Restifying a WCF Service

WCF is a complex beast and sometimes not very suited for a light weight RESTful http API. But you could well have pre-existing WCF services that you might want to reuse and turn them restful. We will have a simple WCF service here as a demo on the steps needed.

Let’s begin at the beginning and have a look at the service. We will not delve much on details but just show the service here.

1 Creation of the demo WCF service

For the purposes of this tutorial we’re not going to decouple the parts of the WCF service (where, if we had multiple services, we could put the interfaces together into one dll, services implementations into another, etc…).

1.1 The interface

```
namespace RestfulWCF
{
    [ServiceContract]
    public interface IEmployeeService
    {

        [OperationContract]
        Employee Get ( string value );

    }
}
```

1.2 The implementation

```
    public class EmployeeService : IEmployeeService
    {
        public Employee Get ( string value )
        {
            return new Employee ()
            {
                ID = value,
                Name = "John Doe"
            };
        }

    }
```

1.3 Dummy Business Class

And the dummy business class used in the sample (defined in a separate .dll) with the usual attributes for exposing the type to serialization.

```
namespace BusinessObjects
{
    [DataContract]
    public class Employee
    {
        [DataMember]
        public string ID { get; set; }

        [DataMember]
        public string Name { get; set; }

    }
}
```

In order to use the `[DataContract]` and `[DataMember]` attributes you need to add a reference to `System.Runtime.Serialization` in your business .dll project.

1.4 The Web.Config file

For the moment, we leave the web.config as created by the default configuration of the newly created project, but we will have to make changes to adapt the service to a restful orientation.

We now have a working simple WCF project, let's proceed to restify it.

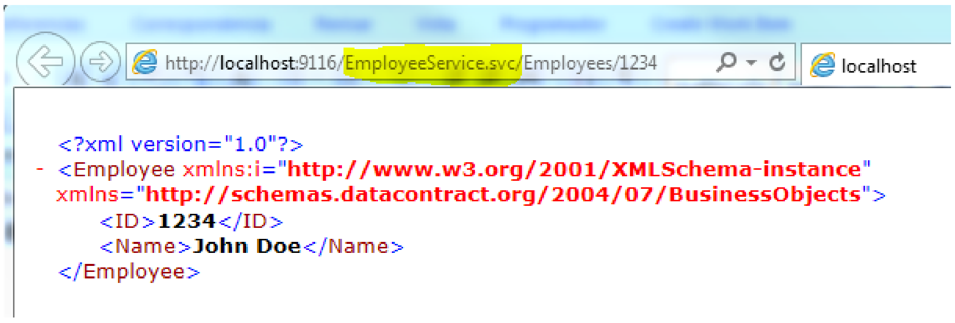
2 Restifying the WCF service

2.1 Add the WebGet attribute

Proceed to modify the web service interface as shown here:

```
[ServiceContract]
public interface IEmployeeService
{

    [OperationContract]
    [WebGet(UriTemplate="/Employees/{value}")]
    Employee Get ( string value );

}
```

We have added the highlighted line to define an URI for the service. `WebGet` Is defined in the `System.ServiceModel.Web` namespace. We press F5 and the have the service running.



http://localhost:9116/EmployeeService.svc/Employees/1234

```
<?xml version="1.0"?>
- <Employee xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://schemas.datacontract.org/2004/07/BusinessObjects">
    <ID>1234</ID>
    <Name>John Doe</Name>
  </Employee>
```

The thing is that we can access the Employees URI that we defined in our interface previously but still have a very ugly thing in the middle, the name of the service in the middle of the URL makes this service not very restful, the ideal URL we're trying to get is obviously `http://server/Employees/1234`, with nothing else in between. Enter routing.

