Visual Studio Magazine

ASP.NET Web API: End-to-End

Integrating the ASP.NET Web API into your applications isn't only the best way to add AJAX to your Web-based applications, it's also the easiest. And that's just the start of the story.

BY PETER VOGEL

1105 MEDIA

TABLE OF CONTENTS

Visual Studio Magazine

ASP.NET Web API: End-to-End

Integrating the ASP.NET Web API into your applications is not only the best way to add AJAX to your Web-based applications, it's also the easiest. And that's just the start of the story. BY PETER VOGEL

In a BYOD world, using the Web API lets developers build multiple front-ends with a variety of workflows supported by a single set of Web API services.

The ASP.NET Web API ("Web API" from here on) is the latest technology built on the ASP.NET foundation (the other two being ASP.NET Web Forms and ASP.NET MVC). And because the Web API shares that foundation with both of Microsoft's platforms for creating Web applications, the Web API provides a flexible, testable way to create applications that execute in the browser while reaching back to integrate with the server.

The Web API enables ASP.NET developers (both MVC and Web Forms) to create an AJAX-enabled application—which means creating applications with more responsive UIs and improved scalability. In a Bring Your Own Device (BYOD) world, using the Web API lets developers build multiple front-ends (for small-, medium- and large-screen devices) with a variety of workflows (from the focused mobile user to the intense desktop user) supported by a single set of Web API services. And, because the Web API is fully integrated with the ASP.NET framework, if you've secured your Web site then you've secured the AJAX code in the Web pages accessing a site's services. But Web API services aren't limited to being used just by JavaScript running in a browser: Your services, with the appropriate security included, can be called from any .NET platform.

Building the Right Services

The first step in building a Web API application is to decide what services you need. While it's tempting to build services that are tied to individual tables or objects (for example, creating a service that

updates the Customer table or provides access to the Customer object) you should build your Web API services by thinking in terms of business resources.

For example, a sales order that lists all the information about a customer's purchases is a business resource. However, the database design and the object design to support that business resource would include multiple entities: Customer, CustomerAddress, SalesOrderHeader, SalesOrderDetail, Product and, potentially, others. Creating services based around those entities would be fatal to your applications. Your users need the business resource and, in assembling that resource, any application would have to make repeated trips to the server to fetch each of those entities. Performance, especially for mobile users sending signals out over the air, would suffer as the number of trips to the server increase.

While there are costs associated with sending large packets of data, those costs are trivial compared to the transmission time involved in getting a request to the server and getting a response back.

While there are costs associated with sending large packets of data, those costs are trivial compared to the transmission time involved in getting a request to the server and getting a response back. To put it another way: It will always take twice as long to make two trips to your service; you probably can't measure the time to send twice as much data. In designing your services, first consider the business resources you must support.

The second issue you want to consider in designing your services is what a user will do with the business resource your service provides. Users will, obviously, want to create a new sales order, add purchases to a sales order and cancel a sales order. Users are also going to want to go beyond those simple updates to, for instance, perform "what-if" calculations that can answer questions such as, "Can I afford to buy something else?" and, "Will this change qualify for free shipping?" Your service must not only support managing your business resources but also your users' activities.

What you shouldn't do is design your services to support a particular workflow—you shouldn't assume specific activities take place either before or after the service is called. This helps ensure that your services will be reusable. At the very least, in a BYOD world, workflows often vary from one kind of device to another: The workflow you implement for a smartphone probably concentrates on essential actions and doesn't offer as many options as the workflow you create for desktop users.

As an example, consider a service that allows an application to add another product to a sales order. Don't force the client to send a

complete copy of the sales order with any new order lines added—that assumes the application has previously retrieved and stored a copy of the sales order at the client (not likely in a mobile device). For an action's input, you should think in terms of the minimum data required: a product name and the quantity that's to be added to the user's currently open sales order, for example. For the return value, on the other hand, you should consider returning all the data the user might need. For the sales order update service, that means you should consider returning a complete copy of the new, updated sales order so the user can see the results of his changes.

Starting Your Project

A service needs a host: Some application that will listen for requests and, when a request comes in, wake up your service to process the request. The obvious host to use is IIS, in which case you begin creating your Web API service by creating an ASP.NET or ASP.NET MVC project. However, if you want to run you service on a computer without IIS you can create your own host (not hard to do). In that scenario, you can begin creating your Web API service by creating a Windows Service, Windows Presentation Foundation, Console or Windows Forms project.

