IN THIS CHAPTER

» Exploring the principles of REST

» Implementing REST services using ASP.NET Core

» Consuming REST services

Online Chapter 7

Building Web Services with REST

In contrast to some of the other methods of transferring data that are described in this book, such as Simple Object Access Protocol (SOAP), REpresentational State Transfer (REST) is an architectural style. The biggest problem with REST is that most people really don't understand it. From the perspective of this chapter, REST is simply a means of defining an interface between two entities by using six constraints, which differs greatly from the specific rules required by protocols. Because this is a book about C# and the example relies on HTTP as the underlying protocol, the view of REST that you find in this chapter is quite specific. You can obtain a more general view of REST at `http://www.restapitutorial.com/lessons/whatisrest.html` and `https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`.

What you do find in this chapter is the C# view of REST. This chapter introduces you to REST and guides you through its advantages and drawbacks. You also create your own RESTful service and test it by using the same approach as Online Chapters 5 and 6 (so that you can make comparisons).

58_9781119428114-ancmat07.indd 1 Trim size: 7.375 in × 9.25 in December 30, 2017 10:15 AM

Getting to Know REST

REST, as described in this book, is basically the use of the traditional GET and POST patterns the old folks will remember from CGI. For you young pups, it is the basic format of web requests. For instance, when you click a link that looks like this:

```
http://mydomain.com/start.aspx?id=3
```

you're using REST. This isn't an implementation; rather, it's a remote procedure call mechanism. It's just a way to get parameters for a query to a remote machine and to get data back. REST is an architecture that has the following guidelines. A REST interface has four goals:

- Scalability of component interactions
- Generality of interfaces
- Independent deployment of components
- Intermediary components to reduce latency, enforce security, and encapsulate legacy systems

Use of REST with Windows Communication Foundation (WCF) meets most, but not all, of those goals. This chapter takes a quick look at the details of the implementation of REST in WCF, and you can make your own call.

Understanding the Guiding Principles of REST

REST has four guiding principles. According to the standard, all RESTful interfaces must provide interfaces that adhere to these principles. In the real world, compliance is up for discussion. Here are the principles:

TECHNICAL STUFF

- **Identification of resources:** Individual resources (like a data item, for instance) are identified in requests (for example, by using Uniform Resource Identifiers, or URIs, in web-based REST systems). The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server doesn't send its database, but rather, perhaps, some HTML, XML, or JSON that represents some database records expressed, for instance, in French and encoded in Unicode Transformation Format 8-bit (UTF-8), depending on the details of the request and the server implementation.

- **Manipulation of resources through these representations:** When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource on the server, provided that it has permission to do so.
- **Self-descriptive messages:** Each message includes enough information to describe how to process the message (for example, which parser to invoke). An example of this is the use of Internet media types, previously known as Multipurpose Internet Mail Extension (MIME) types. From the media type alone, the client must know how to process its contents. If it needs to look inside the message's contents to understand them, the message is not self-descriptive. For example, merely using the "application/xml" media type is not sufficient for knowing what to do with its contents, unless code-download is used.
- **Hypermedia as the engine of application state:** If it is likely that the client will want to access related resources, these should be identified in the representation returned (for example, by providing their URIs in sufficient context, such as hypertext links). Identifying these resources creates an environment in which the software system consuming the service has more than normal knowledge of the way the data is stored.

Diving into the details of REST

REST as a concept is as old as the web, but as a web service implementation, it's fairly new — a little newer than SOAP. The largest implementation of REST as a standard is the web itself. CGI is based on the REST interface. The call to a REST interface is clearly smaller than a SOAP call. Seriously. Look at these two examples:

REST:

```
POST /Start.asmx HTTP/1.1
Host: localhost
Content-Type: http; charset=utf-8
http://mydomain.com/start.aspx?id=3
```

SOAP:

```
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>
    <HelloWorld xmlns="http://tempuri.org/"/>
  </soap12:Body>
</soap12:Envelope>
```

The XML of SOAP kind of gets in the way. On the other hand, SOAP has a list of security features. There is a list of transaction features. There is a list of attachment features. SOAP has a lot of features. REST, not so much. REST has only four features under consideration in this chapter:

- GET – Request a resource
- PUT – Upload a resource
- POST – Submit data
- DELETE – Delete resource

Setting up a ASP.NET Core Web Application to Use REST

Previous versions of Visual Studio made you jump through all sorts of odd hoops in order to create a REST application. That's not the case any longer. You don't have to download any special templates, write code in odd ways, or hold your tongue in a certain way to get REST to work. The following sections describe how to use REST the easy way in Visual Studio 2017.

