Creating REST-ful Services with WCF

Estimated time for completion: **45 minutes**

Overview:

In Part1 of this exercise you will learn how to create a REST-ful WCF service that exposes POX (Plain Old XML) over SOAP-less endpoints with rich Uri mapping. In Part 2 you'll create a service that exposes a syndication feed in both RSS 2.0 and ATOM 1.0 formats. And in Part 3 (which is optional) you'll create a service that accepts and returns JSON encoded objects for an ASP.NET AJAX client.

Goals:

- Build and parse complex uri's using the UriTemplate class.
- Employ WCF's web programming model to provide XML that can be consumed by browsers and other non-SOAP-aware clients.
- Expose RSS and ATOM syndication feeds.
- Create a WCF service that can be invoked by an ASP.NET AJAX web app using JSON.

Lab Notes:

This lab requires the use of Internet Information Server (IIS) to host an ASP.NET AJAX application and WCF service. Users will need admin rights in order to create and configure web apps in IIS and grant file access permission to the NetworkService account.

WCF services will need permission to register a prefix with http.sys. If you do not possess admin rights, or you are running Windows Vista, an administrator will need to execute the following at a Visual Studio 2008 command prompt (replace "student" with the current user account name):

```
netsh  http add urlacl url=http://+:1234/BookService user=student
```

Part 1 – Create a WCF Service that adheres to the principles of the Web

In this part of the exercise you will create a WCF service that adheres to the principles of the web, which delivers POX (Plain Old XML) without an enclosing SOAP envelope. Doing so will allow accommodate clients that do not use SOAP and which do not require the higher order WS- protocols for interoperability pertaining to things like security, reliable messaging or transactions.*

Steps:

1. Open the WcfRest.sln solution located in the Before\Part1 directory.
2. Your first step will be to modify the contract for each operation your service exposes and to enable mapping of uri's to operations and method parameters.
    a. Open BookService.cs in the Service project.
    b. Add a [WebGet] attribute to the first operation in IBookService: GetBookTitles. Place it beneath the [OperationContract] attribute.
    c. Specify the UriTemplate property of the WebGet attribute you just added, indicating that a Uri ending with "titles" should be mapped to the GetBookTitles method.

```
[ServiceContract(Namespace = "http://develop.com")]
interface IBookService
{
    [OperationContract]
    [WebGet(UriTemplate = "titles")]
    List<BookTitle> GetBookTitles();
    ...
```

    d. Add a [WebGet] attribute to the GetBookInfo operation, specifying a UriTemplate property starting with "books" and accepting a query string that maps an isbn parameter to the a method parameter of the same name.
    e. Do the same for the GetBookPhoto method, replacing "books" with "photos" in the UriTemplate string.

```
[ServiceContract(Namespace = "http://develop.com")]
interface IBookService
{
    ...
    [OperationContract]
    [WebGet(UriTemplate = "books?isbn={isbn}")]
    BookInfo GetBookInfo(string isbn);

    [OperationContract]
    [WebGet(UriTemplate = "photos?isbn={isbn}")]
    Stream GetBookPhoto(string isbn);
    ...
```

3. Next you will need to configure the service endpoint to use the new wsHttpBinding binding, which will also require a <webHttp> endpoint behavior to encode messages as Plain Old XML (POX).

    a. First we're going to add an endpoint behavior so that we can reference it in the binding. To do this you will open the **App.config** file located in the **Service** project.

    b. Directly beneath the opening tag of the <behaviors> element, add a new <endpointBehaviors> section that includes a <behavior> element. Add a name attribute to the behavior, specifying "web" for the name. Then within the behavior add a <webHttp/> element.

```
<behaviors>
  <endpointBehaviors>
    <behavior name="web">
      <webHttp/>
    </behavior>
  </endpointBehaviors>
    ...
```

    c. Next you need to configure the endpoint to use **webHttpBinding** as the binding and **"web"** as the **behaviorConfiguration**.

```
<endpoint address=""
          binding="webHttpBinding"
          contract="WcfBindings.IBookService"
          behaviorConfiguration="web"/>
```

