Chapter 1: Getting Acquainted with Web Services

In This Chapter
✓ Exploring the basic principles of service-oriented apps
✓ Creating service-oriented apps
✓ Making XML Web services available

A web service is just the provision of functionality over the Internet using an open interface. A web page provides functionality that you can see; a web service provides the underlying data, in a format that you can use in another application. Web services are a data source — in a way, much like files and databases — but on the web.

Web services are straightforward — at least until the software manufacturers start messing with them. Web services are standards driven, just as HTML is, and the World Wide Web Consortium (W3C) owns its documentation. Web services have been a hot topic for the past decade, but only in the past five years or so (in step with the ubiquitous nature of the Internet) have they become a viable option for the delivery of hard-to-find software functionality.

A few different web services formats exist in the .NET world, and they solve two basic problems:

✦ Making part of your application available past the physical boundary of the application.
✦ Making a distributed middle to your application so that you can scale paragraphs if your site suddenly has a lot of traffic.

After covering a few web service principles, I present some code to show you how each format works.

Understanding Web Services

Web services provide a way to extend methods past the normal boundary of a software system. You usually write something like the following chunk of code in order to start building a program:
public bool AddStuff(String stuff) {
    //Add it here
    return true;
}

and then you call it like this:

bool DidItWork = AddStuff("This is the new Stuff");

Using web services, however, you call the function this way:

POST /Service1.asmx HTTP/1.1
Host: localhost
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length
<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
    <soap12:Body>
        <AddStuff xmlns="http://tempuri.org/">
            <stuff>string</stuff>
        </AddStuff>
    </soap12:Body>
</soap12:Envelope>

and then see a response like this one:

HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length
<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
    <soap12:Body>
        <AddStuffResponse xmlns="http://tempuri.org/">
            <AddStuffResult>boolean</AddStuffResult>
        </AddStuffResponse>
    </soap12:Body>
</soap12:Envelope>

Whoa! Hey, why would anyone want to build a function that way? I’m glad you asked. In the later section “Building Service-Oriented Applications,” I answer your questions. For now, I describe some concepts that make building web services easier to handle.

Three basic principles define web services:

✦ **Loosely coupled:** They don’t require a constant connection to the server
✦ **Contract driven:** They provide an interface that describes all of their functionality
✦ **More likely to be chunky, not chatty:** Rather than lots of properties with single values, they provide big methods that return collections.
I discuss these topics more fully in the sections that follow.

**Loosely coupled**

Because web services are, like web applications, *loosely coupled*, web service conversations aren’t guaranteed to make sense. You might get a sentence from three minutes ago, after four other sentences have gone past, or you might not hear from them again after the first line.

When you stop to think about it, loose coupling makes sense: Web service calls are just like navigation in a web page. Sometimes you click a link and then close your browser, and sometimes you click the link twice. Web applications are up to the whim of the user, and so are web services.

For this reason, a client of a web service must be *loosely coupled* to the service. For example, you don’t want to make a user wait until a web service call is complete. You must call ‘x’ asynchronously and have the result show up when it is ready.

Fortunately, .NET handles the asynchronous call. You just tell your application that you’re calling a service asynchronously and then provide a delegate for the service to call when it’s ready. As long as you handle the code properly, the service will work as expected.

In this example, you can see that we implemented the loose coupling with an asynchronous call:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            //Here YOU are calling the service asynchronously.
            AddStuffReference.Service1SoapClient client =
                new AddStuffReference.Service1SoapClient();
            client.AddStuffCompleted += new EventHandler(client_
                AddStuffCompleted);
            client.AddStuffAsynch("This is the stuff");
        }

