REST

Estimated time for completion: 45 minutes

Overview:
In this lab you will be creating a contact manager REST service. The service will support multiple response data formats. You will also be creating a client that can consume the service.

Goals:
- Learn how to expose resources via REST using the WebAPI
- Learn how to consume a REST service from C#
- Learn how to customize the handling of media types

Lab Notes:
This lab will be using Visual Studio 2012.

Part 1: Exposing a Resource as a Service

In the first part of the lab you will create the initial cut of the service modeling the functionality as resources accessible via HTTP. You will also use Firefox and a custom client to consume that service so seeing content type negotiation in action

1. In Visual Studio 2012, open the starter solution **before/REST.sln.** In the solution there are two existing projects: Repository which provides the underlying model that you will be using and Hypermedia which is used in part 3 of the lab
2. Look at the `Contact` class in the Repository project. This is the resource you will be working with
3. Add a new a New Project to the solution of type MVC4 application (this is in the Web section of the new project dialog) called `Contacts`
4. In the wizard select the Web API project type, leave the rest of the settings as their defaults and press OK
5. In the newly created project add a reference to the Repository project so we have some data to work with
6. In the **Controllers** folder rename **ValuesController.cs** to **ContactsController.cs** (say Yes to also rename the class)
7. In the `ContactsController` class change the method signatures to use the `Contact` resource rather than `string`
    a. Change the first `Get` to return `IEnumerable<Contact>`
    b. Change the second `Get` to return `Contact`
    c. Change the `Post` to take a `Contact`

d. Change the `Put` to take an `id` of type `int` and `name`, `email` and `notes` of type `string`

8. Create a member variable in the `ContactsController` of type `IContactRepository` called `repository` and initialize it to a new instance of `ContactRepository`
9. Implement the first `Get` by returning `repository.All`
10. Implement the second `Get` by returning `respository.GetById`
11. Implement `Post` by calling `repository.Add` – this will be for inserting new items into the repository
12. Implement `Put` by call `respository.GetById` and then updating the fields. This will be for updating existing contacts
13. Implement `Delete` by calling `repository.Delete`
14. Run the Contacts project to start IIS Express
15. Open Firefox and browse to **api/contacts** uri. You should see the data returned as XML. This is because Firefox passes an `Accept` header of `application/xml`
16. Browse to **api/contacts/2** and you should see the single contact for Andy Clymer
17. Browse to api/contacts/42 and you should see an empty nil contact. This is a problem as you have asked for a resource that does not exist and so really the service should return an HTTP status code of 404 (not found). Change the `Get` that takes an `id` to check that if the respository returns `null` then throw a new `HttpResponseException` containing a message with a Not Found status code

```
public Contact Get(int id)
{
    Contact contact = repository.GetById(id);
    if (contact == null)
    {
        throw new HttpResponseException(new
                        HttpResponseMessage(HttpStatusCode.NotFound));
    }

    return contact;
}
```

18. Your service is now working. Next we will build a client to talk to the service
19. Add a console application project to the solution called `Client`
20. In the new `Client` project add references to The ASP.NET WebAPI via NuGet (Manage NuGet Packages). This will also add a reference to the Newtonsoft Json parser.
21. In `Main` create an instance of the `HttpClient` class called `client`. Set its `BaseAddress` to **"http://localhost:<port number>/api/"** where the `<port number>` is the one being used by IIS Express to host the site
22. All of the potentially blocking methods in `HttpClient` are asynchronous so put a `Console.WriteLine/Console.ReadLine` pair of calls at the end of `Main` to stop the process exiting
23. Create a `static` method called `PrintContacts` that returns `void` and takes an `HttpClient`, called `client`, and mark it as `async`

24. In the `PrintContacts` method add an `Accept` header to the `DefaultRequestHeaders` property of the client. This will trigger the content type negotiation in Web API to return JSON rather than XML

```
client.DefaultRequestHeaders.Accept.Add(new
   MediaTypeWithQualityHeaderValue("application/json"));
```