2.2 Adding routing

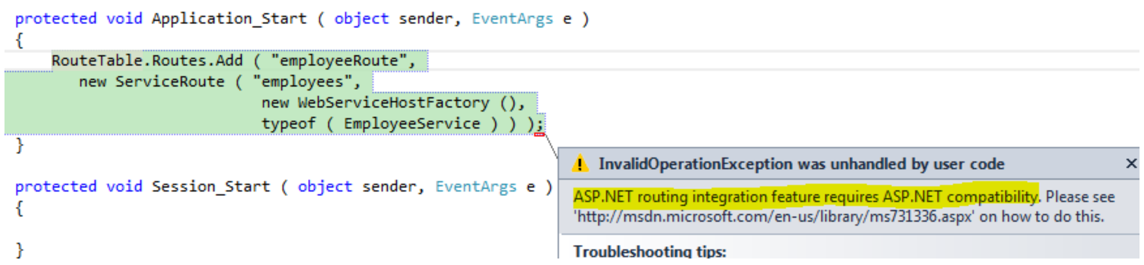
The first thing we need to have routing is to add a `global.asax` file to our project, if we did not have one (as in this case).

Add also a reference to `System.ServiceModel.Activation` in your project, because we will need to use that to configure routing.

Now, modify the `Application_Start` like so, adding one route for our new service (make sure to use the implementation, `EmployeeService`, not the interface, `IEmployeeService`, when defining the service type):

```
        protected void Application_Start ( object sender, EventArgs e )
        {

            RouteTable.Routes.Add ( "employeeRoute",
                new ServiceRoute ( "employees",
                                    new WebServiceHostFactory (),
                                    typeof ( EmployeeService ) ) );

        }
```

Press F5 to test your brand new routed service, but bang! we find this exception:



```
protected void Application_Start ( object sender, EventArgs e )
{
    RouteTable.Routes.Add ( "employeeRoute",
        new ServiceRoute ( "employees",
                            new WebServiceHostFactory (),
                            typeof ( EmployeeService ) ) );
}

protected void Session_Start ( object sender, EventArgs e )
{

}
```

InvalidOperationException was unhandled by user code

ASP.NET routing integration feature requires ASP.NET compatibility. Please see 'http://msdn.microsoft.com/en-us/library/ms731336.aspx' on how to do this.

Troubleshooting tips:

So we need to modify our service implementation accordingly.

```
    [AspNetCompatibilityRequirements(
                RequirementsMode=AspNetCompatibilityRequirementsMode.Required)]
    public class EmployeeService : IEmployeeService
    {

        public Employee Get ( string value )
        {
            return new Employee ()
            {
                ID = value,
                Name = "John Doe"
            };
        }

    }
```

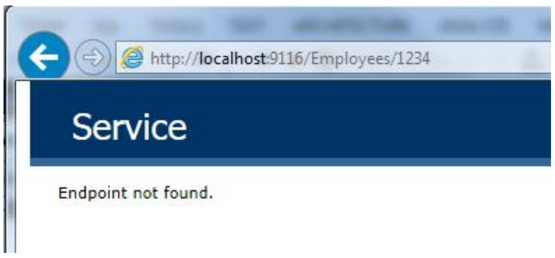
This attribute is also to be found in the library that we added a reference to earlier (`System.ServiceModel.Activation`). You will need to modify your web.config accordingly, by modifyin the `<system.webServer>` section in order to support routing:

```
<system.webServer>
    <modules runAllManagedModulesForAllRequests="true">
      <add name="UrlRoutingModule"
        type="System.Web.Routing.UrlRoutingModule, System.Web, Version=4.0.0.0,
              Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
    </modules>
    <handlers>
      <add name="UrlRoutingHandler" preCondition="integratedMode" verb="*"
path="UrlRouting.axd"/>
    </handlers>
    <directoryBrowse enabled="true"/>
  </system.webServer>
```

And also make sure your `serviceHostingEnvironment` contains also the `aspNetCompatibilityEnabled` attribute:

```
      <serviceHostingEnvironment multipleSiteBindingsEnabled = "true"
                                 aspNetCompatibilityEnabled = "true"/>
```

We press F5 again to view our newly restified WCF, but oh! something is still wrong

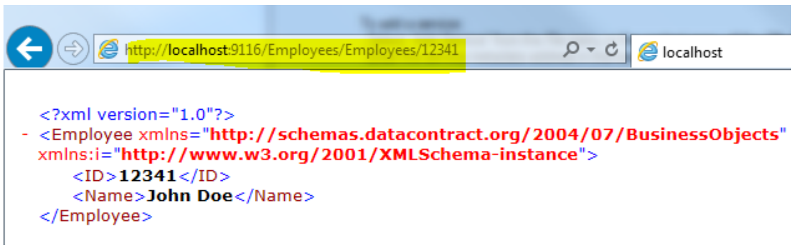


http://localhost:9116/Employees/1234

Service

Endpoint not found.