4. Now that you've implemented a WCF service using the web programming API, it's time to test it. First, make sure to set the **Service** project as the startup project (right-click and choose "Set as Startup Project). Then press Ctrl+F5 to start the service without debugging. If all goes well, you'll see a console app pop up stating that the service is running.

    a. Now open a browser (Internet Explorer will do nicely) and type the following as the address, then press Enter.

       http://localhost:1234/BookService

    b. Because metadata is turned on and httpGet has been enabled, you should see a friendly help page with a link to the WSDL file. That's a good start. Now add "/titles" to the end of the url and press Enter again.

       http://localhost:1234/BookService/titles

    c. You should now see an XML file with a series of <BookTitle> nodes, each with <Isbn> and <Title> nodes. Copy the first Isbn number, then change the url to end with books?isbn=0735621632. This should return BookInfo for CLR via C#.

       http://localhost:1234/BookService/books?isbn=0735621632

   d. Lastly, place “books” with “photos” in the url and press Enter again. This should display an image of the book’s cover.

      http://localhost:1234/BookService/photos?isbn=0735621632

5. You now need to write some code in order for the Windows Forms client to utilize the Books service using simple web protocols, without relying on WPF or SOAP messaging.
   a. Start by opening MainForm.cs (right-click on MainForm.cs in the Solution Explorer and select View Code. Then go to the **GetBookTitles** method, where you’ll be doing most your work.
   b. Create a Uri instance, passing the following string into the constructor: “http://localhost:1234/BookService/titles”.

```
private List<BookTitle> GetBookTitles()
{
    Uri uri = new Uri("http://localhost:1234/BookService/titles");
    ...
```

   c. Next create a web client and add a header setting the Content-Type to “application/xml”.

```
WebClient client = new WebClient();
client.Headers.Add("Content-Type", "application/xml; charset=utf-8");
```

   d. Then you’ll need to call DownloadString on the WebClient, passing the uri you just created. This will return a string containing the XML for all the book titles.

```
string xmlStr = client.DownloadString(uri);
```

   e. Simply pass this string into the BookTitle.ParseTitles method (the code already written for you). This will populate a List<BookTitle> and set it as the data source of the drop down list on the main form. Set the Client project as the startup and press Ctrl+F5 to run it. Clicking the Get button should fill the drop down with book titles.

6. Now that you’ve got a list of book titles for the drop down list, it’s time to retrieve the complete information for each book selected by the user. Fleshing out the **GetBookInfo** method will allow you to do just that.
   a. You’ll need to create another uri with which to invoke the Books web service. But this time you’ll utilize the **UriTemplate** class the build a uri that matches what is expected by the GetBookInfo operation of the IBookService contract.
   b. Start by creating a new Uri instance to serve as the base uri.

```
Uri baseUri = new Uri("http://localhost:1234/BookService");
```

   c. Then create a new UriTemplate object, supplying the same string expression to the constructor as with that supplied to the WebGet attribute of the GetBookInfo operation of the IBookService contract.

```
UriTemplate template = new UriTemplate("books?isbn={isbn}");
```

   d. Finally, call **BindByPosition** on the template, passing the base uri and the isbn number.

```
Uri boundUri = template.BindByPosition(baseUri, isbn);
```

   e. Next create a WebClient, add a header that sets the Content-Type to application/xml, and call DownloadString to obtain the XML for the book.

```
WebClient client = new WebClient();
client.Headers.Add("Content-Type", "application/xml; charset=utf-8");
string xmlStr = client.DownloadString(boundUri);
```

   f. Now run the Client application again. This time when you click the **Get** button you'll see the other fields on the form populated with other pieces of information pertaining to the book selected in the drop down list.

   g. If you have Internet connectivity, clicking the **Browse** button on the form will display the book's listing on Amazon.com in the web browser control.