25. Await a call to `client.GetAsync` passing in the additional part of the Uri, **contacts**. Assign the result to a variable of type `HttpResponseMessage`
26. Await a call to the `ReadAsStringAsync` method of the `Content` property of the `HttpResponseMessage`. Assign the result to a `string` called `contactsString`.
27. We will now parse out the returned JSON using the dynamic support introduced in C# 4.0. Create a `dynamic` variable called `contacts` and assign to it the result of a call to `JArray.Parse` passing in the `contactsString`.
28. Create a `foreach` loop again using a `dynamic` variable on the `contacts` collection and write out the `Name` and `Email` property of each contact.

```
HttpResponseMessage responseMessage = await client.GetAsync("contacts");

string contactsString  = await responseMessage.Content.ReadAsStringAsync();

dynamic contacts = JArray.Parse(contactsString);

foreach (dynamic contact in contacts)
{
    Console.WriteLine("{0} : {1}", contact.Name, contact.Email);
}
```

29. Call the `PrintContacts` method from `Main`
30. Compile and test your code. You should see the three contacts details printed
31. As an optional exercise try adding a new contact using the `PostAsync` method of the `HttpClient`. You can create the Json using a `JObject` and use a `StringContent` to set the content of the `HttpRequestMessage` from the `JObject`. Remember to specify the `ContentType` of the content as `application/json`

Solution: **After/Part1/Rest.sln**

Part 2: Taking Control of the Resource Formatting

If you look at the output of getting the contacts in Firefox you can see that the output is not ideal. The XML namespaces have the .NET namespace of the Contact class and the collection is called ArrayOfContact – Contacts would be a better root element name in this scenario. Rather than annotate the resource itself we will change the way it is projected to XML clients using a custom MediaTypeFormatter

1. In the `Contacts` project create a new folder called `Formatters`
2. Add a class into the `Formatters` folder called `ContactsFormatter` and derive this class from `MediaTypeFormatter`

3. Implement the two abstract members of `MediaTypeFormatter` (Ctrl . -> Enter will do this for you)
4. This formatter will only be writing out content so leave the `CanReadType` as the default implementation. We will now implement `CanWriteType`
5. We want to be able to format both `Contact`s and collections of `Contact`s, We therefore need to say we can format anthing that is collection-like (implements `IEnumerable<Contact>`) or a `Contact`. We can use the `IsAssignableFrom` member of `System.Type` for the first and just compare type objects for the second

```
public override bool CanWriteType(Type type)
{
    return typeof(IEnumerable<Contact>).IsAssignableFrom(type) ||
           type == typeof(Contact);
}
```

6. Next we need to implement the serialization. We do this by overriding the `WriteToStreamAsync` method. Notice the method is an asynchronous one so we will need to create a `Task` and return that. Return the result of `Task.Run` and do the following in the lambda expression
    a. Test the incoming `type` – if it can be assigned to `IEnumerable<Contact>` call a method `WriteToStream` passing the `stream` and the `value` cast to `IEnumerable<Contact>`
    b. If the type if a `Contact` then call a method `WriteToStream` passing the `stream` and the `value` cast to `Contact`
    c. Implement these two methods

```
public override Task WriteToStreamAsync(Type type, object value,
                                        Stream stream,
                                        HttpContentHeaders contentHeaders,
                                        TransportContext transportContext)
{
    return Task.Run(() =>
    {
        if (typeof(IEnumerable<Contact>).IsAssignableFrom(type))
        {
            WriteToStream(stream, (IEnumerable<Contact>)value);
        }
        else if (type == typeof(Contact))
        {
            WriteToStream(stream, (Contact)value);
        }
    });
}
```

7. Implement the version of `WriteToStream` that takes a `Contact` by calling a method `CreateContactXml` that takes the `contact` and returns an `XElement` then call `Save` on the returned `XElement` passing the `stream`.
8. Implement the `CreateContactXml` by creating XML for the `contact` using LINQ to XML

```
private void WriteToStream(Stream stream, Contact contact)
{
    XElement contactXml = CreateContactXml(contact);

    contactXml.Save(stream);
}

private XElement CreateContactXml(Contact contact)
{
    return new XElement("Contact",
                new XElement("Id", contact.Id),
                new XElement("Name", contact.Name),
                new XElement("Email", contact.Email),
                new XElement("Notes", contact.Notes));
}
```

9. Implement the version of `WriteToStream` that takes the `IEnumerable<Contact>`. Use LINQ to XML to create the XML for the collection: create an element called `Contacts` and then create its content using a LINQ expression that projects using the `CreateContactXml` you implemented earlier. `Save` this created `XElement` to the stream

```
private void WriteToStream(Stream stream, IEnumerable<Contact> contacts)
{
    XElement contactsXml = new XElement("Contacts",
                                        from c in contacts
                                        select CreateContactXml(c));

    contactsXml.Save(stream);
}
```