Terminology

The term "Web API service" is often used to refer to two things:

- A Web site that provides support for business activities
- A single Web API class that supports some of the activities on the Web site

So someone might refer to "our customer service" meaning a Web site that supports a wide variety of customer activities; that same person might also refer to "our customer update service," meaning the single Web API class on the site that handles adding, updating and deleting customer data in the database (other Web API classes in the same site might support customer credit checking, the customer loyalty program and so on). In this paper the term "a service" will refer to a Web API class. That means that a Web site will usually contain multiple Web API services.

The activities a service supports might be called "operations" by an SOA architect but, in the ASP.NET world, they're called "actions." For every action there will be at least one method but, as you'll see, in the Web API you can associate multiple methods with a single action. In this paper I'll use the terms "action" and "method." A SOA architect would also say that actions are "invoked"—I'll use the more programmer-friendly "called."

Before you can use Web API in any project, though, you need to ensure the Microsoft ASP.NET Web API package is part of your project. With Visual Studio 2012, the Web API package is included automatically with ASP.NET MVC projects; with other project types or in earlier versions of Visual Studio, you'll need to use NuGet to add the Web API package (you can add it from the Tools | Extension Manager menu). If you're not sure if you have the ASP.NET Web API included in your project, the easiest way to check is by right-clicking on your project in Solution Explorer and selecting the NuGet menu choice. Then, in the Manage NuGet Package dialog, select Installed packages and do a search for "webapi." If the Web API has been added, it will appear in the resulting list.

It's a good idea to create a folder in your project and right-click on it to add your service to that folder.

Once the Web API package is installed, to add a Web API service to your project, you can just right-click on your project and select Add New Item. It's a good idea, however, to create a folder in your project and right-click on it to add your service to that folder. As the base for your service, you can use the Web API Controller Class template you'll find in the Web node of the Add New Item dialog. However, using the template creates a class with a lot of default code that you'll just have to delete later. Adding an ordinary class file and having it inherit from the System.Web.Http.ApiController class gives you a Web API service without that extra code. Whichever way you create your service, when you name your Web API class make sure the name ends with the word "controller" (for example, the case study for this paper uses a controller called SalesOrderController).

This example creates a Web API controller called Customer:

```
using System.Web.Http;
namespace WebApplication1
{
  public class SalesOrderController : ApiController
  {
```

The simplest Web API method (the equivalent of the "Hello, World" program) just returns an HTTP status code and looks like this:

```
public HttpResponseMessage Get()
{
  HttpResponseMessage hrm = new HttpResponseMessage(HttpStatusCode.OK);
  return hrm;
}
```

Tying Services to URLs

Your next step is to tie your service to a URL using a routing rule. A routing rule consists of a template that specifies the format of the URLs to which the rule applies and which controller is to handle the request. The template also specifies where in the URL find values that are to be passed to actions in your service can be found.

In an ASP.NET MVC 4 application, you add a routing rule to the WebAPIConfig class in the App_Start folder; in ASP.NET MVC 3 or ASP.NET, you put routing rules in the Application_Start method of the Global.asax file). If you're creating your own host, you add routing rules to the code that defines your host. The code is almost identical in all of these environments.

In ASP.NET MVC 4, the code for the default rule supplied by the Web API temple in the WebAPIConfig class looks like this:

```
public static void Register(HttpConfiguration config)
{
  config.Routes.MapHttpRoute(
```

Creating Services Outside of ASP.NET

If you aren't using an ASP.NET Web Forms or ASP.NET MVC project, you must also create your own host in your project. You could add this code to an event in a form, for instance, to have it host your SalesOrder controller:

```
private HttpSelfHostServer shs;
HttpSelfHostConfiguration hcfg =
  new HttpSelfHostConfiguration("http://localhost:1867/MyServices");
shs = new HttpSelfHostServer(hcfg);
shs.OpenAsync();
```

The URL passed to the HttpSelfHostConfiguration specifies the URL you should use to access your service. When your host is shut down, you should close your service connection and dispose of it:

```
shs.CloseAsync().Wait();
shs.Dispose();
```

If you're creating your own host, make sure your host is your project's startup item.

```
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
   );
}
```

In an ASP.NET application or an ASP.NET MVC 3 application, the equivalent code in the Application_Start method of the Global.asax file looks like this:

```
GlobalConfiguration.Configuration.Routes.MapHttpRoute(
  name: "DefaultApi",
  routeTemplate: "api/{controller}/{id}",
  defaults: new { id = RouteParameter.Optional }
);
```

If you're creating your own host, you'd add your routing code following the line where you instantiated the HttpSelfHost-Configuration object. That code would look like this:

```
hcfg.Routes.MapHttpRoute(
  name: "DefaultApi",
  routeTemplate: "api/{controller}/{id}",
  defaults: new { id = RouteParameter.Optional }
);
```

The literal text "api" in the routeTemplate parameter forces all URLs that request your service to have the text "api" after the server and site name that your host establishes.