The real URL where the service is listening is actually `http://localhost:9116/Employees/Employees/12341`, but why?


http://localhost:9116/Employees/Employees/12341

```
  <?xml version="1.0"?>
- <Employee xmlns="http://schemas.datacontract.org/2004/07/BusinessObjects"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
     <ID>12341</ID>
     <Name>John Doe</Name>
  </Employee>
```

The reason is that we have defined "employees" as part of the URL twice, both in the attribute of the service and on the routing defined in the Global.asax file.

```
[ServiceContract]
public interface IEmployeeService
{

    [OperationContract]
    [WebGet(UriTemplate="/Employees/{value}")]
    Employee Get ( string value );

}
```

```
 RouteTable.Routes.Add ( "employeeRoute",
     new ServiceRoute (   "employees",
                          new WebServiceHostFactory (),
                          typeof ( EmployeeService ) ) );
```

Note: the name "employeeRoute" is irrelevant, as it is just an optional identifier for this entry in the routing table. The Important thing is that we defined these double, so to say, url fragment. So we will make some changes to that.

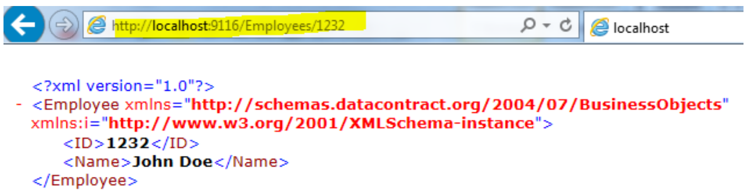
We can obviously remove one of them, and get the URL that we wanted. Either this

```
RouteTable.Routes.Add ( "employeeRoute",
     new ServiceRoute (  "",
                          new WebServiceHostFactory (),
                          typeof ( EmployeeService ) ) );
```

Or this

```
[OperationContract]
[WebGet(UriTemplate="/{value}")]
Employee Get ( string value );
```

will work to get this URL


http://localhost:9116/Employees/1232

```
  <?xml version="1.0"?>
- <Employee xmlns="http://schemas.datacontract.org/2004/07/BusinessObjects"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
     <ID>1232</ID>
     <Name>John Doe</Name>
  </Employee>
```

Which is what we intended to have from the very beginning. However this URL is only good if we're dealing with only one service that is not part of a wider set of services.

And even in that case, a web service being a public facing API, you should seriously consider leveraging this configuration mechanism to introduce versioning.

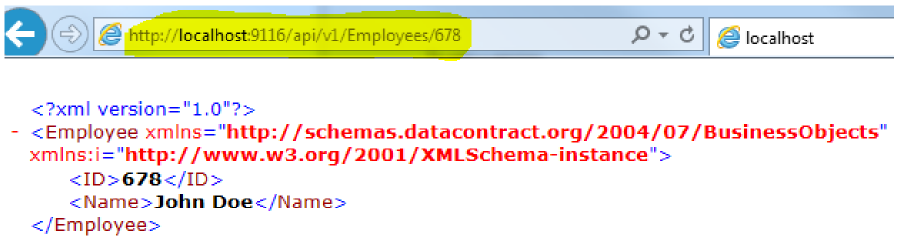
2.3 Adding Versioning

So, instead of eliminating one of the two URL fragments that we had before (called "employees"), we could actually use one of them to introduce versioning in the URL, so we can release newer versions of our service without breaking preexisting clients.

So, for example we could configure our Routing mechanism in the `Global.asax` file like this (leave the "employees" part in the configuration of the `WebGet` attribute in the interface definition)

```
RouteTable.Routes.Add ( "employeeRoute",
    new ServiceRoute (  "api/v1",
                        new WebServiceHostFactory (),
                        typeof ( EmployeeService ) ) );
```

So, we now have this URL



```
<?xml version="1.0"?>
- <Employee xmlns="http://schemas.datacontract.org/2004/07/BusinessObjects"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
    <ID>678</ID>
    <Name>John Doe</Name>
  </Employee>
```

