

IN THIS CHAPTER

- » Getting acquainted with SOAP
- » Creating an ASMX service
- » Running an ASMX service
- » Testing your service by using a web page and an application

Online Chapter **5**

Building Web Services with ASMX

Online Chapter 4 states that services provide access to functional code over the wire. Though it wasn't covered, it probably became obvious that one can't just call a method on a remote machine without some kind of wrapper. The black magic that makes .NET methods work in a client program doesn't work over the Internet.

There have been a bunch of remote procedure call (RPC) protocols (as they are called) over the years. Some you might be familiar with include

- » CORBA
- » DCOM
- » RCW
- » OpenBinder
- » LINX
- » DLPI
- » STREAMS

- » DDE
- » Even AJAX, in its own way.

The benefit to using services is that they are based on a

- » **Standard:** Using standards means that everyone follows the same path to working with the services. The technique used to access the service follows particular rules.
- » **Human readable:** Accessing a service usually requires the use of a human-language-like query that is easier to interpret than the binary calls used by older standards.
- » **Extendable:** Relying on services may mean less work on your part. Many services rely on other services to perform tasks. These nested services build on each other to provide the required support.
- » **Protocol:** Defining a protocol means creating a set of rules to perform specific tasks. They lend a formality to working with the service that makes the service easier to use and understand.

None of the other messaging protocols are all of those. They all have some small (or occasionally large) problem that prevents the benefits of RPC access from really shining. Web services provide what is actually needed. The first web service protocol to provide these benefits in usable form is the Simple Object Access Protocol (SOAP), which was introduced in 1998 by Dave Winer, Don Box, Bob Atkinson, and Mohsen Al-Ghosein for Microsoft. (REpresentational State Transfer, or REST, introduced in 2000 by Roy Fielding, is the second, but that is a story for a later chapter.)

Getting to Know SOAP

SOAP is an XML-based protocol for sending messages over the Internet, usually via HTTP. However, you can also use it with other protocols such as Simple Message Transfer Protocol (SMTP). You can think of it as an envelope for remote procedure calls because that is exactly what it is.

The major benefit to SOAP, aside from global acceptance and longevity, is the rich experience that it provides the client. There are a *lot* of developer features in SOAP, such as transactions and security, and they all work pretty well.

SOAP and standards

Standards are discussed throughout this book, but it bears discussing here, too. Here is how standards-based development works:

1. Either because of industry need or a company idea, some organization develops a standard. This is usually a recognized organization like the International Organization for Standardization (ISO) or the World Wide Web Consortium (W3C). If the organization isn't large enough, a large company, such as Microsoft or IBM, could sponsor the standard.
2. The standard is distributed to the standards community for review.
3. After community acceptance, the standard is certified by the host standards organization.
4. Some company, when developing a product, realizes that it needs a feature that happens to be described by that standard.
5. After reviewing the standard, the company decides to implement the standard.

Realistically, regardless of certification, only when a large number of companies implement a standard does it actually become a genuine standard. Many so-called web browsers over the years supported standard protocols like Virtual Reality Modeling Language (VRML) and such that never made it.

The WS-* standards

The WS-* (usually pronounced WS-*star*, referring to the wildcard * character that refers to all standards that begin with WS) standards fit right in here. These web service standards apply to protocols like SOAP. They're written by a standards organization, reviewed, certified, and used by the tool developers — like Microsoft.

Even better, the additional standards include a lot of neat functionality that makes SOAP a rich development experience. Usually, distributed communication standards leave transactions, security, and other such functionality up to the developer. SOAP isn't like that. It is supposed to have all that stuff baked in. The WS-* standards include all this useful functionality (and more):

- » **Web Services Transactions (WS-TX):** Coordinates the outcome of broadly distributed communications.
- » **Web Services Reliable Exchange (WS-RX):** Provides a confirmation of communication for service calls.

- » **Web Service Federation (WS-FED):** Allows for a federation of trust between service providers.
- » **Web Service Remote Portlets (WS-RP):** A standard for web parts using services (as you see in SharePoint).
- » **Web Service Security (WS-SX):** A supported trusted exchange.
- » **Web Services Discovery (WS-DD):** A way to find services in a large enterprise.
- » **Building Information Exchange (oBIX):** Allows buildings to talk to each other about their wiring. No, we're *not* kidding!
- » **OASIS ebXML:** A business XML standard that is designed to provide a standardized data model for communications.