The literal text "api" in the routeTemplate parameter forces all URLs that request your service to have the text "api" after the server and site name that your host establishes. So, for example, a Web site with the URL phvis.com would use a URL that begins phvis.com/api to request your services.

In the routeTemplate, the word "controller" inside the braces ({controller}) means that the text following "api" must be the name as one of your controllers (though without the "controller" suffix). The "id" inside the braces means that any part of the URL following the controller name will be assigned to the name "id." In the routing rule, the default parameter specifies that the id parameter can be omitted from the URL requesting your service.

The simplest possible service has a single action, named Get. When you issue a URL from the address bar of your browser, you're using the HTTP Get verb and, by default, the Web API looks for methods with the same name as the HTTP verb used with the request. With your routing rule in place, you can test a SalesOrder-Controller service by typing this into the address bar of any Web browser:

```
http://localhost:1867/api/salesorder
```

If you've done everything right, your browser should display a blank page—not very exciting but you know that your service is accessible.

If having a method name that duplicates the name of an HTTP verb violates your coding convention, you can use the HttpGet attribute to flag the method you want to use to respond to HTTP Get requests. The following code creates a method called RetrieveSalesOrder in the Customer controller that will handle Get requests:

```
public class SalesOrderController : ApiController
{
  [HttpGet]
  public void RetrieveSalesOrder()
  {
```

The default routing rule begins with the string "api" so that only URLs containing that string will be used with your Web API services.

Unfortunately, the default routing rule can also be used with URLs that can have invalid controller names embedded in them (or, in an ASP.NET MVC application, the names of controllers that aren't Web API controllers). To reduce those problems, the default routing rule begins with the string "api" so that only URLs containing that string will be used with your Web API services. Even with that convention, because the default rule is so flexible, as the number of rules in your application increases it can be difficult to predict which rule will be used to process a request's URL. In addition, the default routing rule also tightly couples URLs to your Web API Services. If you rename your Web API controller then any clients using URLs with the old controller name will stop working.

To avoid those problems, it's a better idea to use more explicit routing rules that omit controller name from the routing rule's template (this also eliminates the need for the "api" prefix). Routing rules can, instead, set your controller's name in the default values parameter. As an example, the following routing rule applies only to URLs beginning with the word "SalesOrder" (still following the server and site name, of course). This rule specifies the controller value in the default parameter. This example is for an ASP.NET Web Forms app:

```
GlobalConfiguration.Configuration.Routes.MapHttpRoute(
  name: "SalesOrderGet",
  routeTemplate: "SalesOrder",
  defaults: new { Controller = "SalesOrder" }
);
```

This new rule would apply to a URL like http://localhost:49695/salesorder.

If you were to change the name of your Web API controller to, for example, SalesOrderManagementController, you would only need to change the value in the default parameter—existing URLs would continue to work. This example is for an ASP.NET MVC 4 application:

```
config.Routes.MapHttpRoute(
  name: "SalesOrderGet",
  routeTemplate: "SalesOrder",
  defaults: new { Controller = "SalesOrderManagement" }
);
```

Routing rule templates also allow you to assign names to values passed in a URL so that those values can be passed—as parameters—to your actions. This template, for instance, specifies that the part of the URL following the text "SalesOrder" is to be called "SalesOrderId":

Routing rule templates allow you to assign names to values passed in a URL so that those values can be passed—as parameters—to your actions.

```
GlobalConfiguration.Configuration.Routes.MapHttpRoute(
  name: "SalesOrderGet",
  routeTemplate: "SalesOrder/{SalesOrderId}",
  defaults: new { Controller = "SalesOrder" });
```

To access these parameters you just need to use the name for a parameter to your method, like this:

```
public class CustomerController : ApiController
{
  [HttpGet]
  public void RetrieveSalesOrder(string SalesOrderId)
{
```

This process of binding URL values to parameters is extremely flexible. If, for example, your method has a parameter that doesn't have a value specified, your parameter will be set to null. Therefore, it's a best practice to have your method parameters use data types that can be set to null (string, for example). If your method doesn't use a value from the URL that's specified in the routing rule template, the value from the URL is simply discarded. The process will also attempt to perform any data conversions required by your method parameters. If, for example, you specify that your method parameter is an integer, the model binder will attempt to convert the value from the URL into an integer (and throw an exception if it can't).

