

# RESTful Day #4: Custom URL Re-Writing/Routing using Attribute Routes in MVC 4 Web APIs –by Akhil Mittal

---

## Table of Contents

<b>Table of Contents</b>	<b>1</b>
<b>Introduction</b>	<b>1</b>
<b>Roadmap</b>	<b>2</b>
<b>Routing</b>	<b>2</b>
<b>Existing Design and Problem</b>	<b>3</b>
<b>Attribute Routing</b>	<b>8</b>
<b>Setup REST endpoint / WebAPI project to define Routes</b>	<b>10</b>
<b>More Routing Constraints</b>	<b>15</b>
Range:	15
Regular Expression:	16
Optional Parameters and Default Parameters:	16
<b>RoutePrefix: Routing at Controller level</b>	<b>19</b>
<b>RoutePrefix: Versioning</b>	<b>20</b>
<b>RoutePrefix: Overriding</b>	<b>20</b>
<b>Disable Default Routes</b>	<b>21</b>
<b>Running the application</b>	<b>21</b>
<b>Conclusion</b>	<b>26</b>
<b>References</b>	<b>26</b>

## Introduction

We have already learnt a lot on WebAPI. I have already explained how to create WebAPI, connect it with database using Entity Framework, resolving dependencies using Unity Container as well as using MEF. In all our sample applications we were using default route that MVC provides us for CRUD operations. In this article I'll explain how to write your own custom routes using Attribute Routing. We'll deal with Action level routing as well as Controller level routing. I'll explain this in detail with the help of a sample application. My new readers can use any Web API sample they have, else you can also use the sample applications we developed in my previous articles.

## Roadmap

Let's revisit the road map that I started on Web API,



Here is my roadmap for learning RESTful APIs,

- RESTful Day #1: Enterprise level application architecture with Web API's using Entity Framework, Generic Repository pattern and Unit of Work.
- RESTful Day #2: Inversion of control using dependency injection in Web API's using Unity Container and Bootstrapper.
- RESTful Day #3: Resolve dependency of dependencies using Inversion of Control and dependency injection in Asp.net Web APIs with Unity Container and Managed Extensibility Framework (MEF).
- **RESTful Day #4: Custom URL Re-Writing/Routing using Attribute Routes in MVC 4 Web APIs.**
- RESTful Day #5: Basic Authentication and Token based custom Authorization in Web API's using Action Filters.
- RESTful Day #6: Request logging and Exception handling/logging in Web API's using Action Filters, Exception Filters and nLog.
- RESTful Day #7: Unit testing Asp.Net Web API's controllers using NUnit.
- RESTful Day #8: Extending OData support in Asp.Net Web API's.

I'll purposely use Visual Studio 2010 and .net Framework 4.0 because there are few implementations that are very hard to find in .Net Framework 4.0, but I'll make it easy by showing how we can do it.

## Routing

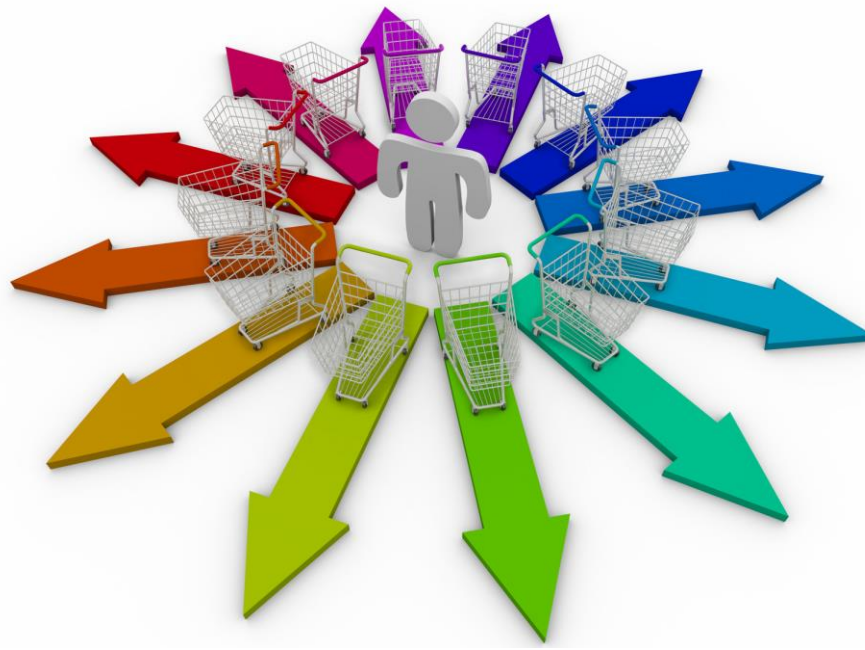
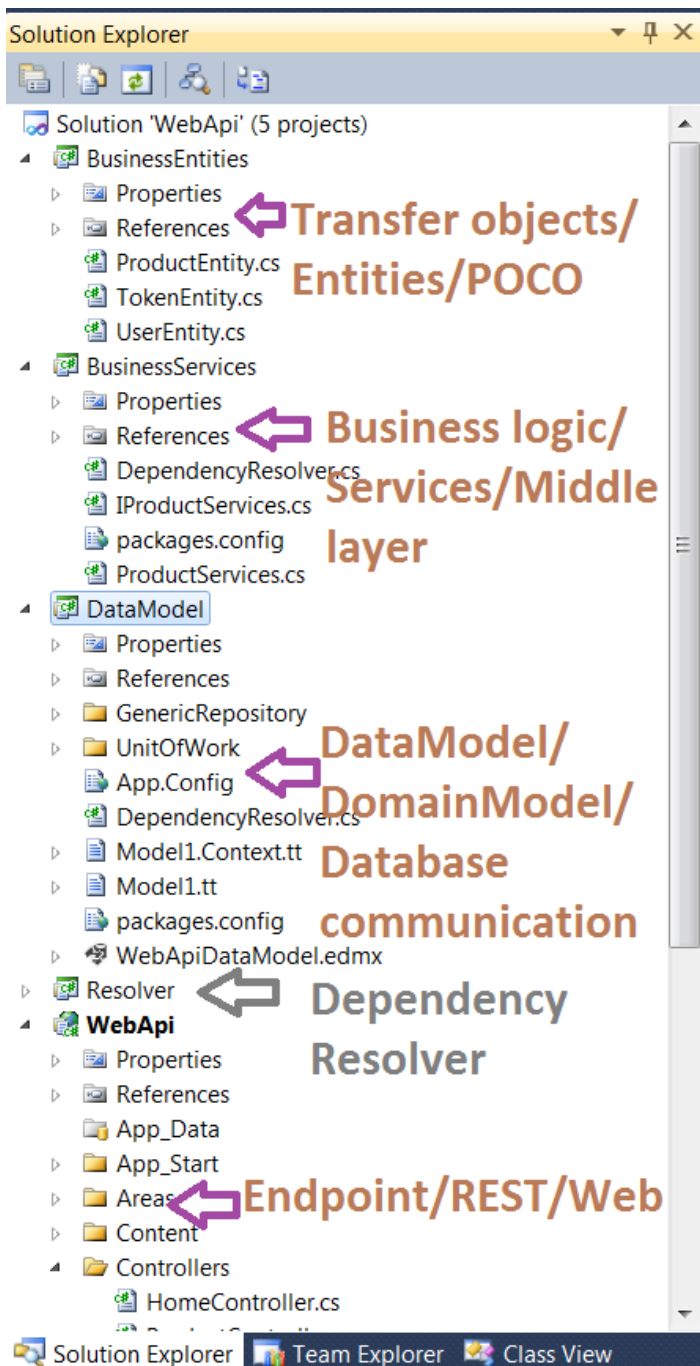