In general, these are fantastic additions. They define a set of functionality that all web service development software providers — Microsoft, IBM, Sun, open source initiatives, whomever — can implement. If you need transactions, they are there. Security? Baked in.

The impact to you

A problem arises from the fact that the standards discussed in the preceding section were used differently by every company that implemented them. The problem with standards this detailed in scope is that in order to be useful, they must either leave a lot to the imagination or define everything. The Organization for the Advancement of Structured Information Standards (OASIS) erred on the side of being too loose, and the implementations are a mess.

If you're working inside the Microsoft stack — meaning, you're communicating with other .NET projects — you are golden. Within the Microsoft platform, everything is defined the same. However, if you're communicating outside the .NET Framework — say, with IBM or Sun — you should expect problems if you're using WS-* defined functionality.

The take-home is that SOAP includes a lot of standard functionality that isn't found anywhere else. Sure, some distributed message contracts have a lot of features found in SOAP, and SOAP might not be completely implemented the same everywhere, but there isn't anything else that even tries to provide this functionality in a standardized way.

From this perspective, SOAP is a fantastic platform. Need transactions, especially *secure* transactions? SOAP has them. Need large binary attachments? SOAP has that. Have Business Process Execution Language requirements? There's a SOAP for that.

Big, fat, and slow

All this eating at the trough of standardized features has made SOAP, well, a little large-boned. Let me give you an example. The XML required just to set the context for transactions (the service equivalent of a cookie) is

```
<wscoor:CoordinationContext
  xmlns:wsa="http://www.w3.org/2005/08/addressing"
  xmlns:wscoor="http://docs.oasis-open.org/
    ws-tx/wscoor/2006/06"
  xmlns:myApp="http://www.example.com/myApp"
  S11:mustUnderstand="true">
  <wscoor:Identifier>
    http://Fabrikam123.com/SS/1234
  </wscoor:Identifier>
  <wscoor:Expires>3000</wscoor:Expires>
  <wscoor:CoordinationType>
    http://docs.oasis-open.org/ws-tx/wsat/2006/06
  </wscoor:CoordinationType>
  <wscoor:RegistrationService>
    <wsa:Address>
      http://Business456.com/
        mycoordinationsservice/registration
    </wsa:Address>
    <wsa:ReferenceParameters>
      <myApp:BetaMark> ... </myApp:BetaMark>
      <myApp:EBDCode> ... </myApp:EBDCode>
    </wsa:ReferenceParameters>
    </wscoor:RegistrationService>
    <myApp:IsolationLevel>
      RepeatableRead
    </myApp:IsolationLevel>
  </wscoor:CoordinationContext>
```

In the Java world, 10MB SOAP messages are not uncommon — although that includes the payload — and that works out to, what, 200,000 lines? In one HTTP call? That's probably a bit much.

SOAP dramatically increases your overhead in communication. If you don't need the security, federation, and transaction capabilities of SOAP, consider REST — covered in Online Chapter 7. If you're communicating in a homogenous Microsoft environment, consider binary encoding. On the other hand, if you're communicating in moderately heterogeneous environments and do in fact need an encrypted, federated transaction using a common enterprise data model, by all means look at SOAP.

Making an ASMX Service

You can write a SOAP service in ASMX or Windows Communication Foundation (WCF). The ASMX moniker isn't an abbreviation; it doesn't stand for anything. It's an ASP.NET web file extension that provides a service rather than a web page. ASMX allows only for SOAP output, while WCF can also create other kinds of output. The following sections cover ASMX; Online Chapter 6 demonstrates techniques for writing in WCF.

Creating a new service

ASMX web services are part of ASP.NET website projects. As such, they seem a lot like ASP.NET web applications — because they *are* ASP.NET web applications. You can actually put an ASMX file in a regular ASP.NET application, and a web file in a web service application, as we do in the following step list. The template is just there to help you get started.

- 1. Choose File ⇨ New ⇨ Project.**

You see the New Project dialog box.

- 2. Select the Visual C# Web folder.**

Visual Studio displays the list of projects.

- 3. Highlight the ASP.NET Web Application (.NET Framework) project type.**

- 4. Type ANewService in the Name field, type ServiceApp in the Solution field, and click OK.**

Using different names for the project name and the solution name makes it easier to add other projects to the solution. In this case, you need a second project to test the service. The wizard displays the New ASP.NET Web Application dialog box.