Note on versioning: there are several ways to include versioning in our APIs, such as adding a timestamp, an extra parameter in the querystring or including one in the URL, as in this example. In general, it is recommended to use this one as it is easier to type and it is less obstrusive than the following examples:

- /server/2012-01-01/employees
- /server/employees?v=1.0

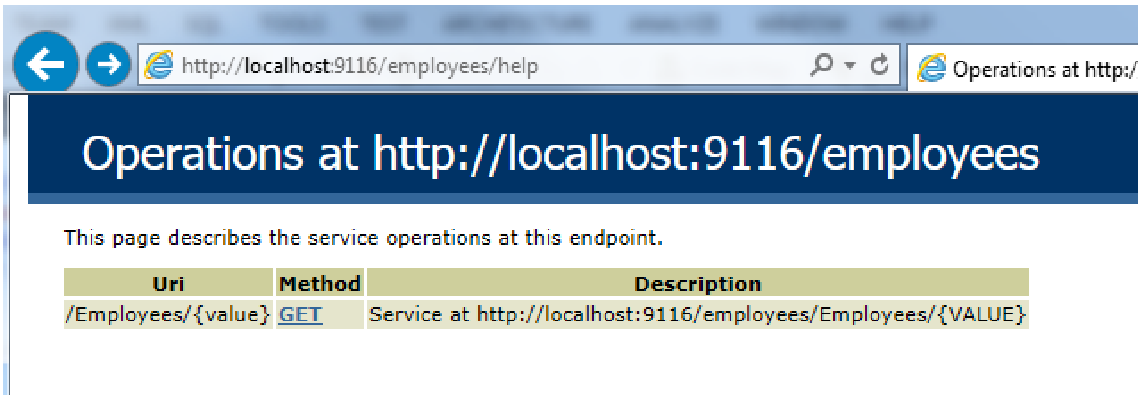
In general it is also recommended to use literals such as v1, v2 etc, since having v1.0 or v1.2 in the middle of the URL suggests too many changes and it feels somewhat awkward to have to type this extra dot.

2.4 Enabling automatic help (optional step)

We can actually enable automatic help pages provided by the WCF infrastructure by setting the `helpEnabled` attribute to true in the `webHttp` element of web.config file. This can be helpful when trying to discover usage of the service interface. Update your web.config to reflect this change:

```
 <endpointBehaviors>
  <behavior name="restfulBehaviour">
    <webHttp helpEnabled="true"  />
  </behavior>
</endpointBehaviors>
```

Now you can type the URL of your service ending the url with the word “help”, so you can get this screen:



Operations at http://localhost:9116/employees

This page describes the service operations at this endpoint.

Uri	Method	Description
/Employees/{value}	GET	Service at http://localhost:9116/employees/Employees/{VALUE}

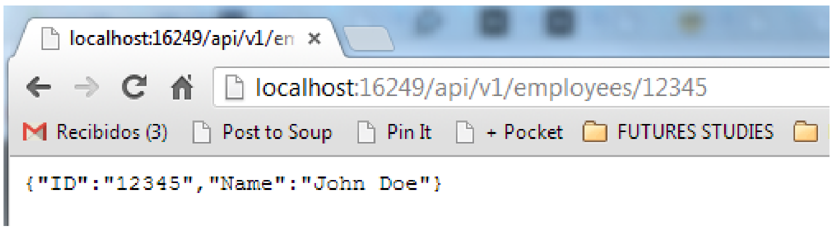
3 What about Json?

So far, we've seen that all the examples show the data in xml, but xml is verbose and has fallen a bit out of favour in modern APIs, and all the smart people prefer json.

Fortunately, it's just a matter of changing a bit our `WebGet` attribute, in the service interface definition:

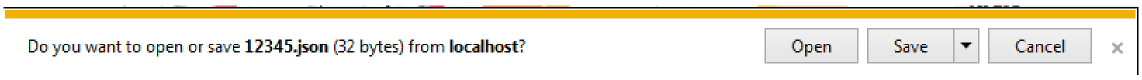
```
[OperationContract]
 [WebGet ( UriTemplate = "/Employees/{value}",
                         ResponseFormat=WebMessageFormat.Json)]
 Employee Get ( string value );
```

If we open Chrome now and call the service, we will see the json on screen:


localhost:16249/api/v1/en
localhost:16249/api/v1/employees/12345
Recibidos (3)
Post to Soup
Pin It
+ Pocket
FUTURES STUDIES

```
{"ID":"12345","Name":"John Doe"}
```

IE behaves a bit different, and prompts us to download the file. Seems it does not know what to do exactly with it.