Image credit : <http://www.victig.com/wp-content/uploads/2010/12/routing.jpg>

Routing in generic terms for any service, api, website is a kind of pattern defining system that tries to maps all request from the clients and resolves that request by providing some response to that request. In WebAPI we can define routes in WebAPIConfig file, these routes are defined in an internal Route Table. We can define multiple sets of Routes in that table.

## Existing Design and Problem

We already have an existing design. If you open the solution, you'll get to see the structure as mentioned below,



In our existing application, we created WebAPI with default routes as mentioned in the file named WebApiConfig in App\_Start folder of WebApi project. The routes were mentioned in the Register method as,

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

Do not get confused by MVC routes, since we are using MVC project we also get MVC routes defined in RouteConfig.cs file as,

```

public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id =
UrlParameter.Optional }
    );
}

```

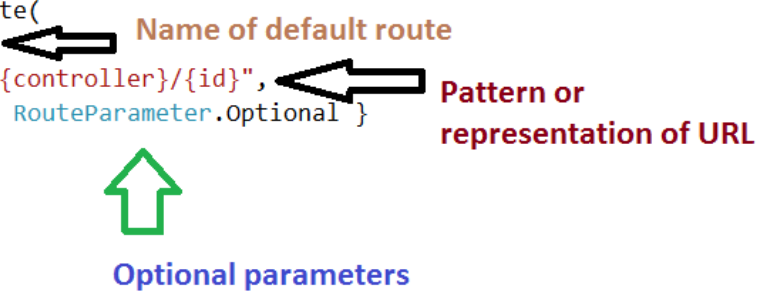
We need to focus on the first one i.e. WebAPI route. As you can see in the following image, what each property signifies,

```

using System.Linq;
using System.Web.Http;

namespace WebApi
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}

```



We have a route name, we have a common URL pattern for all routes, and option to provide optional parameters as well.

Since our application do not have particular different action names, and we were using HTTP VERBS as action names, we didn't bother much about routes. Our Action names were like ,

1. `public HttpResponseMessage Get()`
2. `public HttpResponseMessage Get(int id)`
3. `public int Post([FromBody] ProductEntity productEntity)`
4. `public bool Put(int id, [FromBody]ProductEntity productEntity)`
5. `public bool Delete(int id)`

Default route defined does not takes HTTP VERBS action names into consideration and treat them default actions, therefore it does not mentions {action} in routeTemplate. But that's not limitation, we can have our own routes defined in WebApiConfig , for example, check out the following routes,

```

public static void Register(HttpConfiguration config)
{
    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );

    config.Routes.MapHttpRoute(
        name: "ActionBased",
        routeTemplate: "api/{controller}/{action}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );

    config.Routes.MapHttpRoute(
        name: "ActionBased",
        routeTemplate: "api/{controller}/action/{action}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
}

public static void Register(HttpConfiguration config)
{
    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );

    config.Routes.MapHttpRoute(
        name: "ActionBased",
        routeTemplate: "api/{controller}/{action}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );

    config.Routes.MapHttpRoute(
        name: "ActionBased",
        routeTemplate: "api/{controller}/action/{action}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
}

```

And etc. In above mentioned routes, we can have action names as well, if we have custom actions. So there is no limit to define routes in WebAPI. But there are few limitations to this, note we are talking about WebAPI 1 that we use with .Net Framework 4.0 in Visual Studio 2010. Web API 2 has overcome those limitations by including the solution that I'll explain in this article. Let's check out the limitations of these routes,



**Every Controller will have to follow the same common route**



**The routes will be common to each and every action that we add to our controller**



**We can not have more than one endpoints to our service, and we'll be forced to stick to only one route**



**We can not have custom names of the routes. Routes depend on names of controller and action**



**We can not change the routes once they are defined, unless we agree to change the name of controllers and actions**



**Two or more action routes may conflict with each other, hence the priority will be given to the first route defined.**

Yes, these are the limitations that I am talking about in Web API 1.