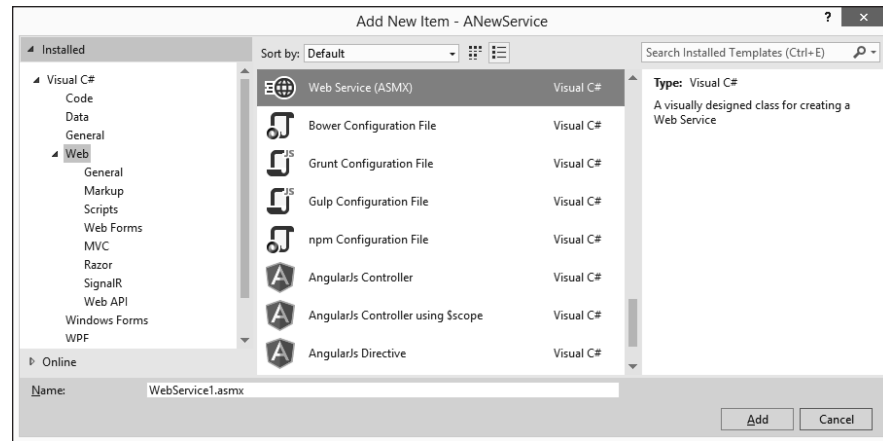
- 5. Select the Empty template and click OK.**

The Empty template creates an empty project that you can use to create whatever sort of project you want. You can still add folders and core references used for any of the other template types, as well as add unit tests and security as needed. The Empty template simply gives you the flexibility to decide what your project contains, or whether it contains anything at all when you start it.

Analyzing the file setup

Right-click the project and choose Add ➤ New Item. Select the Web folder to see the Add New Item dialog box, shown in Figure OC5-1. The dialog box has all the usual suspects. Near the end is the Web Service (ASMX) item, which is what you want. Select it and name it `TheService.asmx`.

FIGURE OC5-1:
Add a new web service to the project.



Note that although `TheService.asmx` (and the usual code-behind file) is created, you don't actually see the markup file. Visual Studio automatically opens the `TheService.asmx.cs` file for you. This is by design. Nothing goes in the markup file. All the magic is in the code-behind.

Breaking down the sample code

The `TheService.asmx.cs` file does all the work in the new service, starting with the code in the template. The following text runs through that code a line at a time (the unneeded using statements are already removed):

```
using System.Web.Services;

namespace ANewService
{
    /// <summary>
    /// Summary description for TheService
    /// </summary>
    [WebService(Namespace = "http://tempuri.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [System.ComponentModel.ToolboxItem(false)]
```

```
// To allow this Web Service to be called from script, using ASP.NET AJAX,
// uncomment the following line.
// [System.Web.Script.Services.ScriptService]
public class TheService : System.Web.Services.WebService
{
    [WebMethod]
    public string HelloWorld()
    {
        return "Hello World";
    }
}
```

There are eight things to point out:

- » Line 1 brings in the `System.Web.Services` namespace; it's essential to the rest of the template.
- » Line 8 sets the Namespace for the service. The namespace, as with the namespace of your .NET classes, can be anything you want. It doesn't refer to a real place on the web.
- » Line 9 sets the Web Service Binding. WS-I is the Web Services Interoperability organization who (you guessed it) sets *even more* standards for web services. Basic Profile 1.1 is more or less the industry standard. You can find out more at <http://www.ws-i.org/>.
- » Line 10 declares whether the item should show up in design time Toolboxes.
- » Line 12 (when uncommented) activates runtime support for AJAX.
- » Line 13 is a normal, everyday class definition, but notice that it inherits `System.Web.Services.WebService`.
- » On Line 16, you can see the `[WebMethod]` attribute. This allows you to control what methods are available to the service from the class, and there are a few attributes for fine-grained control.
- » The code itself is fairly boring — the functional code is exactly the same as it would be in a normal class. All the things that make it a web service are in the declaration.

Adding some code

Obviously, a web service that outputs Hello World isn't particularly useful. Then again, it's important not to bury the functionality of this first web service in too much complexity. Replace the `HelloWorld()` method with the following method,

which shows how to perform input and output with a WebMethod (something that most WebMethods do).

```
[WebMethod]
public int DoAdd(int Value1, int Value2)
{
    return Value1 + Value2;
}
```

In addition, you normally replace the namespace with something else to keep your site secure. Change the namespace URL from "http://tempuri.org/" to "http://www.mysite.com/". Obviously, on a real project, you would use the namespace for your organization.