Do you want to open or save **12345.json** (32 bytes) from **localhost**?

But the modification was indeed easy.

Note: to convince IE to show json on screen, there are several options here.

4 Modifying the restified service

The service is very simple, so let's say that we need now to keep a list of projects for each Employee, so let's just pretend we're getting those from the database, but first let's create a `Project` class and add a list of these to our `Employee` class.

```
    [DataContract]
    public class Project
    {
        [DataMember]
        public string ProjectID { get; set; }

        [DataMember]
        public string ProjectName { get; set; }

        [DataMember]
        public bool IsTopSecret { get; private set; }

        public Project ( string name, string id, bool isTopSecret )
        {
            this.ProjectName = name;
            this.ProjectID = id;
            this.IsTopSecret = isTopSecret;
        }
    }
```

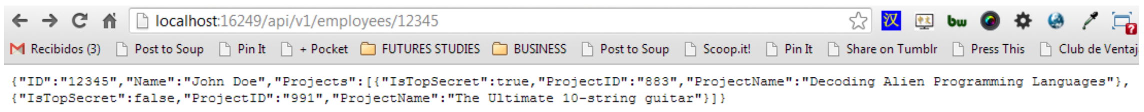
and

```
    [DataContract]
    public class Employee
    {
        [DataMember]
        public string ID { get; set; }

        [DataMember]
        public string Name { get; set; }

        [DataMember]
        public List<Project> Projects { get; set; }

    }
```

Obviously if we just modify the Get method of our interface this works without a hitch

```
public Employee Get ( string value )
{
    return new Employee ()
    {
        ID = value,
        Name = "John Doe",
        Projects = new List<Project> () {
            new Project ("Decoding Alien Programming Languages",
                         "883", true ),
            new Project ("The Ultimate 10-string guitar",
                         "991", false) }
    };
    }
```

We just get a json string that's a bit more complex



```
{"ID":"12345","Name":"John Doe","Projects":[{"IsTopSecret":true,"ProjectID":"883","ProjectName":"Decoding Alien Programming Languages"},
{"IsTopSecret":false,"ProjectID":"991","ProjectName":"The Ultimate 10-string guitar"}]}
```

But what we really want is to be able to read the projects of a given with an URL such as `employees/1234/projects`, meaning get me the projects for employee 1234, so let's change the service a bit to accommodate this.

4.1 Modify the service interface

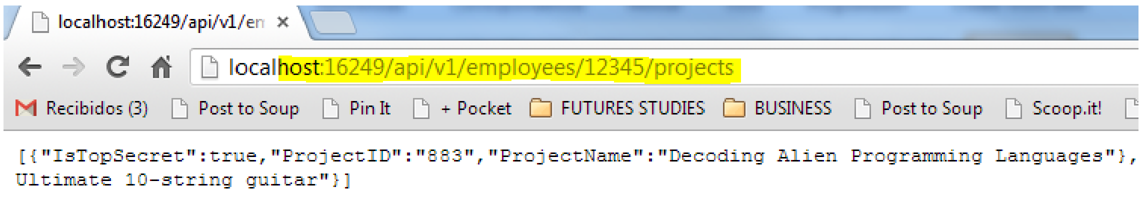
First we add a new method in the definition of our service interface, with the URL we want in the `WebGet` attribute.

```
[OperationContract]
[WebGet ( UriTemplate = "/Employees/{value}/Projects",
                        ResponseFormat = WebMessageFormat.Json )]
List<Project> GetEmployeeProjects ( string value );
```

And implement it accordingly

```
public List<Project> GetEmployeeProjects ( string value )
{
    return new List<Project> () {
            new Project ("Decoding Alien Programming Languages",
                         "883", true ),
            new Project ("The Ultimate 10-string guitar",
                         "991", false) };
}
```

Simple as that, now we can have the url that we wanted.



```
[{"IsTopSecret":true,"ProjectID":"883","ProjectName":"Decoding Alien Programming Languages"},
Ultimate 10-string guitar"}]
```