If we have route template like `routeTemplate: "api/{controller}/{id}"` or `routeTemplate: "api/{controller}/{action}/{id}"` or `routeTemplate: "api/{controller}/action/{action}/{id}"`, We can never have custom routes and will have to stick to old route convention provided by MVC. Suppose your client of the project wants to expose multiple endpoints for the service, he can't do so. We also can not have our own defined names for the routes, so lots of limitation.

Let's suppose we want to have following kind of routes for my web api endpoints, where I can define versions too,

```
v1/Products/Product/allproducts
v1/Products/Product/productid/1
v1/Products/Product/particularproduct/4
v1/Products/Product/myproduct/<with a range>
v1/Products/Product/create
v1/Products/Product/update/3
```

and so on, then we cannot achieve this with existing model.

Fortunately these things have been taken care of in WebAPI 2 with MVC 5, but for this situation we have `AttributeRouting` to resolve and overcome these limitations.

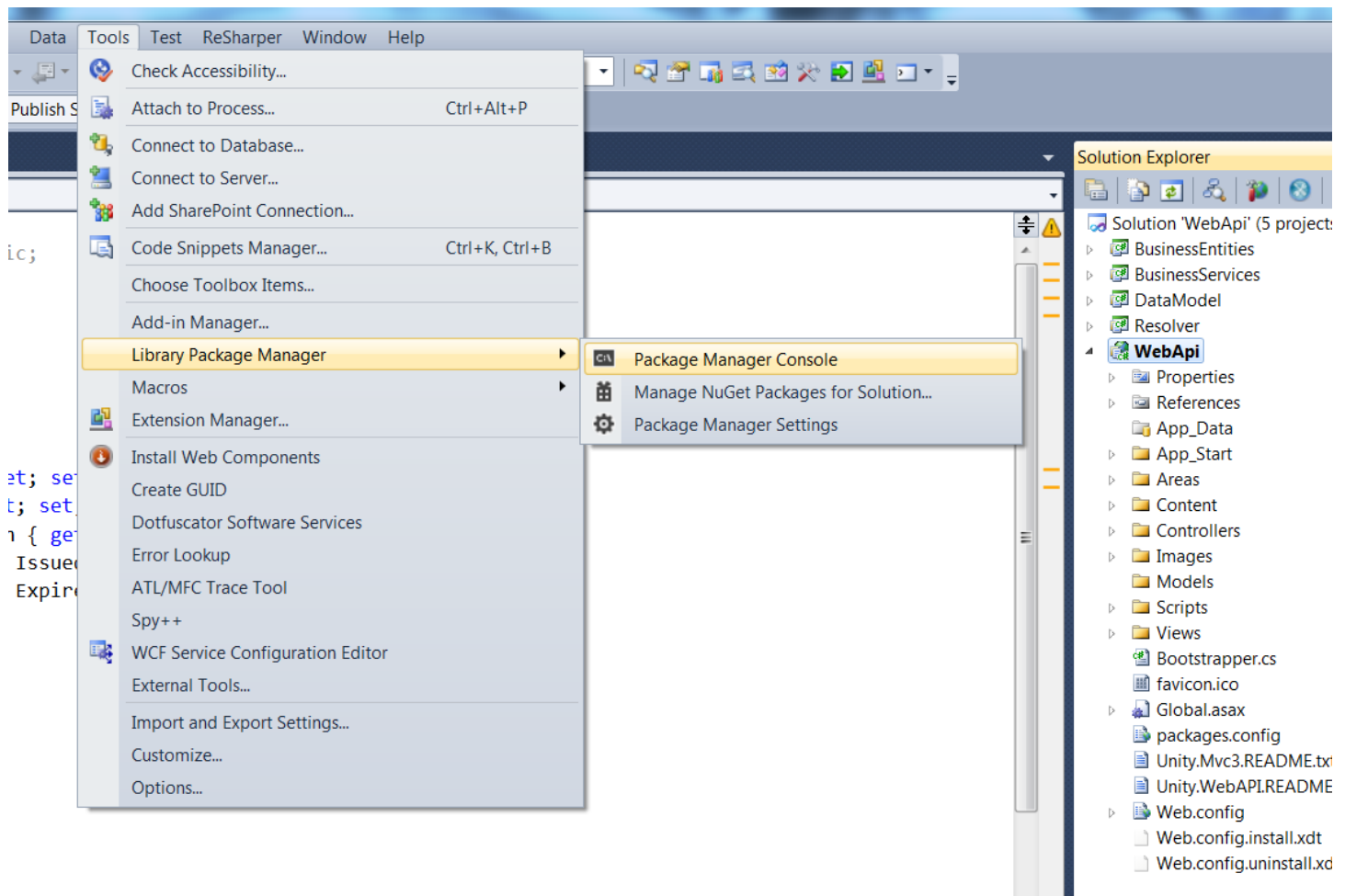


## Attribute Routing

Attribute Routing is all about creating custom routes at controller level, action level. We can have multiple routes using Attribute Routing. We can have versions of routes as well, in short we have the solution for our existing problems. Let's straight away jump on how we can implement this in our existing project. I am not explaining on how to create a WebAPI, for that you can refer my first [post](#) of the series.