Running the service

Press F5 and run the service. If all goes well, you should see something like Figure OC5-2. This is how your web service should look when you have done everything correctly. Remember that you didn't make this user interface; IIS did this for you, to make it easier to test. External users can't see this.

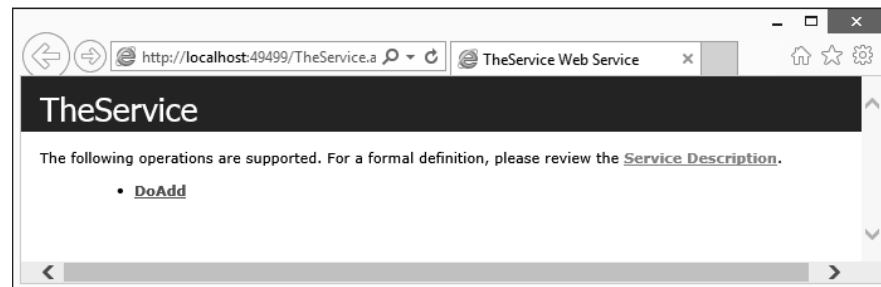
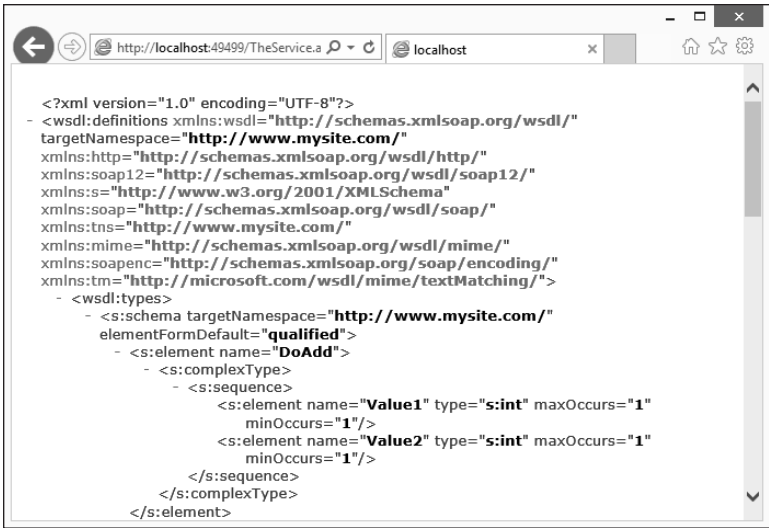


FIGURE OC5-2:
Running the
default service.

Clicking the Service Description link gives you the WSDL for the whole service, as shown in Figure OC5-3. This is what an application needs to build its proxy. Note in the browser's address bar that all it does is append a ?WSDL to the end of the URL.

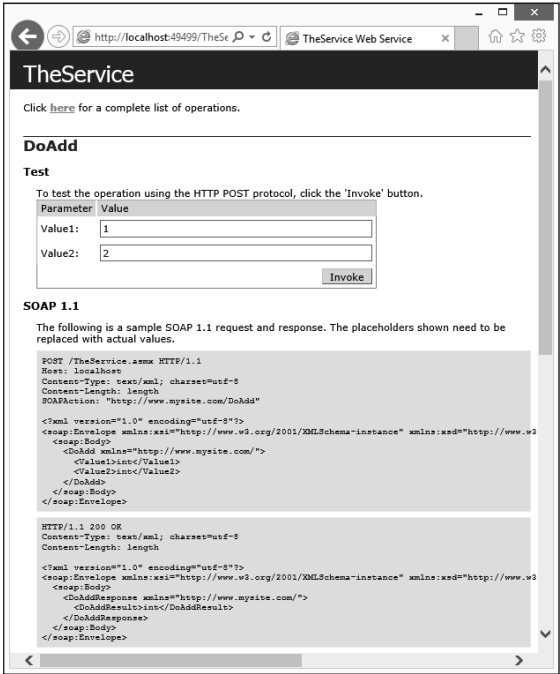
Click the DoAdd link to see the test page shown in Figure OC5-4. Below the test entries, you see the SOAP 1.1 and SOAP 1.2 code used to access your web service. You will have one of these links for every one of the methods in the class marked with WebMethod.