Creating the Web API

You have access to a number of Web API templates. The example in this case uses the ASP.NET Core Web Application (.NET Core) approach because it allows you to create a Web API that doesn't have a lot of extras added to it. The following steps will help you create the initial project:

1. **Choose File ➪ New ➪ Project.**

   You see the New Project dialog box.

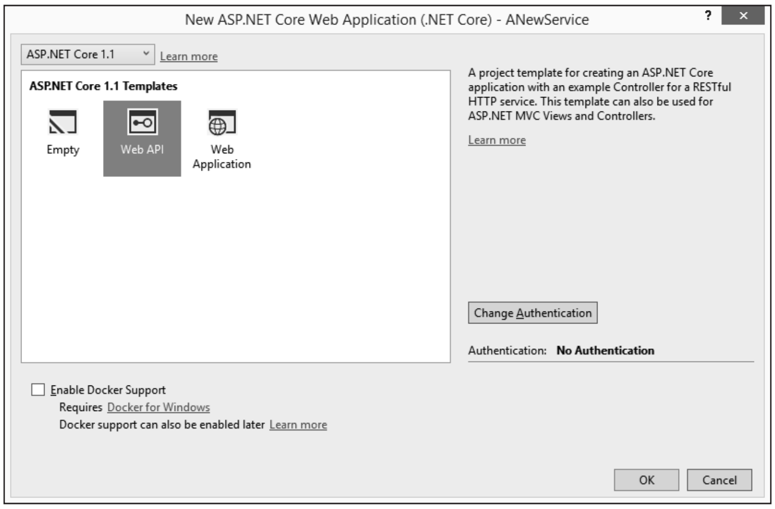
2. **Select the Visual C#\Web folder.**

   Visual Studio displays the list of projects.

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

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

   You see the New ASP.NET Core Web Application (.NET Core) dialog box, shown in Figure OC7-1. Note that unlike other project types, this one doesn't include a lot of extras, such as Model-View-Controller (MVC) support; its focus is on simplicity.



A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET MVC Views and Controllers.

Authentication: **No Authentication**

Requires Docker for Windows

Docker support can also be enabled later Learn more

FIGURE OC7-1: Choose an ASP.NET Core 1.1 Template from the list.

5. **Select Web API and click OK.**

   The wizard creates the project for you.

Adding some code

Normally you see examples online of RESTful applications doing something really complicated and you get lost in all the extraneous details. To make this REST example work, open the `ViewController.cs` file. Delete the existing code in the `ValuesController` class and add the following code to it:

```
[HttpGet("{values}")]
public string Get(string values)
{
    // Split the input values for parsing.
    string[] inputs = values.Split('&');

    // Verify that the input contains the right
    // number of values.
    if (inputs.Length != 2)
        return values;

    // Parse the input values into numbers.
    int value1 = 0;
    int value2 = 0;

    try
    {
        value1 = Int32.Parse(inputs[0]);
        value2 = Int32.Parse(inputs[1]);
    }
    catch (System.FormatException ex)
    {
        return values;
    }

    // Add the values together and return a result.
    string result = (value1 + value2).ToString();
    return result;
}
```

The task that this code performs is to accept a number of input values that are separated by an ampersand (&). It splits them into individual values. The example is looking for precisely two numbers. Consequently, `inputs` should contain just two entries. If not, the example outputs the original value. You'd probably output an exception of some sort in a production application, but this example keeps things simple.