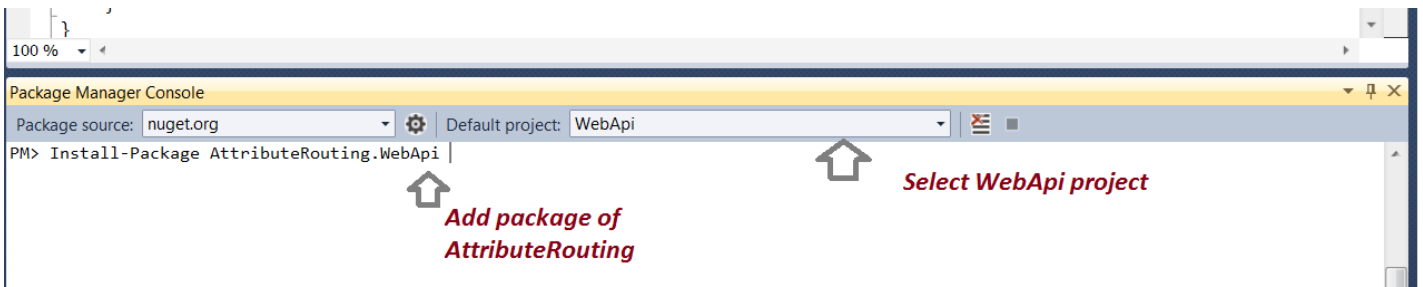
**Step 1:** Open the solution, and open Package Manager Console like shown below in the figure,

Goto Tools->Library Package Manager-> Package Manager Console

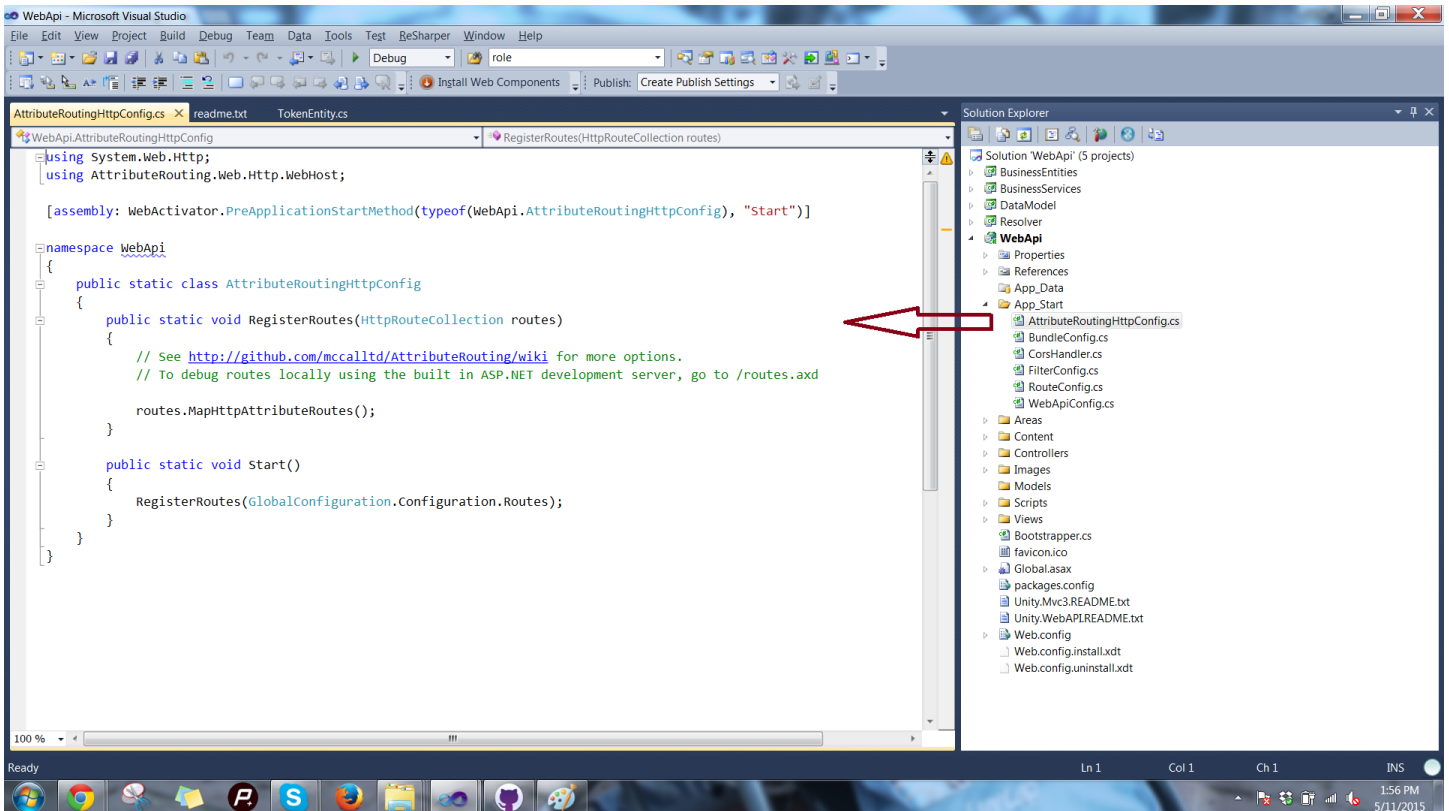


**Step 2:** In the package manager console window at left corner of Visual Studio type, Install-Package AttributeRouting.WebApi, and choose the project WebApi or your own API project if you are using any other code sample, then press enter.





**Step 3:** As soon as the package is installed, you'll get a class named `AttributeRoutingHttpConfig.cs` in your `App_Start` folder.



