp1, sOp2)
{
  var oServiceHandler = document.getElementById("divServiceHandler");
  if (!oServiceHandler.Math)
  {
    oServiceHandler.useService(SERVICE_URL + "?WSDL", "Math");
  }
  iCallId = oServiceHandler.Math.callService(handleResponseFromBehavior,
                                             sMethod, sOp1, sOp2);
}
```

A variable, `iCallId`, is initialized to zero. Although this plays no part in the test, it can be used to keep track of multiple simultaneous calls. Then, a reference to the `<div/>` element that has the behavior attached is stored in `oServiceHandler`. Next, a test is done to see if the behavior has already been used by checking to see whether the `Math` property exists. If it doesn't exist, you must set up the behavior by passing the URL of the WSDL file and a identifying name for the service to `useService()`. The reason for the identifier is to enable the behavior to use more than one service at a time. The `callService()` method is then executed, passing in a callback function (`handleResponseFromBehavior()`), the name of the method, and the two arguments.

When the response is received, the callback function, `handleResponseFromBehavior()`, will be called:

```
function handleResponseFromBehavior(oResult)
{
  var oResponseOutput = document.getElementById("txtResponse");
  if (oResult.error)
  {
    var sErrorMessage = oResult.errorDetail.code
                      + "\n" + oResult.errorDetail.string;
    alert("An error occurred:\n"
          + sErrorMessage
          + "See message pane for SOAP fault.");
    oResponseOutput.value = oResult.errorDetail.raw.xml;
  }
  else
  {
    var oResultOutput = document.getElementById("txtResult");
    oResultOutput.value = oResult.value;
    oResponseOutput.value = oResult.raw.xml;
  }
}
```

The callback function is passed an `oResult` object containing details about the call. If the `error` property is not zero, the relevant SOAP fault details are displayed; otherwise `oResult.value`, the returned value, is displayed on the page.

You can place the `performSpecificOperation()` and `handleResponseFromBehavior()` functions in an external JavaScript file and include them in the test harness page using the `<script/>` element, as follows:

```
<script type="text/javascript" src="WebServiceExampleBehavior.js"></script>
```

As you can see, using the web service behavior is fairly straightforward. All the work is done by the behavior behind the scenes, and although the `webservice.htc` file is a bit large for a script file (51KB), it can provide some very useful functionality.

> *If you want to see how the behavior works, feel free to examine the* `webservice.htc` *file in a text editor. Be warned, however; it was not meant as a tutorial and contains nearly 2300 lines of JavaScript.*

The Mozilla Approach

Modern Mozilla-based browsers, such as Firefox and Netscape, have some high-level SOAP classes built into their implementation of JavaScript. These browsers seek to wrap the basic strategy of making SOAP calls with easier-to-use classes. As with the previous example, you first must define the `performSpecificOperation()` function:

```
function performSpecificOperation(sMethod, sOp1, sOp2)
{
  var oSoapCall = new SOAPCall();
  oSoapCall.transportURI = SERVICE_URL;
  oSoapCall.actionURI = SOAP_ACTION_BASE + "/" + sMethod;
  var aParams = [];
  var oParam = new SOAPParameter(sOp1, "op1");
  oParam.namespaceURI = SOAP_ACTION_BASE;
  aParams.push(oParam);
  oParam = new SOAPParameter(sOp2, "op2");
  oParam.namespaceURI = SOAP_ACTION_BASE;
  aParams.push(oParam);
  oSoapCall.encode(0, sMethod, SOAP_ACTION_BASE, 0, null, aParams.length, aParams);
  var oSerializer = new XMLSerializer();
  document.getElementById("txtRequest").value =
                       oSerializer.serializeToString(oSoapCall.envelope);
  setUIEnabled(false);


  //more code here}
```

This script takes advantage of a number of built-in classes, the first of which is `SOAPCall`. This class wraps web service functionality in a similar way to the web service behavior for Internet Explorer. After creating an instance of `SOAPCall`, you set two properties: `transportURI`, which points to the web service location, and `actionURI`, which specifies the SOAP action and method name.

Next, two parameters are created using the `SOAPParameter` constructor, which takes the value and the name of the parameter to create. Each parameter has its namespace URI set to the value of the `targetNamespace` in the WSDL schema section. In theory this shouldn't be necessary, but the Mozilla SOAP classes seem to be designed with the RPC style in mind and our service uses the document style, so this extra step is needed. Both of these parameters are pushed onto the `aParams` array. The `encode()` method prepares all the data for the call. There are seven parameters for this call. The first is the version of SOAP being used, which can be set to zero unless it is important that a specific version is needed. The second parameter is the name of the method to use, and the third is that of the `targetNamespace` from the schema portion of the WSDL file. The next parameter is the count of how many extra headers are needed in the call (none in this case), followed by an array of these headers (here set to `null`). The last two parameters contain the number of `SOAPParameter` objects being sent and the actual parameters, respectively.

Next, you actually need to send the request, which you can do by using the `asyncInvoke()` method, as follows:

```
function performSpecificOperation(sMethod, sOp1, sOp2)
{
  var oSoapCall = new SOAPCall();
  oSoapCall.transportURI = SERVICE_URL;
  oSoapCall.actionURI = SOAP_ACTION_BASE + "/" + sMethod;
  var aParams = [];
  var oParam = new SOAPParameter(sOp1, "op1");
  oParam.namespaceURI = SOAP_ACTION_BASE;
  aParams.push(oParam);
  oParam = new SOAPParameter(sOp2, "op2");
  oParam.namespaceURI = SOAP_ACTION_BASE;
  aParams.push(oParam);  oSoapCall.encode(0, sMethod, SOAP_ACTION_BASE, 0, null,
                  aParams.length, aParams);
  document.getElementById("txtRequest").value =
                        oSerializer.serializeToString(oSoapCall.envelope);
  setUIEnabled(false);

  oSoapCall.asyncInvoke(
                        function (oResponse, oCall, iError)
                        {
                          var oResult = handleResponse(oResponse, oCall, iError);
                          showSoapResults(oResult);
                        }
                        );
}
```

The `asyncInvoke()` method accepts only one argument: a callback function (`handleResponse()`). This function will be passed three arguments by the SOAP call: a `SOAPResponse` object, a pointer to the original SOAP call (to track multiple instances, if necessary), and an error code. These are all passed to the `handleResponse` function for processing, when the call returns:

```
function handleResponse(oResponse, oCall, iError)
{
  setUIEnabled(true);
  if (iError != 0)
  {
```

```
      alert("Unrecognized error.");
      return false;
    }
    else
    {
      var oSerializer = new XMLSerializer();
      document.getElementById("txtResponse").value =
                    oSerializer.serializeToString(oResponse.envelope);
      var oFault = oResponse.fault;
      if (oFault != null)
      {
        var sName = oFault.faultCode;
        var sSummary = oFault.faultString;
        alert("An error occurred:\n"  + sSummary
                                + "\n" + sName
                                + "\nSee message pane for SOAP fault");
        return false;
      }
      else
      {
        return oResponse;
      }
    }
}
```

If the `error` code is not zero, an error has occurred that can't be explained. This happens only rarely; in most cases an error will be returned through the `fault` property of the response object.

Another built-in class is used now, `XMLSerializer`. This takes an XML node and can convert it to a string or a stream. In this case a string is retrieved and displayed in the right-hand text area.

If `oResponse.fault` is not `null`, a SOAP fault occurred, so an error message is built and displayed to the user and no further action taken. Following a successful call, the response object is passed out as the function's return value and processed by the `showSoapResults()` function:

```
function showSoapResults(oResult)
{

  if (!oResult) return;
  document.getElementById("txtResult").value =
  oResult.body.firstChild.firstChild.firstChild.data;
}
```

After checking that `oResult` is valid, the value of the `<methodResult>` element is extracted using the DOM.

> *There is a method of the* `SoapResponse` *named* `getParameters`*, which, in theory, can be used to retrieve the parameters in a more elegant manner. It does not seem to work as advertised with document style calls, however, necessitating the need to examine the structure of the* `soap:Body` *using more primitive methods.*

The Universal Approach

The only way to consume web services on almost all modern browsers is by using XMLHttp. Because Internet Explorer, Firefox, Opera, and Safari all have some basic support for XMLHttp, this is your best bet for cross-browser consistency. Unfortunately, this type of consistency comes at a price — you are responsible for building up the SOAP request by hand and posting it to the server. You are also responsible for parsing the result and watching for errors.

The two protagonists in this scenario are the XmlHttp ActiveX class from Microsoft and the `XmlHttpRequest` class that comes with the more modern Mozilla-based browsers listed above. They both have similar methods and properties, although in the time-honored fashion of these things Microsoft was first on the scene before standards had been agreed on, so the later released Mozilla version is more W3C compliant. The basic foundation of these classes is to allow an HTTP request to be made to a web address. The target does not have to return XML — virtually any content can be retrieved — and the ability to post data is also included. For SOAP calls, the normal method is to send a POST request with the raw `<soap:Envelope>` as the payload.

In this example, you'll once again be calling the test harness. This time, however, you'll be using the zXML library to create XMLHttp objects and constructing the complete SOAP call on your own. This library uses a number of techniques to examine which XML classes are supported by the browser. It also wraps these classes and adds extra methods and properties so that they can be used in a virtually identical manner. The generation of the SOAP request string is handled in a function called `getRequest()`:

```
function getRequest(sMethod, sOp1, sOp2)
{
    var sRequest = "<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/\">\n"
                 + "<soap:Body>\n"
                 + "<" + sMethod + " xmlns=\"" + SOAP_ACTION_BASE + "\">\n"
                 + "<op1>" + sOp1 + "</op1>\n"
                 + "<op2>" + sOp2 + "</op2>\n"
                 + "</" + sMethod + ">\n"
                 + "</soap:Body>\n"
                 + "</soap:Envelope>\n";
    return sRequest;
}
```

The `getRequest()` function is pretty straightforward; it simply constructs the SOAP string in the appropriate format. (The appropriate format can be seen using the .NET test harness described in the creation of the `Math` service.) The completed SOAP string is returned by `getRequest()` and is used by `performSpecificOperation() to build the SOAP request`:

```
function performSpecificOperation(sMethod, sOp1, sOp2) {
  oXmlHttp = zXmlHttp.createRequest();
  setUIEnabled(false);
  var sRequest = getRequest(sMethod, sOp1, sOp2);
  var sSoapActionHeader = SOAP_ACTION_BASE + "/" + sMethod;
  oXmlHttp.open("POST", SERVICE_URL, true);
  oXmlHttp.onreadystatechange = handleResponse;
  oXmlHttp.setRequestHeader("Content-Type", "text/xml");
```

```
    oXmlHttp.setRequestHeader("SOAPAction", sSoapActionHeader);
    oXmlHttp.send(sRequest);
    document.getElementById("txtRequest").value = sRequest;
  }
```

First, a call is made on the `zXmlHttp` library to create an XMLHttp request. As stated previously, this will be an instance of an ActiveX class if you are using Internet Explorer or of `XmlHttpRequest` if you are using a Mozilla-based browser. The `open` method of the object received attempts to initialize the request. The first parameter states that this request will be a `POST` request containing data, and then comes the URL of the service, followed by a Boolean parameter specifying whether this request will be asynchronous or whether the code should wait for a response after making the call.

The `onreadystatechange` property of the request specifies which function will be called when the state of the request alters.

The `performSpecificOperation()` function then adds two headers to the HTML request. The first specifies the content type to be `text/xml`, and the second adds the `SOAPAction` header. This value can be read from the .NET test harness page or can be seen in the WSDL file as the `soapAction` attribute of the relevant `<soap:operation/>` element. Once the request is sent, the raw XML is displayed in the left text box. When the processing state of the request changes, the `handleResponse()` function will be called:

```
function handleResponse()
{
  if (oXmlHttp.readyState == 4)
  {
    setUIEnabled(true);
    var oResponseOutput = document.getElementById("txtResponse");
    var oResultOutput = document.getElementById("txtResult");
    var oXmlResponse = oXmlHttp.responseXML;
    var sHeaders = oXmlHttp.getAllResponseHeaders();
    if (oXmlHttp.status != 200 || !oXmlResponse.xml)
    {
      alert("Error accessing Web service.\n"
            + oXmlHttp.statusText
            + "\nSee response pane for further details.");
      var sResponse = (oXmlResponse.xml ? oXmlResponse.xml : oXmlResponseText);
      oResponseOutput.value = sHeaders + sResponse;
      return;
    }
    oResponseOutput.value = sHeaders + oXmlResponse.xml;
    var sResult =