7. The last step in this part of the exercise is to flesh out the **ShowBookPhoto** method in MainForm.cs. Here you simply build a uri using the UriTemplate class, just as you did in the last step, but using an expression for the template that matches the **GetBookPhoto** operation of the IBookService contract.

   a. Create a base uri, then a UriTemplate with the correct expression, followed by a call to the template's BindByPosition method that accepts the base uri and the isbn number.

```
Uri baseUri = new Uri("http://localhost:1234/BookService");
UriTemplate template = new UriTemplate("photos?isbn={isbn}");
Uri boundUri = template.BindByPosition(baseUri, isbn);
```

   b. Then call Navigate on webBrowser1, passing boundUri.

```
webBrowser1.Navigate(boundUri);
```

   c. Now when you run the Client application, you'll see an image of the book cover displayed each time to select a book from the drop down list.

Solutions:

The full solution for this lab is available at After\Part1.

Part 2 – Create a Syndication Feed

In this part you'll create a syndication feed using the new .NET 3.5 Syndication API. Like other web-oriented WCF services, you'll need to configure an endpoint to use the wsHttpBinding binding, which in turn needs to reference an endpoint behavior with a <wsHttp/> element in order to serve Plain Old XML (POX). In addition, however, you'll need a service contract with an operation that returns an object of type SyndicationFeedFormatter, which is an abstract class. Consequently, the method that implements the service contract interface needs to select a derived type. Because both Rss20FeedFormatter and Atom10FeedFormatter come out of the box with .NET 3.5, you can select which one to use depending on a method parameter, which you can map with a UriTemplate.

Steps:

1. First, go ahead and run the Service application by pressing Ctrl+F5. You'll just see a console app appear, which states that the service is up and running. Then open up a browser and navigate to the service base address: http://localhost:1234/BookService. Because metadata with httpGet is enabled, you should see a nice HTML page with a WSDL link.

2. Close the console app by pressing any key. Then open the App.config file in the Service project. There you'll see a TODO item asking you to add an <endpointBehaviors> element with a <behavior> element called "xml" that has a **<webHttp/>** element.

   a. Do as it says and add the **<endpointBehaviors>** element.

```
<endpointBehaviors>
  <behavior name="xml">
    <webHttp/>
  </behavior>
</endpointBehaviors>
```

   b. Then scroll to the lower portion of the App.config file, and change the endpoint binding from basicHttpBinding to **webHttpBinding**.

   c. After that, add a **behaviorConfiguration** attribute to the endpoint referencing the **"xml"** endpoint behavior.

```
<endpoint address=""
          binding="basicHttpBinding"
          contract="WcfBindings.IBookService"
          behaviorConfiguration="xml"/>
```

   d. Then scroll to the lower portion of the App.config file, and change the endpoint binding from basicHttpBinding to **webHttpBinding**.

3. Open the BookService.cs file in the Service project, so that you can start adding some attributes to the service contract interface.

   a. First you'll need to add a couple more attributes to the IBookService interface. That's because the GetRssFeed method of the BookService class will return a class that *derives from* the abstract base class SyndicationFeedFormatter.

   b. You'll need to add two [ServiceKnownType] attributes, one that supplies the type of Rss20FeedFormatter, and another for the type of Atom10FeedFormatter.

```
[ServiceContract(Namespace = "http://develop.com")]
[ServiceKnownType(typeof(Rss20FeedFormatter))]
[ServiceKnownType(typeof(Atom10FeedFormatter))]
interface IBookService
{
    ...
```

   c. In order to expose the GetRssFeed operation as POX you must adorn it with a [WebGet] attribute. You should also specify a UriTemplate property for the attribute, with a placeholder corresponding to the format parameter.

```
[OperationContract]
[WebGet(UriTemplate = "{format}")]
SyndicationFeedFormatter GetRssFeed(string format);
```