This class has its own method to `RegisterRoutes`, which internally maps attribute routes. It has a start method that picks Routes defined from `GlobalConfiguration` and calls the `RegisterRoutes` method,

```
using System.Web.Http;
using AttributeRouting.Web.Http.WebHost;

[assembly: WebActivator.PreApplicationStartMethod(typeof(WebApi.AttributeRoutingHttpConfig),
"Start")]

namespace WebApi
{
    public static class AttributeRoutingHttpConfig
    {
        public static void RegisterRoutes(HttpRouteCollection routes)
        {
            // See http://github.com/mccalltd/AttributeRouting/wiki for more options.
        }
    }
}
```

```

        // To debug routes locally using the built in ASP.NET development server,
        go to /routes.axd

        routes.MapHttpAttributeRoutes();
    }

    public static void Start()
    {
        RegisterRoutes(GlobalConfiguration.Configuration.Routes);
    }
}

```

We don't even have to touch this class, our custom routes will automatically be taken care of using this class. We just need to focus on defining routes. No coding 😊. You can now use route specific stuff like route names, verbs, constraints, optional parameters, default parameters, methods, route areas, area mappings, route prefixes, route conventions etc.

## Setup REST endpoint / WebAPI project to define Routes

Our 90% of the job is done.



We now need to setup or WebAPI project and define our routes. Our existing ProductController class looks something like shown below,

```

using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using BusinessEntities;
using BusinessServices;

namespace WebApi.Controllers
{
    public class ProductController : ApiController
    {
        private readonly IProductServices _productServices;
    }
}

```

```

#region Public Constructor

/// <summary>
/// Public constructor to initialize product service instance
/// </summary>
public ProductController(IProductServices productService)
{
    _productServices = productService;
}

#endregion

// GET api/product
public HttpResponseMessage Get()
{
    var products = _productServices.GetAllProducts();
    var productEntities = products as List<ProductEntity> ?? products.ToList();
    if (productEntities.Any())
        return Request.CreateResponse(HttpStatusCode.OK, productEntities);
    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products not
found");
}

// GET api/product/5
public HttpResponseMessage Get(int id)
{
    var product = _productServices.GetProductById(id);
    if (product != null)
        return Request.CreateResponse(HttpStatusCode.OK, product);
    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found for
this id");
}

// POST api/product
public int Post([FromBody] ProductEntity productEntity)
{
    return _productServices.CreateProduct(productEntity);
}

// PUT api/product/5
public bool Put(int id, [FromBody] ProductEntity productEntity)
{
    if (id > 0)
    {
        return _productServices.UpdateProduct(id, productEntity);
    }
    return false;
}

// DELETE api/product/5
public bool Delete(int id)
{
    if (id > 0)
        return _productServices.DeleteProduct(id);
    return false;
}

```

```
}  
}  
}
```

Where we have a controller name **Product** and Action names as Verbs. When we run the application, we get following type of endpoints only (please ignore port and localhost setting. It's because I run this application from my local environment),

## ASP.NET Web API Help Page

### Introduction

Provide a general description of your APIs here.

### Product

API	Description
<a href="#">GET api/Product</a>	Documentation for 'Get'.
<a href="#">GET api/Product/{id}</a>	Documentation for 'Get'.
<a href="#">POST api/Product</a>	Documentation for 'Post'.
<a href="#">PUT api/Product/{id}</a>	Documentation for 'Put'.
<a href="#">DELETE api/Product/{id}</a>	Documentation for 'Delete'.

Get All Products:

<http://localhost:40784/api/Product>

Get product By Id:

<http://localhost:40784/api/Product/3>

Create product :

<http://localhost:40784/api/Product> (with json body)

Update product:

<http://localhost:40784/api/Product/3> (with json body)

Delete product:

<http://localhost:40784/api/Product/3>

**Step 1:** Add two namespaces to your controller,

```
using AttributeRouting;
```

```
using AttributeRouting.Web.Http;
```

**Step 2:** Decorate your action with different routes,

```
using AttributeRouting;
using AttributeRouting.Web.Http;
using BusinessEntities;
using BusinessServices;

namespace WebApi.Controllers
{
    public class ProductController : ApiController
    {
        private readonly IProductServices _productServices;

        // GET api/product
        public HttpResponseMessage Get()
        {
            var products = _productServices.GetAllProducts();
            var productEntities = products as List<ProductEntity> ?? products.ToList();
            if (productEntities.Any())
            {
                return Request.CreateResponse(HttpStatusCode.OK, productEntities);
            }
            return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products not found");
        }

        // GET api/product/5
        [GET("productid/{id?}")]
        public HttpResponseMessage Get(int id)
        {
            var product = _productServices.GetProductById(id);
            if (product != null)
            {
                return Request.CreateResponse(HttpStatusCode.OK, product);
            }
            return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Product not found");
        }
    }
}
```


Like in above image, I defined a route with name productid which took id as a parameter. We also have to provide verb (GET, POST, PUT, DELETE, PATCH) along with the route like shown in image. So it is [GET("productid/{id?}")] . You can define whatever route you want for your Action like [GET("product/id/{id?}")] , [GET("myproduct/id/{id?}")] and many more. Now when I run the application and navigate to /help page, I get this,

# ASP.NET Web API Help Page

## Introduction

Provide a general description of your APIs here.

## Product

API	Description
<a href="#">GET productid</a>	Documentation for 'Get'.
<a href="#">GET productid/{id}</a>  <i>One new route for get by id</i>	Documentation for 'Get'.
<a href="#">GET api/Product</a>	Documentation for 'Get'.
<a href="#">GET api/Product/{id}</a>	Documentation for 'Get'.

i.e. I got one more route for Getting product by id. When you'll test this service you'll get your desired url something like : <http://localhost:55959/Product/productid/3>, that sounds like real REST 😊.

Similarly decorate your Action with multiple routes like show below,


```
// GET api/product/5
[GET("productid/{id?}")]
[GET("particularproduct/{id?}")]
[GET("myproduct/{id:range(1, 3)}")]
public HttpResponseMessage Get(int id)
{
    var product = _productServices.GetProductById(id);
    if (product != null)
        return Request.CreateResponse(HttpStatusCode.OK, product);
    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found for
this id");
}
```

## Introduction

Provide a general description of your APIs here.