        //This method is called when the response comes back.
        //No timers or anything.  .NET handles it for you.
        void proxy_AddStuffCompleted(object sender, AddStuffReference.
            Service1SoapClient.AddStuffCompletedEventArgs e)
        {
            string result = e.Result.ToString();
        }
    }
}
```
The preceding example only protects your user interface from experiencing a tie-up. You still have no real indication that the messages will ever be delivered, so you should never write software that depends on the delivery of the data from the service. It has to fail gracefully.

**Contract driven**

In client-server development, an interface defines a contract of sorts between the domain model and the user interface. You can use that contract to drive development in the same way that you use a contract to drive a business deal. If you know what I require and I know what you require, we can quickly and easily make a deal.

A web service is required to have an interface that conforms to the Web Services Description Language (WSDL) standard. WSDL describes expected inputs and allowed outputs just as an interface would, creating a contract between the service provider and the client.

In .NET, your contract is created automatically from the code for the service. If you call a service in the browser (which isn’t the way it’s designed to be called — I tell you more about that topic later in this minibook) from the development machine, you see the test screen shown in Figure 1-1. Even though the browser gains some basic important information about the service, it isn’t the contract. The contract is shown behind the Service Description link.
Clicking the Service Description link appends the text `?WSDL` to the URI, and the web browser shows the contract. The client system used this contract to determine exactly which information the service wants and how it will respond to input. Though the WSDL is displayed in a browser, as shown in Figure 1-2, the interesting part is the XML that’s behind the screen.

For compilable languages, the client machine does essentially the same thing no matter which platform it’s on (Java or Basic or PHP or Ruby): At compile time, it reads the WSDL and creates a proxy that the client talks to. This

✦ Brokers the communication process between the client and the service
✦ Provides type safety, if it’s supported
✦ Generally makes your life easier because WSDL (and the contract it provides) is an important part of a service developer’s work

The contract isn’t legally binding (and contains only a small amount of fine print). The service provider reserves the right to change the service any time it wants, and can even forgo updating the WSDL with no fear of legal reprisal. Because you’re a .NET developer, though, your WSDL is generated automatically.

**Chunky versus chatty**

Although the chunky-versus-chatty services issue might sound like a face-off between candy bars, it isn’t.
Rather than perform small, incremental operations via web services, you use them to make large, sweeping “strokes.” The length of individual calls might be larger, but the number of communications is fewer, which reduces network overhead.

Suppose that an application changes the settings of a piece of hardware located miles away from it — perhaps your home heating system (which is a good use of a service). The client application (your computer) is absolutely local, the remote device (your heater) is certainly remote, and a network (probably the Internet) is in the middle. If you have a service with a series of individual controls, such as `TurnFanOn` and `TurnFanOff`, it’s a common interface for a local application, such as connecting a heater directly to your computer.

The following chunk of code gives an example of that chatty sort of interface:

```csharp
namespace HomeHeater
{
    public class Chatty : IChatty
    {
        public bool TurnFanOn()
        {
            return true;
        }
        public bool TurnFanOff()
        {
            return true;
        }
        public bool SetTemperature(int newTemperature)
        {
            return true;
        }
        public bool SetFanSpeed(string newFanSpeed)
        {
            return true;
        }
    }
}
```

The interface in this example is *chatty* (in case you couldn’t tell from its class name). Every time you change a knob on the controller, a call is made to a service. Move the temperature to 72 degrees and call the service. Turn the fan to its High setting and call the service. Turn the temperature back down to 71 degrees and call the service. The client “chats” with the service.

Nothing is intrinsically wrong with this implementation. For a service, though, with more network overhead for every call, it isn’t the best way to build methods. Instead, you want your client to change settings and then pull a big lever on the side to make all the changes at one time.

A *chunky* interface provides a domain model for the client to use, with properties to set. Then, after all settings are in the model object, you send the whole shootin’ match to the service. It looks like this:
namespace HomeHeater
{
    public class Chunky : IChunky
    {
        public bool UpdateHeaterSettings(HeaterModule heaterModule)
        {
            return true;
        }
    }
    public class HeaterModule
    {
        public int Temperature { get; set; }
        public bool FanOn { get; set; }
        public string FanSpeed { get; set; }
    }
}

Now, no matter how often you make changes, the service is called only when you “pull the big lever” (update it). Whether this action prevents users from updating after every change depends on your interface. In general, this design principle is the best one for service development.

You may wonder why anyone would bother with a concept as convoluted as web services. I can assure you that good reasons exist and that you will want to use this technology on a project.

**Building Service-Oriented Applications**

The first and most obvious use of a service is to create a Service-Oriented Application, or SOA. This overloaded, overused term means that your application (like the heating system I mention in the previous section) uses services to assist a remote client with server communications.

Unfortunately, the term SOA is no longer useful in the marketplace. So many so-called “experts” have chimed in with large, unmanageable ideas for the creation of systemwise services that it is impossible for anyone to separate the wheat from the chaff.

The concept is simple: The user interface calls a service at some point in the process to communicate with the server. You do this for two reasons:

- **Scalability:** It’s the main reason to use a service inside an application — especially a web application.

  Up to a point, most web applications have built-in scaling. If you need more access points, you just add more servers and then use a device to sort the traffic to another machine, as shown in Figure 1-3.
Web applications are different, though, because layers have different scalability needs. Sometimes the database is loaded, and sometimes the web server is. You must be able to separate the layers of the application, as shown in Figure 1-4 — that’s where services start to become useful.

Because the services use a common format — usually XML over HTTP — you can relatively easily install parts of the application on their own machines to isolate them physically. Because the functionality of the deleted part is called via a service, you can scale the application horizontally — from one to multiple servers.

✦ **Reusability:** Every organization has a list of its participants — clients, users, voters, cooks, or mailing list subscribers, for example. Regardless of the type of participant, all the people on the list have first and last names and other identifying characteristics.
It seems like nearly every application written today has a table of People. Savvy programmers use slick tricks to keep these tables in sync or to share information, for example, but only one way exists to share the People table — by using a data silo, as shown in Figure 1-5.

![Figure 1-5: A data silo.](image)

A data silo works this way:

1. A database on some server somewhere contains all participants’ demographic information.
2. The database is surrounded by a service layer containing all allowed operations.
3. The service layer is consumed by all other applications in the system, as shown in Figure 1-5.

This set of steps shows you the concept of reuse. It isn’t about bits of code (no matter what you might hear) — it’s about data. Services help to provide access to the data silo.

**Providing XML Web Services**

A common use of services is to give other programmers public access to your information by using an open API that can be consumed anywhere. You might recall the WSDL file, described in the earlier section “Contract driven” — it can make your cool function or valuable data available to anyone who needs it.
In the demographic silo example, described in the earlier section “Building Service-Oriented Applications,” if you have a valuable mailing list and you want to give your customers access to it, you can send them the list. If you do, however, they can use it forever. Instead, suppose that you could bill your customers every time they use the list. If you provide web services to implemented functions on the list, you can then track actual usage of the list.

You can see this concept, known as provision past the boundary, everywhere. Take Twitter, for example. Rather than making users go to the web page to search through billions of messages, Twitter provides a WSDL contract that gives developers the ability to search in their own applications.

Web services can be controlled like any other web application can. The field-level or operation-level security that used to be handled at the database level can now be built into a semipublic API.

**Building Three Sample Apps**

You can build services in C# in a few ways, as described in the next three chapters. This section’s heading isn’t exactly right, however: I show you how to build the same app in three different ways. The app, the SHARP conference management system (or a simple version of it), was built using these formats:

- **ASMX**: This first version is in the venerable ASMX format: ASP.NET web services. Though this format has been largely superseded by WCF, it’s still available. I cover it in this chapter and the next because it’s a viable solution for certain situations.

- **WCF**: Windows Communication Foundation (WCF), the most important service platform, is given enough screen time here to get you started. The topic is broad, and you can find lots of references to other resources.

- **REST**: REpresentational State Transfer (REST), the oldest service protocol on the planet, has a new life, thanks to Asynchronous JavaScript and XML, or AJAX. REST isn’t a first-class citizen in the .NET world, but it’s important. I show you how to use a WCF project template to design and build effective REST services.

The entire process uses a single entity data model, shown in Figure 1-6, and provides allowed operations to it.
The idea is to provide a service model similar to the heating system platform described earlier in this chapter, except using a real, stateful, data-driven application.

Finally, I show you how to solve three different problems. Then you can shoot for three different outcomes. Otherwise, what fun would it be?