In your URL you can combine values and text in any combination that makes sense for your service. However, as the number of your services increases, it can be difficult to be sure which rule applies to which URL (there's more about routing rules on p. 17).

Getting Data Using JavaScript

Once you've created a basic Web service, you can call it from JavaScript in your Web page. The simplest way to do that is to tie a JavaScript function to the onclick event of a button. The following example shows the HTML you'd put in an ASP.NET MVC view to define a button that calls a JavaScript function named GetSalesOrder. The sample HTML also includes textboxes for displaying a sales order's Id and Priority. Below the button, a span element displays messages from your JavaScript function:

```
@Html.EditorFor(m => m.SalesOrderId)
@Html.EditorFor(m => m.Priority)
<input type="button" value="Retreive"
  onclick="GetSalesOrder();"  />
<span id="StatusMessage"></span>
```

This process of binding URL values to parameters is flexible.

In an ASP.NET application, a similar form would look like this:

```
<asp:TextBox ID="SalesOrderID" runat="server"></asp:TextBox>
<asp:TextBox ID="Priority" runat="server" text="1"></asp:TextBox>
<input id="Button1" type="button" value="Retrieve" onclick="GetSalesOrder()"/>
<asp:Label ID="StatusMessage" runat="server"></asp:Label>
```

The next step is to create the JavaScript function that will issue a request to the URL of your Web API service. This example uses the routing rule (described earlier) that includes a value for the sales order Id in the URL (this code also assumes that you've added jQuery to your project and to this page so that you can use the jQuery getJSON function to call the service):

```
<script>
function GetSalesOrder()
{
  $.getJSON('SalesOrder/' + $(\"#SalesOrderID").val(),
    function (soDTO)
    {
      alert(soDTO.SalesOrderId);
    }
   );
  }
</script>
```

In an ASP.NET Web Forms application you might want to consider enhancing that function to call any client-side code on the page generated by your Validators before you send data to the server:

```
function GetSalesOrder()
{
  if (Page_ClientValidate())
  {
    $.getJSON('SalesOrder/' + $("#SalesOrderID").val(),
      function (soDTO)
      {
        alert(soDTO.SalesOrderId);
      }
    );
  }
}
```

This function builds the URL that's passed as the first parameter to the getJSON function using the name of the controller and whatever value is currently in the SalesOrderID text box. When the service returns an object to the getJSON method, that object will be passed to the function in the second parameter to the getJSON method. This example just displays a single property from the object returned from the service—in real life you'd use some more jQuery to insert the data retrieved into your page.

When the service returns an object to the getJSON method, that object will be passed to the function in the second parameter to the getJSON method.

The next step is to create the service method that will return a sales order object. With the Web API, that code can be very simple to write—whatever code you would use in any other application to retrieve, create and return the appropriate object. Because your object will combine the results of several different server-side objects and tables, you should return a Data Transfer Object (DTO) that holds all of the data the client needs.

This example has the method's return type set to SalesOrderDTO and returns the object created in the method:

```
[HttpGet]
public SalesOrderDTO RetrieveSalesOrder(string SalesOrderId)
{
  SalesOrderDTO soDTO = new SalesOrderDTO();
  // ... Set properties on soDTO
  return soDTO;
}
```

The getJSON method is just a quick way to call a more flexible jQuery function: the AJAX function. The equivalent AJAX version of the getJSON method just shown only requires one new parameter that holds the function to execute when an error occurs. Here's that

AJAX method, using named parameters to make the code easier to read:

```
function GetSalesOrder()
{
  $.ajax({
    url: 'SalesOrder/' + $("#SalesOrderID").val(),
    success: function (soDTO, status)
      {
        if (status == "success")
        {
          alert(soDTO.SalesOrderId);
        }
      },
    error: function (err)
      {
        $("#StatusMessage").text(err.statusText);
      }
    });
  }
```

Because your object will combine the results of several different server-side objects and tables, you should return a DTO that holds all of the data the client needs.