oXmlResponse.documentElement.firstChild.firstChild.firstChild.firstChild.data;
    oResultOutput.value = sResult;
  }
}
```

The `handleResponse()` function reacts to any change of state in the request. When the `readyState` property equals `4`, which equates to complete, there will be no more processing and it can be checked to see if there is a valid result.

If the `oXmlHttp.status` does not equal `200` or the `responseXML` property is empty, an error has occurred and a message is displayed to the user. Should the error be a SOAP fault, that information is also displayed in the message pane. If, however, the error wasn't a SOAP fault, the `responseText` is displayed. If the call has succeeded, the raw XML is displayed in the right text box.

Assuming that the XML response is available, there are a number of ways to extract the actual result including XSLT, using the DOM or text parsing; unfortunately, very few of these work across browsers in a consistent manner. The DOM method of gradually stepping through the tree is not very elegant, but it does have the merit of being applicable to whichever variety of XML document is in use.

Cross-Domain Web Services

So far in this chapter you've been calling a service that resides on the same domain as the page accessing it. By doing this, you have avoided the problem of cross-domain scripting (also known as *cross-site scripting*, or XSS). As discussed earlier in this book, this problem is caused by the fact that there are security risks in allowing calls to external web sites. If the web service is on the same domain as the calling page, the browser will permit the SOAP request, but what if you want to use one of the Google or Amazon.com services?

For this you'll need to use a *server-side proxy*, which runs on your web server and makes calls on behalf of the client. The proxy then returns the information it receives back to the client. The setup resembles that of a web proxy server that is commonplace in corporate networks. In that model, all requests are passed to a central server that retrieves the web page and passes it to the requestor.

The Google Web APIs Service

Google provides a number of methods that can be called through its web service, including methods for retrieving cached copies of pages as well as the more obvious results for a given search phrase. The method you will use to demonstrate a server-side proxy is `doSpellingSuggestion`, which takes a phrase and returns what Google believes you meant to type. If your phrase is not misspelled or is so obscure that it cannot hazard a suggestion, an empty string is returned.

> **The developers' kit for the Google Web APIs service can be found at `www.google.com/apis/index.html`. The kit contains the service's SOAP and WSDL standards, as well as examples in C#, Visual Basic .NET, and Java.**
>
> **You will need to register with Google to receive a security key to allow you to access the service. There are also rules about how it can be used both commercially and for development purposes.**

Setting Up the Proxy

After you have downloaded the Google documentation and received your key, you need to set up a service on your own server that will accept the calls from your users and pass them to Google. The basic service is built in the same way as the Math service discussed earlier.

To begin, open the IIS admin tool (Start➪Administrative Tools➪Internet Information Services). Expand the tree on the left to show the Default Web Site node, and then right-click and choose New➪Virtual Directory. In the Alias field of the Virtual Directory Creation Wizard, name your new directory **GoogleProxy**, and then click Next. On the next screen, browse to the standard IIS directory of `C:\InetPub\wwwroot`, and create a new folder, also named **GoogleProxy**. Accept the defaults for the remaining screens of the wizard, and then use Windows Explorer to create a new folder below GoogleProxy named **bin**.

Next, open your text editor and create the following file, all on one line:

```
<%@ WebService Language="c#"
    Codebehind="GoogleProxy.asmx.cs" Class="Wrox.Services.GoogleProxyService" %>
```

Save this as `GoogleProxy.asmx` in the GoogleProxy directory.

Now create the main file, `GoogleProxy.asmx.cs`:

```
using System;
using System.Web;
using System.Web.Services;
using GoogleService;

namespace Wrox.Services
{
  [WebService (Description = "Enables calls to the Google API",
               Namespace = "http://www.wrox.com/services/googleProxy")]
  public class GoogleProxyService : System.Web.Services.WebService
  {
    readonly string GoogleKey = "EwVqJPJQFHL4inHoIQMEP9jExTpcf/KG";

    [WebMethod(
     Description = "Returns Google spelling suggestion for a given phrase.")]
    public string doSpellingSuggestion(string Phrase)
    {
      GoogleSearchService s = new GoogleSearchService();
      s.Url = "http://api.google.com/search/beta2";
      string suggestion = "";
      try
      {
        suggestion = s.doSpellingSuggestion(GoogleKey, Phrase);
      }
```

```
        catch(Exception Ex)
        {
          throw Ex;
        }
        if (suggestion == null) suggestion = "No suggestion found.";
        return suggestion;
      }
    }
}
```

Remember to enter the value of your Google key for the `GoogleKey` variable, and then save this file to the same location as the others.

The code itself is fairly straightforward; all the real work is done by the `GoogleSearchService`. The method `doSpellingSuggestion` creates an instance of the `GoogleSearchService` class. The URL of the service is then set. This step is not always necessary, but we've found that it often helps to be able to change the URL of services easily. In a production environment the URL would be read from a configuration file, enabling you to move between servers easily.

The `doSpellingSuggestion` method is now called, passing in the Google key and the `Phrase` argument. This is another advantage of using a server-side proxy: you can keep sensitive information such as your key away from the client-side environment of the browser.

If an exception is thrown, it will be re-thrown and returned as a SOAP fault. If the returned `suggestion` is `null`, a suitable string is returned; otherwise, the `suggestion` is passed back directly.

You now create the class to interact with Google. Begin by copying the `GoogleSearch.wsdl` file into the GoogleProxy folder, and then open a command prompt, navigate to GoogleProxy, and run the following (you can ignore warnings about a missing schema):