4. In spite of all your work thus far, your syndication service will not be functional until you flesh out the BookService's GetRssFeed method.
   a. The first line of code, which has been written for you, obtains a List of Books from an xml file.
   b. The second line of code, which has also been provided, is a LINQ query that returns an **IEnumerable<SyndicationItem>,** using the books collection as its source. IEnumerable<SyndicationItem> is simply an iterators that streams a series of items in order to minimize memory footprint.
   c. Now you get to write some code. Create a new SyndicationFeed object, then set the feed title by passing it a new TextSyndicationContent with the feed title. Then set the Items property of the feed to the **IEnumerable<SyndicationItem>** feedItems created in the previous line of code.

```
SyndicationFeed feed = new SyndicationFeed();
feed.Title = new TextSyndicationContent("Best .Net Books");
feed.Items = feedItems;
```

   d. The last part is easy. Just switch on the format method parameter, returning a new Atom10FeedFormatter for "atom" and a new Rss20FeedFormatter for "rss". Pass the SyndicationFeed object to the constructor for each of these.
   e. For the default switch case, simply throw an ArgumentException.

```
SyndicationFeedFormatter formatter = null;
switch (format)
{
    case "atom":
        return new Atom10FeedFormatter(feed);
    case "rss":
        return new Rss20FeedFormatter(feed);
    default:
        throw new ArgumentException("Unsupported feed format: " +
            format);
}
```

   f. Remove the placeholder code that returns null.

5. That's it! You're ready to run your very first syndication feed using the new .NET 3.5 syndication API. There is much more to the API than this simple exercise, but you're off to a good start.

   a. Press Ctrl+F5 to run the service. Then open up Internet Explorer and navigate to the following url:

      http://localhost:1234/BookService/atom

   b. You'll see the feed displayed nicely in the browser. Clicking the green arrow for an item will navigate to the item link, which is the amazon.com page listing for the book.

   c. Now replace atom with rss as the format, and you'll see the feed displayed in a slightly different manner.

      http://localhost:1234/BookService/rss

   d. If you want to see the actual XML for the feed, from within Visual Studio select File, Open, and then past in either of these two urls. Visual Studio will then display the file in its XML Editor.

Solutions:

The full solution for this lab is available at After\Part2.

Part 3 – Create a WCF service that sends and receives JSON encoded objects for use with an ASP.NET AJAX application (optional)

Your job in this part of the exercise will be to create a WCF service that can send and receive objects encoded in JSON (JavaScript Object Notation), so that an ASP.NET AJAX application can invoke the service using an ASP.NET ScriptManager that generates a JavaScript proxy. The AJAX web application has been written for you, so all you need to do is configure the Books Service to support JSON.

Notes:

- Before starting this part of the exercise, you will need to create a web application in IIS with a virtual directory pointing to the directory containing the project files. This will require admin rights on your local machine.
- From the Windows Start Menu select Programs, Administrative Tools, then open **Internet Information Services (IIS) Manager**. Expand the **Default Web Sites** node (under Web Sites), then right-click on the node and choose **Add Application**. For the Alias type "WcfAjax" and enter the full path for the project directory: Before\WcfRest-Part3\WcfAjax.
- Next, select IIS Authentication for the WcfAjax application, then right-click **Windows Authentication** and select **Disable**.
- Lastly, you will need to grant file access permission to the Network Service account to modify the books.xml file. From Windows Explorer locate the **books.xml** file in the **WcfAjax** directory, right-click and select Properties, then select the **Security** tab. Click the Edit button, highlight NETWORK SERVICE, then check the box under **Allow** for the **Modify** permission.

Steps:

1. Open the WcfRest.sln Visual Studio solution located at Before\Part3, which contains the **WcfAjax** web project. Right-click Service.svc and select Set As Start Page, then press F5 to start the web app in debug mode. You should see the familiar HTML help page with a link to the WSDL file. Close the browser to end the debug session so that you can modify the service.
    a. From within the App_Code folder, open **BookService.cs**. We're going to begin with the **GetBookPhoto** operation from the IPhotoService contract. Start by adding a [WebGet] attribute directly beneath the [ServiceContract] attribute. Specify a **UriTemplate** property with a placeholder for **{isbn}** that will map an isbn number at the end of the url to the isbn method parameter.

```
[ServiceContract(Namespace = "http://develop.com")]
interface IPhotoService
{
    [OperationContract]
    [WebGet(UriTemplate = "{isbn}")]
    Stream GetBookPhoto(string isbn);
}
```