## Product

API
<a href="#">GET myproduct</a>
<a href="#">GET myproduct/{id}</a>
<a href="#">GET particularproduct</a>
<a href="#">GET particularproduct/{id}</a>
<a href="#">GET productid</a>
<a href="#">GET productid/{id}</a>
<a href="#">GET api/Product</a>
<a href="#">GET api/Product/{id}</a>
<a href="#">POST api/Product</a>
<a href="#">PUT api/Product/{id}</a>



**Multiple routes for same API method**

Therefore we see, we can have our custom route names and as well as multiple endpoints for a single Action. That's exciting. Each endpoint will be different but will serve same set of result.

- `{id?}`: here '?' means that parameter can be optional.
- `[GET("myproduct/{id:range(1, 3)}")]`, signifies that the product id's falling in this range will only be shown.

## More Routing Constraints

You can leverage numerous Routing Constraints provided by Attribute Routing. I am taking example of some of them,

### Range:

To get the product within range, we can define the value, on condition that it should exist in database.

```
[GET("myproduct/{id:range(1, 3)}")]
```



```

public HttpResponseMessage Get(int id)
{
    var product = _productServices.GetProductById(id);
    if (product != null)
        return Request.CreateResponse(HttpStatusCode.OK, product);
    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found for
this id");
}

```

## Regular Expression:

You can use it well for text/string parameters more efficiently.

```

[GET(@"id/{e:regex(^[\0-9]$)}")]
public HttpResponseMessage Get(int id)
{
    var product = _productServices.GetProductById(id);
    if (product != null)
        return Request.CreateResponse(HttpStatusCode.OK, product);
    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found for
this id");
}

```

e.g. [GET(@"text/{e:regex(^[A-Z][a-z][\0-9]\$)}")]

## Optional Parameters and Default Parameters:

You can also mark the service parameters as optional in the route. For example you want to fetch an employee detail from the data base with his name,

```

[GET("employee/name/{firstname}/{lastname?}")]
public string GetEmployeeName(string firstname, string lastname="mittal")
{
    .....
    .....
}

```

In the above mentioned code, I marked last name as optional by using question mark '?' to fetch the employee detail.

It's my end user wish to provide the last name or not.

So the above endpoint could be accessed through GET verb with urls,

```

~/employee/name/akhil/mittal
~/employee/name/akhil

```

If a route parameter defined is marked optional, you must also provide a default value for that method parameter.

In the above example, I marked 'lastname' as an optional one and so provided a default value in the method parameter, if user doesn't send any value, "mittal" will be used.

In .Net 4.5 Visual Studio > 2010 with WebAPI 2, you can define DefaultRoute as an attribute too, just try it by your own. Use attribute [DefaultRoute] to define default route values.

You can try giving custom routes to all your controller actions.

I marked my actions as,

```
// GET api/product
[GET("allproducts")]
[GET("all")]
public HttpResponseMessage Get()
{
    var products = _productServices.GetAllProducts();
    var productEntities = products as List<ProductEntity> ?? products.ToList();
    if (productEntities.Any())
        return Request.CreateResponse(HttpStatusCode.OK, productEntities);
    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products not
found");
}

// GET api/product/5
[GET("productid/{id?}")]
[GET("particularproduct/{id?}")]
[GET("myproduct/{id:range(1, 3)}")]
public HttpResponseMessage Get(int id)
{
    var product = _productServices.GetProductById(id);
    if (product != null)
        return Request.CreateResponse(HttpStatusCode.OK, product);
    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found for
this id");
}

// POST api/product
[POST("Create")]
[POST("Register")]
public int Post([FromBody] ProductEntity productEntity)
{
    return _productServices.CreateProduct(productEntity);
}

// PUT api/product/5
[PUT("Update/productid/{id?}")]
[PUT("Modify/productid/{id?}")]
public bool Put(int id, [FromBody] ProductEntity productEntity)
{
    if (id > 0)
    {
        return _productServices.UpdateProduct(id, productEntity);
    }
    return false;
}

// DELETE api/product/5
[DELETE("remove/productid/{id?}")]
[DELETE("clear/productid/{id?}")]
[PUT("delete/productid/{id?}")]
```

```

public bool Delete(int id)
{
    if (id > 0)
        return _productServices.DeleteProduct(id);
    return false;
}

```

And therefore get the routes as,

**GET:**

## Product

API	Description
<a href="#">GET v1/Products/Product/all</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/all?id={id}</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/allproducts</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/allproducts?id={id}</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/myproduct/{id}</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/particularproduct</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/particularproduct/{id}</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/productid</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/productid/{id}</a>	Documentation for 'Get'.

**POST / PUT / DELETE:**

<a href="#">POST v1/Products/Product/Register</a>	Documentation for 'Post'.
<a href="#">POST v1/Products/Product/Create</a>	Documentation for 'Post'.
<a href="#">PUT v1/Products/Product/Update/productid/{id}</a>	Documentation for 'Put'.
<a href="#">PUT v1/Products/Product/Modify/productid/{id}</a>	Documentation for 'Put'.
<a href="#">DELETE v1/Products/Product/delete/productid/{id}</a>	Documentation for 'Delete'.
<a href="#">DELETE v1/Products/Product/remove/productid/{id}</a>	Documentation for 'Delete'.
<a href="#">DELETE v1/Products/Product/clear/productid/{id}</a>	Documentation for 'Delete'.
<a href="#">GET api/Product</a>	Documentation for 'Get'.
<a href="#">GET api/Product/{id}</a>	Documentation for 'Get'.
<a href="#">POST api/Product</a>	Documentation for 'Post'.
<a href="#">PUT api/Product/{id}</a>	Documentation for 'Put'.