FIGURE OC5-3:
The WSDL describes the web service to external users.



```
<?xml version="1.0" encoding="UTF-8"?>
- <wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://www.mysite.com/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.mysite.com/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/">
  - <wsdl:types>
    - <s:schema targetNamespace="http://www.mysite.com/"
      elementFormDefault="qualified">
      - <s:element name="DoAdd">
        - <s:complexType>
          - <s:sequence>
            <s:element name="Value1" type="s:int" maxOccurs="1"
              minOccurs="1"/>
            <s:element name="Value2" type="s:int" maxOccurs="1"
              minOccurs="1"/>
          </s:sequence>
        </s:complexType>
      </s:element>
```

FIGURE OC5-4:
Use the test page to check web service functionality.



TheService

Click [here](#) for a complete list of operations.

DoAdd

Test

To test the operation using the HTTP POST protocol, click the 'Invoke' button.

Parameter	Value
Value1:	<input type="text" value="1"/>
Value2:	<input type="text" value="2"/>

SOAP 1.1

The following is a sample SOAP 1.1 request and response. The placeholders shown need to be replaced with actual values.

```
POST /TheService.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.mysite.com/DoAdd"

<?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>
    <DoAdd xmlns="http://www.mysite.com/">
      <Value1>int</Value1>
      <Value2>int</Value2>
    </DoAdd>
  </soap:Body>
</soap:Envelope>

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?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>
    <DoAddResponse xmlns="http://www.mysite.com/">
      <DoAddResult>int</DoAddResult>
    </DoAddResponse>
  </soap:Body>
</soap:Envelope>
```

Type **1** in the Value1 field and type **2** in the Value2 field. Click Invoke. You see the expected XML output of

```
<?xml version="1.0" encoding="UTF-8"?>
<int xmlns="http://www.mysite.com/">3</int>
```

Building a test page

To truly see your web service at work, you need to create a test page. The test page may not be as fancy or as complete as the final application that you intend to use to access your web service, but it should demonstrate that the web service does, in fact, work. The following steps describe how to add a test page to the existing application:

- 1. Right-click the project entry in Solution Explorer and choose Add ➞ Web Form from the context menu.**

You see a Specify Name for Item dialog box.

- 2. Type TestPage and click OK.**

Visual Studio adds a new Web Form.

- 3. Change to Split view.**

There will be a default div that you find in Book 6, Chapter 2, in the book. However, unlike the Web Form in Book 6, Chapter 2, this one won't contain any added tags or text.

- 4. Add two TextBox controls and one Button as well as one Label control. Set the control properties as follows:**

- TextBox1: ID="Value1" and Text="1".
- TextBox2: ID="Value2" and Text="2".
- Button1: ID="Invoke" and Text="Invoke".
- Label1: ID="Result" and Text="0".

- 5. Double-click Invoke to display the code-behind.**

- 6. Right-click References in Server Explorer and choose Add Service Reference.**

You see the Add Service Reference dialog box.

- 7. Click Discover.**

Visual Studio discovers TheService.asmx, as shown in Figure OC5-5. Note that when you drill down, you see the complete details of the service, including the DoAdd operation.

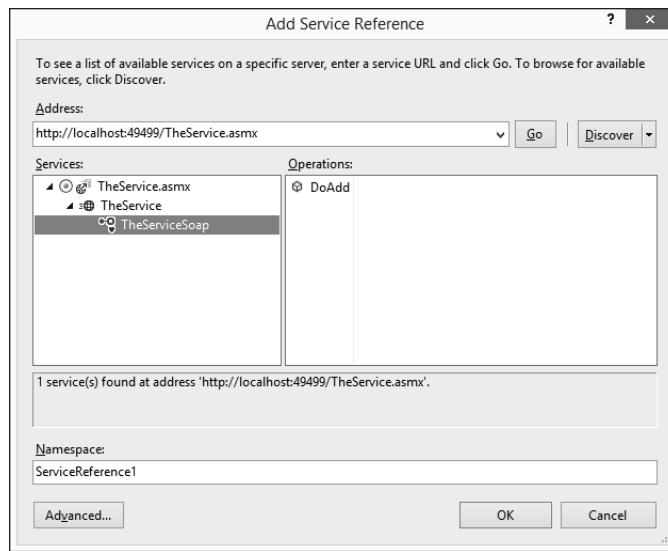


FIGURE OC5-5:
Add the web
service reference
to your web
page to use it.

8. Type UseTheService in the Namespace field and click OK.

The test page can now access the service.