Using the AJAX function positions you to perform updates with your service.

On the server, to return your object, you can use the HttpResponseMessage to give you more control over the status code returned by your method. Returning the HttpResponseMessage object from your method means that the business object your code creates will need to be put in HttpResponseMessage object's Content property as a JSON object. This example returns the SalesOrderDTO inside an HttpResponseMessage:

```
[HttpGet]
public HttpResponseMessage RetrieveSalesOrder(string SalesOrderId)
{
  SalesOrderDTO soDTO = new SalesOrderDTO();
  // ... Set properties on soDTO
  HttpResponseMessage hrm = new HttpResponseMessage(HttpStatusCode.OK);
  hrm.Content = new ObjectContent<SalesOrderDTO>(soDTO,
    new JsonMediaTypeFormatter(),
    new MediaTypeWithQualityHeaderValue("application/jso"));
  return hrm;
}
```

Now, if your code discovers, for example, a security issue when preparing the DTO to be sent to the user, you can return an HttpStatusCode.Forbidden in your HttpResponseMessage.

Updating Data

While the ajax method defaults to the HTTP Get verb, you can specify other verbs when making a request and, as a result, call

The getJSON method is just a quick way to call a more flexible jQuery function: the AJAX function.

Using Your Service from .NET Clients

You're not restricted to calling your services from JavaScript. If you use NuGet to add the Web API Client to a .NET project, you can call your services from that project's code through the HttpClient object.

The following code creates the HttpClient object and then sets its BaseAddress property to the service's Web site. The code then specifies that it wants to receive JSON objects by adding an entry to the default headers sent with the request. It then passes the rest of the URL for the request through the client's GetAsync method to call the service. The methods on the HttpClient are all asynchronous so this code uses the await keyword to pause the client until the service responds:

```
HttpClient hc = new HttpClient();
SalesOrderDTO soDTO;
hc.BaseAddress = new Uri("http://www.phvis.com");
hc.DefaultRequestHeaders.Accept.Add(
  new MediaTypeWithQualityHeaderValue("application/json"));
var resp = await client.GetAsync("SalesOrder/A123");
```

Once you do get a response you can check that everything went well by calling the response object's EnsureSuccessStatusCode as the following code does (EnsureSuccessStatusCode throws an exception if it finds a bad status code in the response). Finally, the code extracts a SalesOrderDTO object from the response's Content property:

```
resp.EnsureSuccessStatusCode();
HttpResponseMessage soDTOHold =
  await response.Content.ReadAsAsync<SalesOrderDTO>();
soDTO.CopyFrom(soDTOHold);
```

other methods in your service. For example, to support deleting data on the server, you can use the HTTP Delete verb in your jQuery AJAX function. In your service you just need to define a method called Delete and the Web API will use that method to satisfy the AJAX request. Rather than return an object, this method returns a string formatted as XML, which reports that the order was deleted:

```
public HttpResponseMessage Delete(string SalesOrderId)
{
  // ... Find and delete a sales order using the SalesOrderId
  HttpResponseMessage hrm = new HttpResponseMessage(HttpStatusCode.OK);
  hrm.Content = new ObjectContent<string>(SalesOrderId + " deleted",
    new XmlMediaTypeFormatter(),
    new MediaTypeWithQualityHeaderValue("text/plain"));
  return hrm;
}
```

In the jQuery AJAX function you specify the verb to use with the function's type parameter. The following code uses the Delete verb

Testing Your Web API Service, *continued on page 14*

If you want to ensure your service is working reliably you can, of course, create a test client and use its UI to test your Web service. It's considerably easier, however, to create a Visual Test project to test your service's behavior.

Creating a Test project to support building a Web API service is just like creating any other Test project: right-click on your solution, select Add New Project and, from the Test node in the Add New Project Dialog, select Unit Test Project. Once the Test project is added, add to it a reference to your Web API project. Rename the default UnitTest1 file to something more meaningful and add a using statement for your Web API project. As with your other Web API projects, add the Web API NuGet package but you'll also need to add a separate reference to System.Web.Http.