Check for more constraints [here](#).

You must be seeing “**v1/Products**” in every route, that is due to RoutePrefix I have used at controller level. Let’s discuss RoutePrefix in detail.

## RoutePrefix: Routing at Controller level

We were marking our actions with particular route, but guess what , we can mark our controllers too with certain route names, we can achieve this by using RoutePrefix attribute of AttributeRouting. Our controller was named Product, and I want to append Products/Product before my every action, there fore without duplicating the code at each and every action, I can decorate my Controller class with this name like shown below,

```
[RoutePrefix("Products/Product")]
public class ProductController : ApiController
{
```

Now, since our controller is marked with this route, it will append the same to every action too. For e.g. route of following action,

```
[GET("allproducts")]
[GET("all")]
public HttpResponseMessage Get()
```

```

{
    var products = _productServices.GetAllProducts();
    var productEntities = products as List<ProductEntity> ?? products.ToList();
    if (productEntities.Any())
        return Request.CreateResponse(HttpStatusCode.OK, productEntities);
    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products not
found");
}

```

Now becomes,

```

~/Products/Product/allproducts
~/Products/Product/all

```

## RoutePrefix: Versioning

Route prefix can also be used for versioning of the endpoints, like in my code I provided “v1” as version in my RoutePrefix as shown below,

```

[RoutePrefix("v1/Products/Product")]
public class ProductController : ApiController
{

```

Therefore “v1” will be appended to every route / endpoint of the service. When we release next version, we can certainly maintain a change log separately and mark the endpoint as “v2” at controller level, that will append “v2” to all actions,

e.g.

```

~/v1/Products/Product/allproducts
~/v1/Products/Product/all

```

```

~/v2/Products/Product/allproducts
~/v2/Products/Product/all

```

## RoutePrefix: Overriding

This functionality is present in .Net 4.5 with Visual Studio > 2010 with WebAPI 2. You can test it there.

There could be situations that we do not want to use RoutePrefix for each and every action. AttributeRouting provides such flexibility too, that despite of RoutePrefix present at controller level, an individual action could have its own route too. It just need to override the default route like shown below,

RoutePrefix at Controller :

```

[RoutePrefix("v1/Products/Product")]
public class ProductController : ApiController
{

```

Independent Route of Action:

```

[Route("~/MyRoute/allproducts")]
public HttpResponseMessage Get()

```

```

{
    var products = _productServices.GetAllProducts();
    var productEntities = products as List<ProductEntity> ?? products.ToList();
    if (productEntities.Any())
        return Request.CreateResponse(HttpStatusCode.OK, productEntities);
    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products not
found");
}

```

## Disable Default Routes

You must be wondering that in the list of all the URL's on service help page, we are getting some different/other routes that we have not defined through attribute routing starting like ~/api/Product. These routes are the outcome of default route provided in `WebApiConfig` file, remember?? If you want to get rid of those unwanted routes, just go and comment everything written in Register method in `WebApiConfig.cs` file under `App_Start` folder,

```

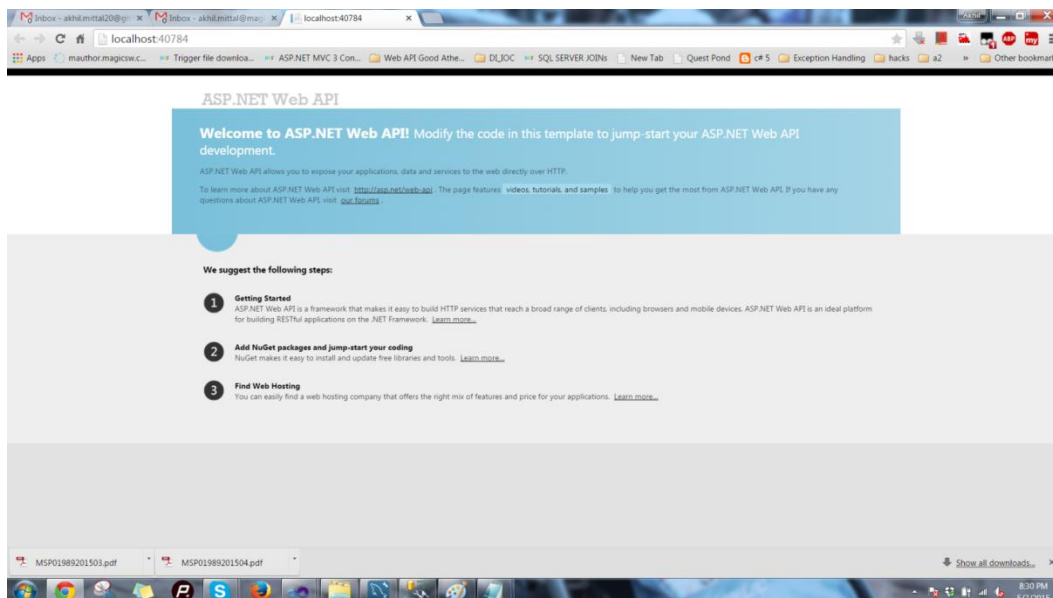
//config.Routes.MapHttpRoute(
//    name: "DefaultApi",
//    routeTemplate: "api/{controller}/{id}",
//    defaults: new { id = RouteParameter.Optional }
//);