b. Next open the web.config file and scroll to the bottom. There you'll see the <system.serviceModel> section. Add an **<endpointBehaviors>** element beneath the <serviceBehaviors> element. It should contain a <behavior> named "xml" that has a **<webHttp/>** element.

```
<endpointBehaviors>
    <behavior name="xml">
        <webHttp/>
    </behavior>
</endpointBehaviors>
```

c. Add a **behaviorConfiguration** attribute to the first endpoint for the service that points to the **"xml"** endpoint behavior you just added.

```
<endpoint address=""
          binding="webHttpBinding"
          contract="IBookService"
          behaviorConfiguration="xml"/>
```

d. Press Ctrl+F5 to run the web app, which will open a browser with the address: http://localhost/WcfAjax/Service.svc. Can you guess what text you will need to append to the address in order to invoke the GetBookPhoto operation? Hint: look in web.config to locate the address of the IPhotoService endpoint. Then add the placeholder you specified in the UriTemplate for the WebGet attribute. Use the following isbn: 0735621632.

e. The url you should type is: http://localhost/WcfAjax/Service.svc/photos/0735621632. This will display the image for the CLR via C# book cover.

2. Go back to the BookService.cs file and add the [WebGet] attribute for the first two methods of IBookService: GetBookTitles and GetBookInfo.

   a. You won't need to specify a UriTemplate for either method because the JavaScript proxy generated by the ASP.NET AXAJ ScriptManager doesn't have any notion of Uri Templates.

   b. For SaveBookInfo add a [WebInvoke] attribute, because this method will be mapped to an HTTP POST request.

```
[ServiceContract(Namespace = "http://develop.com")]
interface IBookService
{
    [OperationContract, WebGet]
    List<BookTitle> GetBookTitles();

   [OperationContract, WebGet]
    BookInfo GetBookInfo(string isbn);

     [OperationContract, WebInvoke]
    void SaveBookInfo(BookInfo book);
}
```

   c. Add a **behaviorConfiguration** attribute to the first endpoint for the service that points to the **"xml"** endpoint behavior you just added.

3. Open web.config once again and add an endpoint behavior called "json" that has an <enableWebScript/> element. Then reference the behavior from the behaviorConfiguration attribute of the IBookService endpoint.

   a. This will result in serialization of BookTitle and BookInfo objects as JSON instead of XML. This relieves AJAX clients from having to parse XML and instead directly utilize the JavaScript objects.

4. Right-click on Default.aspx and select View in Browser. This will launch Internet Explorer and navigate to this page.

   a. Click on the "**Get Books**" button. This will make an AJAX call to the WCF Books Service to retrieve a list of BookTitle objects encoded in JSON format, which are then added to the drop down list. This is all done without posting the page back to the web server.

   b. Selecting a different book title from the drop down list will also result in an AJAX call to the WCF service, retrieving an individual Book object encoded in JSON format. The src of the image is set to a url that is mapped to the GetBookPhoto method of IPhotoService.

5. Now try the price of one of the books and clicking the "**Save Book**" button. The "Book saved" message should appear. But when you select another book from the list and then return to the saved book, you'll notice that the field reverts to the original value. The reason for this is that the browser has cached the original book object and retrieved the item from cache rather than obtaining it from the server.

   a. To remedy this situation you need to add a header to the HTTP response setting **cache-control** to **no-store**. You can access the WebOperationContext from within a method on the service to add the header to the outgoing response.

   b. Open BookService.cs and go to the GetBookInfo method. Uncomment the following line of code.

```
WebOperationContext.Current.OutgoingResponse.Headers["cache-control"] =
"no-store";
```

   c. Save the file and view Default.aspx once again in the browser. If you change a book price again, this time the change will persist when you return to the item because object caching in the browser has been disabled for BookInfo objects.

Solutions:

The full solution for this lab is available at After\Part3.