This code, at the top of a TestClass, instantiates a controller:

```
[TestClass]
public class SalesOrderTest
{
  [TestMethod]
  public void TestRetrieve()
  {
    SalesOrderController soc = new SalesOrderController();
```

Testing Your Web API Service, *continued*

In your test methods, you can instantiate your controller (using its full name with the "controller" suffix) and call its methods like any other class. The only interesting code in your Test methods is the line to extract the Content property of your HttpResponseMessage and convert it into something testable.

This code, as an example, calls the RetrieveSalesOrder method on the SalesOrderController and accepts a result in an HttpResponseMessage. Since the RetrieveSalesOrder method puts a JSON version of the SalesOrderDTO class in the HttpResponseMessage, this code using the JsonConvert object to recreate the SalesOrderDTO object. With the SalesOrderDTO object retrieved, the code tests to see if the method returned the correct result:

```
HttpResponseMessage hrm;
hrm = soc.RetrieveSalesOrder("A123");
SalesOrderDTO so  = JsonConvert.DeserializeObject<SalesOrderDTO>
  (hrm.Content.ReadAsStringAsync().Result);
Assert.AreEqual("A123", so.SalesOrderId);
```

You can now run this method from Visual Studio to see if the method finds and returns the right object. More important, as your Web API service continues to evolve, you can run this test repeatedly to make sure that your service method continues to work as you intended.

and, in the success method, extracts the returned message from the XML document returned by the services. Because I specified string as my ObjectContent type, the code looks for an element called string in the XML document:

```
$.ajax(
  {
    url: 'SalesOrder/' + $("#SalesOrderID").val(),
    type: 'Delete',
    success: function (respMsg, status)
      {
        var resp = $.parseXML(respMsg);
        var xml = $(resp);
        var elm = xml.find("string");
        $("#StatusMessage").text(elm.text());
      },
      error: function (err)
      {
        $("#StatusMessage").text(err.statusText);
      }
```

```
    }
);
```

To handle creating a new object, you'll need to send more than a few pieces of data embedded in a URL—typically you'll need to send a whole object (and extend your routing rules, as described on p. 17). When you're sending a large amount of data the best approach is to use the HTTP Post verb, which puts your data in the body of the message sent to the server. While jQuery's AJAX function supports the Post verb, jQuery also has a dedicated function called post for moving data to the server.

To use the post function, you must first assemble the data into an object you can send to the service. In most cases, your application will have retrieved the object that you're now creating (if only to make sure that the entity doesn't already exist at the server). You can just reuse that object in your client. This example, for instance, moves the SalesOrderDTO object to a client-side variable as part of the success function:

To use the post function, you must first assemble the data into an object you can send to the service.

```
var soDTOHold;
function GetSalesOrder()
{
  $.ajax(
    {
      url: 'SalesOrder/' + $("#SalesOrderID").val(),
      type: 'GET',
      success: function (soDTO, status)
        {
          soDTOHold = soDTO;
        },
      error: function (err)
        {
          $("#StatusMessageh").text(err.statusText);
        }
      });
```

The following code updates the SalesOrderDTO Priority property with data from the textbox in the form. It then uses the jQuery post function to send the updated object to the service. The SalesOrderDTO is passed in the post function's second parameter. The third parameter passed to the post method is the function to run when the result is returned from the server, which will be another XML-formatted message:

```
soDTOHold.Priority = $("#Priority").val();
// ... Update other properties on the DTO from the form
$.post('SalesOrder',
  soDTOHold,
  function (respMsg, status) {
    var resp = $.parseXML(respMsg);
    var xml = $(resp);
    var elm = xml.find("string");
    alert(elm.text());
  });
```

On the server, you need a method whose name either begins with the word "Post" or is decorated with the HttpPost attribute. This example uses the attribute:

On the server, you need a method whose name either begins with the word "Post" or is decorated with the HttpPost attribute.

```
[HttpPost]
public HttpResponseMessage UpdateSalesOrder(SalesOrderDTO soDTO)
{
  // ... Code to update the database with information from soDTO
  HttpResponseMessage hrm = new HttpResponseMessage(HttpStatusCode.OK);
  return hrm;
}
```

Once you start posting data, you ca (if you wish) send all the data in the form displayed in your browser. This example assumes the form that's displaying the data is called "form1" and sends it, as an object, to the service. This example also includes an error function:

```
function UpdateSalesOrder() {
  $.post('SalesOrder', $('#form1').serialize())
  .success(function () {
    var resp = $.parseXML(respMsg);
    var xml = $(resp);
    var elm = xml.find("string");
    $("#StatusMessag").text(elm.text());
  })
  .error(function (err) {
    $("#StatusMessage").text(err.statusText);
  });
}
```

The service method that processes this request will need a parameter with properties whose names correspond to the names assigned to the textboxes and other controls on the form.