9. Add the following code to the Invoke_Click() method:

```
protected void Invoke_Click(object sender, EventArgs e)
{
    // Create the client.
    UseTheService.TheServiceSoapClient Client =
        new UseTheService.TheServiceSoapClient();

    // Use the client to invoke the web service.
    Result.Text = Client.DoAdd(
        Int32.Parse(Value1.Text),
        Int32.Parse(Value2.Text)).ToString();
}
```

The process for accessing a web service can vary, but this example is typical. You create a client and then call methods through that client to perform tasks using the web service.

10. Right-click the TestPage.aspx entry in Solution Explorer and choose Set As Start Page.

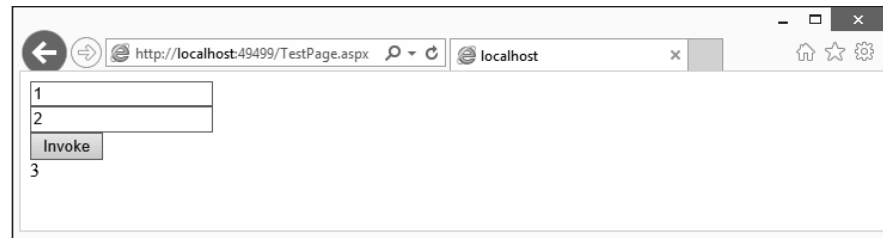
11. Press F5 to run the application.

You see the application start in the browser.

12. Click Invoke.

Figure OC5-6 shows typical output.

FIGURE OC5-6:
This web page
relies on the
web service to
perform the
calculation.



Consuming services in your applications

Consuming an ASMX service works just like consuming any other service, from any provider. The following steps show how to consume the service in a console application, but the same technique applies to any other desktop application type.

- 1. Right-click the solution in Solution Explorer and choose Add ➤ New Project from the context menu.**
You see the Add New Project dialog box.
- 2. Choose the Visual C#>Windows Classic Desktop folder and select the Console App (.NET Framework).**
- 3. Type TestApp in the Name field and click OK.**
Visual Studio creates a new console application for you.
- 4. Right-click the References folder in Solution Explorer and choose Add Service Reference.**
You see the Add Service Reference dialog box.
- 5. Click Discover.**
You see the same information as that shown in Figure OC5 2-5.
- 6. Type UseTheService in the Namespace field and click OK.**
The wizard adds the required reference.
- 7. Type the following code into the Main() method:**

```
static void Main(string[] args)
{
    // Create the service client.
    UseTheService.TheServiceSoapClient Client =
        new UseTheService.TheServiceSoapClient();

    // Get the input values.
    Console.WriteLine("Type a Value1 input: ");
    int Value1 = Int32.Parse(Console.ReadLine());
```

```

Console.Write("Type a Value2 input: ");
int Value2 = Int32.Parse(Console.ReadLine());

// Perform the computation and provide output.
Console.WriteLine(Client.DoAdd(Value1, Value2));
}

```

8. Right-click `TheService.asmx` in Solution Explorer and choose Set As Startup Page.

You must start the service before you can use it in the application.

9. Choose Debug ⇄ Start Without Debugging.

The service will start. Starting the service without debugging makes it easier to keep the service running while you need it. When you finish using the service, right-click the IIS Express icon in the Notification Area and choose Exit from the context menu.

10. Right-click the `TestApp` project in Solution Explorer and choose Set As Startup Project from the context menu.

11. Choose Debug ⇄ Start Without Debugging.

You see the initial dialog box.

12. Type values for each of the inputs and press Enter after each input.

The application outputs a result similar to the one shown in Figure OC5-7.

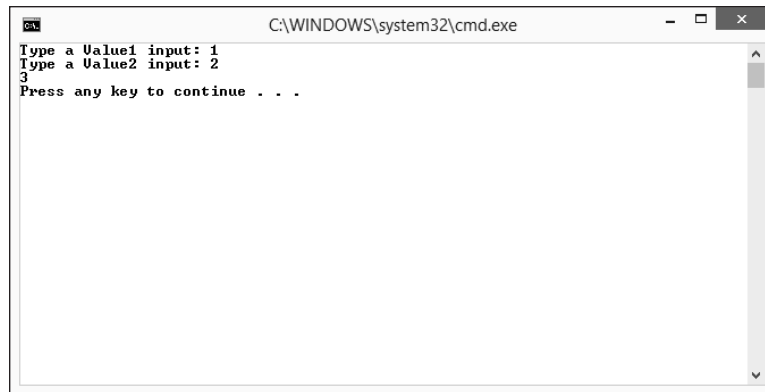


FIGURE OC5-7:
Console applications can easily consume your web services.