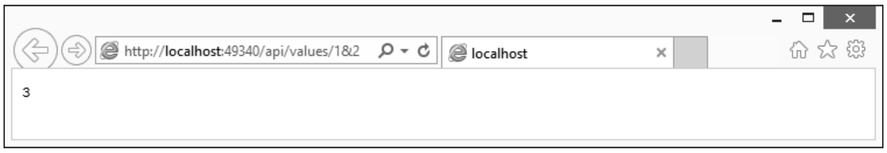
The next step is to determine whether the two input values are numbers. After all, someone could send letters or just about anything else. If the two inputs aren't numbers, then again, the code outputs the original string.

Finally, the example takes the two input numbers, adds them, and converts the result back to a string. The code then returns the string, which appears in the browser.

You need to make one additional change before you can try the example. Open the `launchSettings.json` file. Inside this file, you find a `launchUrl` setting. Change this setting so that it reads:

```
"launchUrl": "api/values/1&2"
```

At this point, you can start the example by pressing F5. Figure OC7-2 shows what you should see: a URL containing the host information, the call to `values`, and the input arguments of `1&2`. The output is 3, as expected.



```
3
```

FIGURE OC7-2: The output of 3 is what you expect.

Creating an Application to Consume the API

You've actually done a lot of the setup work for the test application in the past. As with Online Chapter 6, before you can perform testing, you need a new project. You can't add a Web Forms page directly to the service project as you did in the "Building a test page" section of Online Chapter 5. However, you can add it as a separate project, as you do when creating web pages in the "Creating a standard project" section of Book 4, Chapter 2, in the book. Make sure that you create an Empty project type and name it WebFormTest. After you have an empty web project to use, you can add a page to it and configure it by using precisely the same directions you use in the "Building a test page" section of Online Chapter 5 of this minibook.

Now that you have a user interface to use, you need to create the `Invoke_Click()` method by double-clicking the `Invoke` button on the user interface. To create the required code, you need to add these two `using` statements:

```
using System.Net;
using System.IO;
```

After adding the `using` statements, you can add the code to the `Invoke_Click()` method, like this:

```
protected void Invoke_Click(object sender, EventArgs e)
{
    // Create the request URL.
    string url = "http://localhost:49340/api/values/" +
        Value1.Text + "&" + Value2.Text;

    // Define the request object.
    HttpWebRequest req = (HttpWebRequest)WebRequest.Create(url);
    // Make the request.
    string output = "0";
    using (HttpWebResponse resp = (HttpWebResponse)req.GetResponse())
    using (Stream stream = resp.GetResponseStream())
    using (StreamReader read = new StreamReader(stream))
    {
        output = read.ReadToEnd();
    }

    // Display the result on screen.
    Result.Text = output;
}
```

This example is different from the other two because you're relying on REST's using the `GET` method. The example begins by creating the request URL — the same one you see in Figure OC7-2. It then uses the various System.Net classes to create a request, make the request, and obtain a response. The response is in the form of a `Stream`, so you need to process it as you would any other `Stream`. Because of the way in which the Web API is created, the only information you get back is the actual response: a `string` showing the output value.

To start this example, you must ensure that ANewService is the startup project and start the web service by choosing Debug ➪ Start Without Debugging. If you start the web service in the debugger, you won't be able to start your test page. Next, right-click WebFormTest in Solution Explorer and choose Set As Startup Project. Press F5 to run the example. Figure OC7-3 shows typical output.

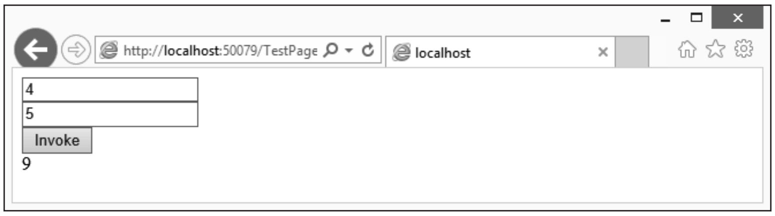


FIGURE OC7-3: The result of an extremely simple REST request.