Return to Routing

However, to make this post code work, you'll need to revisit your routing rules. The original routing rule, for example, applies to URLs that begin with the word "SalesOrder" and also contain another component called SalesOrderId:

```
config.Routes.MapHttpRoute(
  name: "SalesOrderGet:",
  routeTemplate: "SalesOrder/{SalesOrderId}",
  defaults: new { Controller = "SalesOrder" });
```

This rule won't work with the route that the post function uses because the post's URL doesn't include a SalesOrderId component. One solution is to make this route more flexible by specifying in the defaults parameter, that the SalesOrderId is optional:

```
config.Routes.MapHttpRoute(
  name: "SalesOrderGet",
  routeTemplate: "SalesOrder/{SalesOrderId}",
  defaults: new { Controller = "SalesOrder",
    SalesOrderId = RouteParameter.Optional });
```

Once you start posting data, you can (if you wish) send all the data in the form displayed in your browser.

However, the more flexible your routing rules are, the more difficult it is to determine which rule is going to be used with any request URL. A better solution to handling your Post request is to add a second rule (with a new name) and a template that matches the URL actually used by the post method:

```
config.Routes.MapHttpRoute(
      name: "SalesOrderPost",
      routeTemplate: "SalesOrder",
      defaults: new { Controller = "SalesOrder"});
```

As the number of actions in your services increase, you may start running out of HTTP verbs. There are four HTTP verbs that are commonly used in most services to perform standard business activities:

- Get: To retrieve data
- Post: To create a new entity
- Delete: To remove an entity
- Post: To update an entity (technically, to completely replace the entity at the server with the data sent from the client)

While it isn't common, some services use the Patch verb to apply updates that change only some of the data in an object or table. You can probably already see that you'll need more than four actions in your service. However, there are two ways that you can add additional actions.

The first method is through method and route overloading. In the same way that you can have multiple methods with the same name, provided each method has a different parameter list, you can have multiple routing rules for the same HTTP verb, each with a different number of components (or arrangement of components) in the URL. For instance, to retrieve all the sales orders for a particular month, you can specify a URL like this:

```
http://localhost:1867/SalesOrder/July/2013
```

This URL would be handled by a routing rule that also has three components in its template:

```
config.Routes.MapHttpRoute(
  name: "SalesOrderGet",
  routeTemplate: "SalesOrder/{Month}/{Year}",
  defaults: new { Controller = "SalesOrder" });
```

The more flexible your routing rules are, the more difficult it is to determine which rule is going to be used with any request URL.

The method that would be called could still be called Get or RetrieveSalesOrder but would have two parameters corresponding to the named values in the routeTemplate:

```
public HttpResponseMessage Get(string month, string year)
{
```

As you add rules, however, it's possible to add a new rule that prevents an existing rule from being used. For instance, you could specify that the year is optional when retrieving all the sales orders for a month—your service would just default to the current year if a year isn't provided. The routing rule is easy to write:

```
config.Routes.MapHttpRoute(
  name: "SalesOrderGet",
  routeTemplate: "SalesOrder/{Month}/{Year}",
  defaults: new { Controller = "SalesOrder",
    Year = RouteParameter.Optional });
```

The problem is that your service can now accept two URLs that look alike, at least as far as the number of components in each URL. The URL for requesting a sales order and the URL for requesting the sales for a month in the current year both begin with the string "SalesOrder" and have only one other component:

```
http://localhost:1867/SalesOrder/A123
http://localhost:1867/SalesOrder/July
```

The Web API has rules for deciding which routing rule to apply in these situations but it's probably not a good idea to count on them. You'll find it much easier to determine which URLs will call which actions in your service if you give the different URLs unique text:

```
http://localhost:1867/SalesOrder/A123
http://localhost:1867/SalesOrdersForMonth/July
```

The second option for extending your service beyond the HTTP verbs is to specify the name of your action in your routing rule. The following rule specifies the name of the action to use in its default. It also uses a new parameter (constraints) to say that this rule only applies to Get requests. This example would be used in a self-hosting project:

```
hcfg.Routes.MapHttpRoute(
  name: "SalesOrderPost",
  routeTemplate: "SalesOrder",
  defaults: new { Controller = "SalesOrder\{SalesOrderId}",
    Action="EstimateSalesOrderPrice"},
  constraints: new { httpMethod =
    new HttpMethodConstraint(HttpMethod.Get) });
```

As you add rules, it's possible to add a new rule that prevents an existing rule from being used.