```
WSDL /namespace:GoogleService /out:GoogleSearchService.cs GoogleSearch.wsdl
```

The WSDL utility reads the `GoogleSearch.wsdl` file and creates the source code for a class to communicate with the service. The class will reside in the `GoogleService` namespace, as instructed by the first parameter to WSDL. This source code needs to be turned into a DLL and to do this you need to use the C# compiler as you did before. Enter the following at the command prompt or use the batch file named `MakeGoogleServiceDLL.bat` from the code download:

> `WSDL.exe` **seems to be installed in a number of different places, depending on what other Microsoft components are on the machine. On machines with Visual Studio .NET installed, it is likely to be in** `C:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin`**, but you may need to search for it on your machine.**

```
C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\csc.exe /r:System.dll
/r:System.Web.dll /r:System.Web.Services.dll /t:library
/out:bin\GoogleSearchService.dll  GoogleSearchService.cs
```

As before, the `/r:` parameters are telling the compiler which DLLs are needed to provide support classes to the target DLL of `GoogleSearchService`.

The last stage is to compile the `GoogleProxy` class itself:

```
C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\csc.exe /r:System.dll
/r:System.Web.dll /r:System.Web.Services.dll /r:bin\GoogleSearchService.dll
/t:library /out:bin\GoogleProxy.dll GoogleProxy.asmx.cs
```

Notice that a reference is passed in for the `GoogleSearchService.dll` just created.

You are now ready to test the service by entering the following URL in your browser:

```
http://localhost/GoogleProxy/GoogleProxy.asmx
```

You should be greeted with the standard screen, as shown in Figure 6-8.

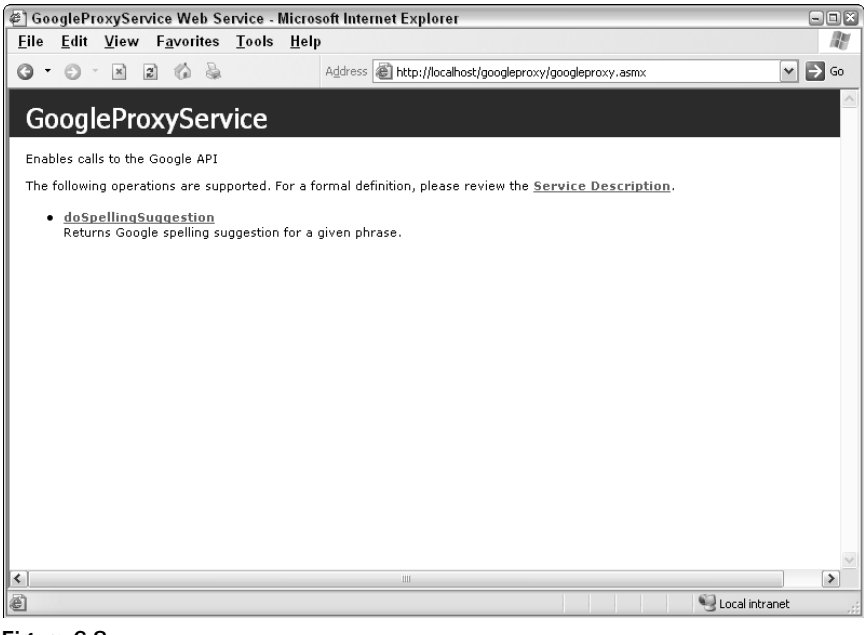

Figure 6-8

Click the doSpellingSuggestion link to try out the method using the built-in test harness, as shown in Figure 6-9.

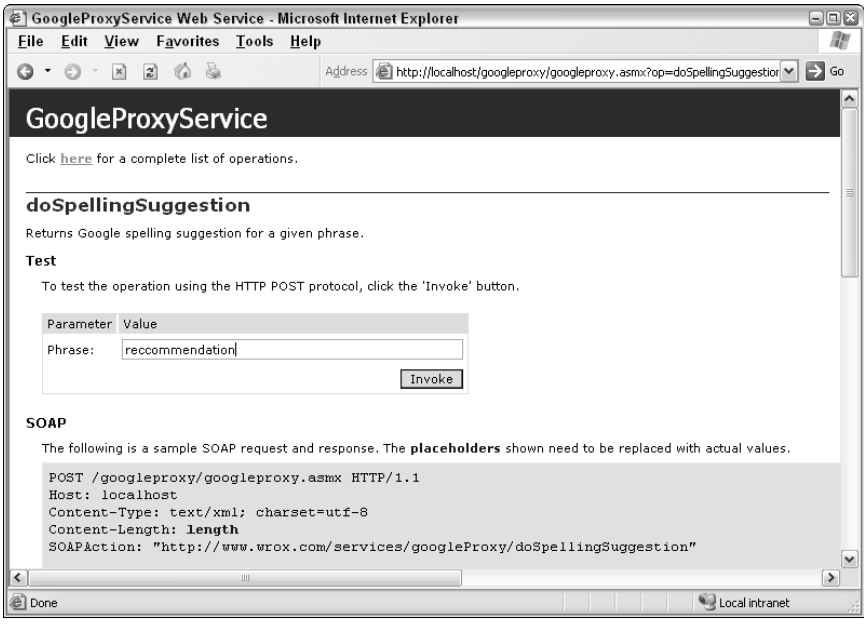

Figure 6-9

When you click the Invoke button, you will see the XML returned, as shown in Figure 6-10.

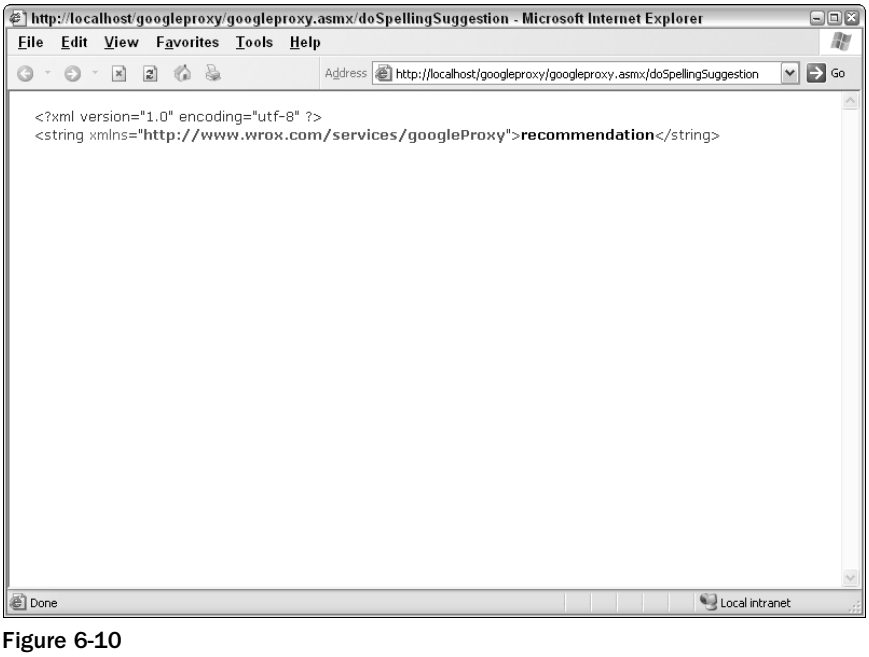

Figure 6-10

Summary

This chapter introduced you to the concept of web services, an architecture allowing the transfer of data from one location on the Internet to another. You learned about the evolution of web services and associated technologies, such as SOAP, WSDL, and REST. The similarities and differences between SOAP and REST services were also discussed.

Next, you learned how to create your own web service using ASP.NET and C#. This involved downloading the .NET SDK and using the built-in web service creation and management tools. You learned how to inspect and test your web service using the generated .NET test harness.

You then moved on to create a test harness client for Internet Explorer, and then one for Mozilla, using different techniques to call the web service. You were introduced to the web service behavior for Internet Explorer and the high-level SOAP classes in Mozilla. The last test harness created is intended to be universal, using XMLHttp to send and receive SOAP messages.

Last, you learned about cross-domain issues with web services and how to avoid them using a server-side proxy.

In this chapter the SOAP specifications were used to pass arguments between the client and the server. Although this is a robust and flexible method, it adds a lot of overhead to the process and means that the client must be able to handle XML and all its attendant complexity. The next chapter shows a simpler and less formal way of passing data between machines called JavaScript Object Notation (JSON).