10. Now we are formatting as XML therefore we need to ensure that this formatter is only invoked if the client has asked for XML. Create a constructor and add `application/xml` to the `SupportedMediaTypes` property
11. Finally we need to tell the infrastructure about our new formatter. Go to **global.asax.cs** and after the code currently in `Application_Start` add the following code. Notice we use `Insert` to ensure this formatter gets used in preference to the standard one for our supported types

```
GlobalConfiguration.Configuration
                   .Formatters
                   .Insert(0, new ContactsFormatter());
```

Solution: **After/Part2/Rest.sln**

Part 3: Introducing Hypermedia (Optional)

So far we have exposed a resource via HTTP but one of the major tenets of REST to hypermedia – that state transitions are encoded as links. In this part of the lab we will introduce hypermedia support by wrapping the resource and customizing the formatting.

1. Review the `Resource<T>` class in the `Hypermedia` project – this is what we will use to support links for our `Contact` resource
2. In the `Contacts` project add a reference to the `Hypermedia` project
3. Change the `Get` methods in the `ContactsController` to use `Resource<Contact>` rather than `Contact`
4. We could hard code the links, but to be more robust we should probably base them on the uri that the client has used to invoke the service. For this we need to integrate with HTTP more closely so change the two `Get` methods to take an `HttpRequestMessage`, called `request`, as the first (or only) parameter
5. In the `Get` that takes an `id`, if the `Contact` is found, wrap it in a `Resource<Contact>` object and pass in a dictionary of links: `self` mapping to the request uri and `parent` mapping to the collection uri (you can build this using the `UriBuilder` class)
6. In the `Get` for the collection use LINQ to project `Resource<Contact>` objects with a single link called `self` poining to the uri for the contact (use the request uri and append the contact's `Id`)

```
public IEnumerable<Resource<Contact>> Get(HttpRequestMessage request)
{
    return repository.All
               .Select(c => new Resource<Contact>(c,
                   new Dictionary<string,string>()
                   {
                      {"self", request.RequestUri.AbsoluteUri + "/" + c.Id},
                   }));
}

public Resource<Contact> Get(HttpRequestMessage request, int id)
{
    Contact contact = repository.GetById(id);
    if (contact == null)
    {
       throw new HttpResponseException(new
                      HttpResponseMessage(HttpStatusCode.NotFound));
    }

    Uri requestUri = request.RequestUri;

    var parentBuilder = new UriBuilder(requestUri.Scheme,
                                       requestUri.Host,
                                       requestUri.Port,
                                       "api/contacts");
    var resource = new Resource<Contact>(contact,
                new Dictionary<string,string>()
                {
                    {"self", requestUri.AbsoluteUri},
          {"parent", parentBuilder.Uri.AbsoluteUri},
   });
   return resource;
}
```

7. If you run the client you will see it no longer produces the desired output – the format of the returned data has changed. We will now fix the client to show not only the contact name but also the links

```
foreach (dynamic contact in contacts)
{
    Console.WriteLine("{0} : {1}", contact.ResourceValue.Name,
                                   contact.ResourceValue.Email);
    foreach (dynamic link in contact.Links)
    {
        Console.WriteLine("\t{0} : {1}", (string)link.Name,
                                         (string)link.Uri);
    }
}
```

8. If you load the contacts in Firefox the service now returns a 500 – the `DataContractSerializer` doesn't know how to serialize a `Resource<T>`. We need to create another formatter that can write a `Resource<Contact>` - fortunately the implementation is very similar to the one we have already built so copy that class as `ContactResourceFormatter` fixing up the constructor and changing all usages of `Contact` to `Resource<Contact>`.
9. In the new `CreateContactXml` add elements for each link setting the `Name` and `Uri` as attributes of the `Link` element

```
private XElement CreateContactXml(Resource<Contact> resource)
{
    return new XElement("Contact",
              new XElement("Id", resource.ResourceValue.Id),
              new XElement("Name", resource.ResourceValue.Name),
              new XElement("Email", resource.ResourceValue.Email),
              new XElement("Notes", resource.ResourceValue.Notes),
              from l in resource.Links
              select new XElement("Link",
                    new XAttribute("Name", l.Name),
                    new XAttribute("Uri", l.Uri)));

}
```

10. Add the formatter to the `Formatters` collection in **Global.asax.cs** at the start of the collection as before
11. Use firefox to hit the uri again and you should now see the data formatted correctly, now with the additional links

Solutions

after\Part3\REST.sln