This rule would route requests to this method:

```
public HttpResponseMessage EstimateSalesOrderPrice(string SalesOrderId)
{
```

Sending the Result the Client Wants

So far, this sample service has returned JSON objects and XML documents. In both cases, the service decided what kind of data to return to the client. However, the Web API allows the client to indicate the kind of data that it wants and for the Web API to decide what to return based on the client's request (a process called "content negotiation" or "conneg" for short). If you're willing to give up some control over the results your service sends, you can support content negotiation by writing less code.

The first step is for the client to specify—as part of requesting your service—what kind of response it wants. When using the post function, for instance, a client can specify a type parameter that indicates the kind of data to be returned. This example specifies that XML should be returned from the service and parses the result from the service as XML:

```
$.post('SalesOrder',
  soDTOHold,
  function (respMsg, status) {
    var resp = $.parseXML(respMsg);
    var xml = $(resp);
    var elm = xml.find("SalesOrderDTO");
    alert(elm.text());
```

```
    },
    "xml");
```

If your service always returns a JSON object then your service isn't going to work well with this client.

However, rather than specifying the return type for the data, you can let the Web API decide on the format of the data returned. Your action will still return an HttpResponseMessage to the client but the Web API will decide on the format of the data put into the HttpResponseMessages object's Content property. You just need to call the CreateResponse method on the original request sent to your service; you can access the request through the Request property built into your service controller. The CreateResponse method is a generic method so you need to specify the type of the object you're returning and, in addition, you must pass the method status code for the request and the data to format.

Errors in Your Routing Code

Routing rules can look mysterious because they make extensive use of anonymous objects. In the following example, for instance, the defaults parameter is set by an anonymous object that's assigned two properties (Controller and Action) that are immediately set to string values ("SalesOrder\{SalesOrderId}" and "EstimateSalesOrderPrice"):

```
hcfg.Routes.MapHttpRoute(
  name: "SalesOrderPost",
  routeTemplate: "SalesOrder",
  defaults: new { Controller = "SalesOrder\{SalesOrderId}",
    Action="EstimateSalesOrderPrice"});
```

You can add any properties you want to an anonymous object (so called because there's no class name between the keyword new and the brace where the properties are specified)—all you have to do is make up a name for your property and set it to a value. It's a flexible system that lets the Web API support almost any routing rule you might need.

However, because you're making up the property names as you type them in, you don't get any IntelliSense support. This opens you up to "spelling counts" errors where you meant to create a property called "Action" but mistyped and created a property called "Acion." In addition, as the example shows, the properties in these objects are often set to strings that aren't checked by the compiler. Here, again, you can make typing errors that will cause your application to fail.

The best advice here is: If your service methods aren't being called when you expect them to be, the first bug you should check for is spelling errors in your routing rules.

This example returns a status code of OK and a SalesOrderDTO object in whatever format the client requested:

```
public HttpResponseMessage Get(string SalesOrderId)
{
  // ... Code to retrieve sales order data
  HttpResponseMessage hrm =
    this.Request.CreateResponse<SalesOrderDTO>(HttpStatusCode.OK, soDTO);
  return hrm;
}
```

If you're willing to give up some control over the results your service sends, you can support content negotiation by writing less code.

Now, the various clients for your service can specify a variety of data formats and you can deal with all of them.

While this article has covered all of the basics for creating an ASP.NET Web API service, there's more that you can do—it's a rich environment. For example, if you have processing that crosses many services, you can centralize that processing in message handlers that manipulate the messages that pass between the client and the service. Content negotiation is supported through media formatters and, if you want to support a new response format, you can create your own custom media formatters. But everything you require to create the services you need to support typical business applications is right here. VSM

Peter Vogel has been developing ASP.NET application since .NET Framework 1.0 was released. Peter also facilitates the design of Service Oriented Architectures with his clients, in addition to helping his clients implement those architectures and build applications that take advantage of them. In his parallel career as a technical writer, Peter writes the Practical .NET column for Visual Studio Magazine Online and the Patterns in Practice column for MSDN Magazine Online.