```

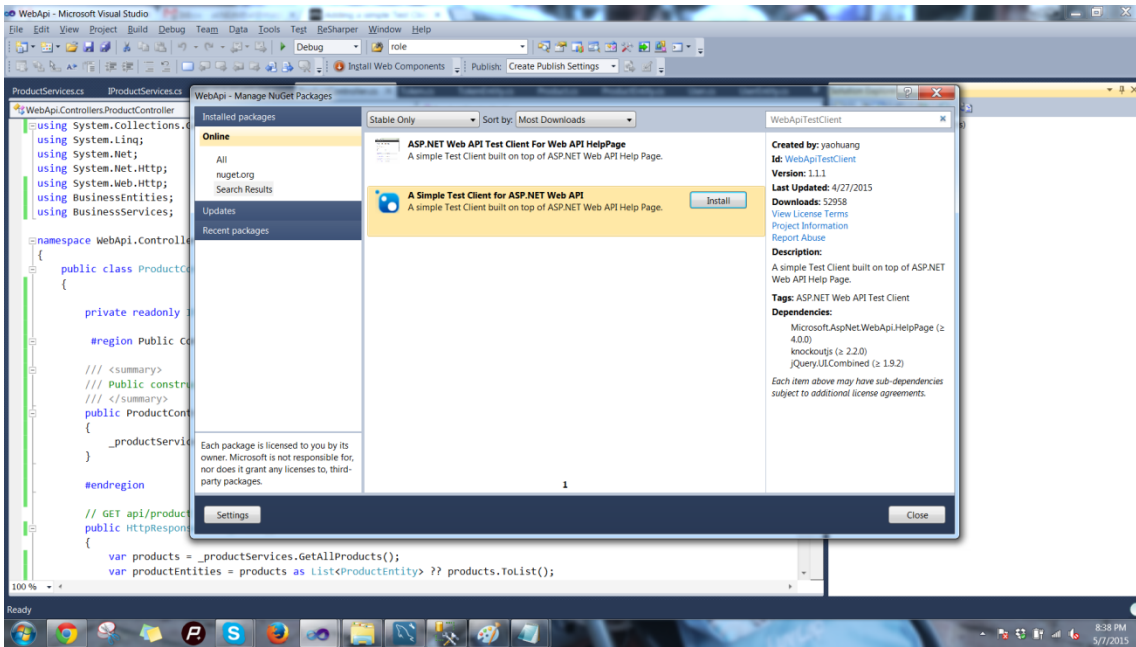
You can also remove the complete Register method, but for that you need to remove its calling too from `Global.asax` file.

## Running the application

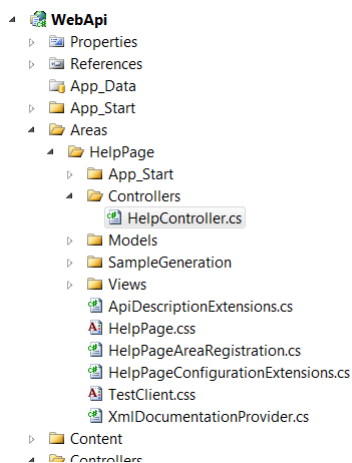
Just run the application, we get,



We already have our test client added, but for new readers, just go to Manage Nuget Packages, by right clicking WebAPI project and type WebAPITestClient in searchbox in online packages,



You'll get "A simple Test Client for ASP.NET Web API", just add it. You'll get a help controller in Areas->HelpPage like shown below,



I have already provided the database scripts and data in my previous article, you can use the same.

Append "/help" in the application url, and you'll get the test client,

**GET:**



# Product

API	Description
<a href="#">GET v1/Products/Product/all</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/all?id={id}</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/allproducts</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/allproducts?id={id}</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/myproduct/{id}</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/particularproduct</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/particularproduct/{id}</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/productid</a>	Documentation for 'Get'.
<a href="#">GET v1/Products/Product/productid/{id}</a>	Documentation for 'Get'.

## POST:

<a href="#">POST v1/Products/Product/Create</a>	Documentation for 'Post'.
<a href="#">POST v1/Products/Product/Register</a>	Documentation for 'Post'.

## PUT:

<a href="#">PUT v1/Products/Product/Update/productid/{id}</a>	Documentation for 'Put'.
<a href="#">PUT v1/Products/Product/Modify/productid/{id}</a>	Documentation for 'Put'.

## DELETE:

[DELETE v1/Products/Product/delete/productid/{id}](#)

[Documentation for 'Delete'.](#)

[DELETE v1/Products/Product/remove/productid/{id}](#)

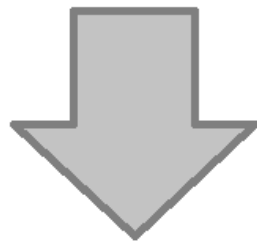
[Documentation for 'Delete'.](#)

[DELETE v1/Products/Product/clear/productid/{id}](#)

[Documentation for 'Delete'.](#)

You can test each service by clicking on it. Once you click on the service link, you'll be redirected to test the service page of that particular service. On that page there is a button Test API in the right bottom corner, just press that button to test your service,

**Press this button to  
test the API**



Test API

Service for Get All products,

[Help](#) [Page](#) [Home](#)

## GET v1/Products/Product/all

[Documentation for 'Get'.](#)

**GET v1/Products/Product/all** ×

**Headers** | [Add header](#)

☐ **Body**

**Response for GET v1/Products/Product/all** x

**Status**

200/OK

**Headers**

Server: ASP.NET Development Server/10.0.0.0  
Date: Sun, 31 May 2015 09:11:10 GMT  
X-AspNet-Version: 4.0.30319  
Cache-Control: no-cache  
Pragma: no-cache  
Expires: -1  
Content-Type: application/ison: charset=utf-8

**Body**

```
"ProductId": 1,  
  "ProductName": "Laptop"  
},  
{  
  "ProductId": 2,  
  "ProductName": "computer"  
},  
{  
  "ProductId": 4,  
  "ProductName": "IPhone"  
}  
}
```

Likewise you can test all the service endpoints.



Image source: <http://www.rsc.org/eic/sites/default/files/upload/cwtype-Endpoint.jpg> and [http://www.guybutlerphotography.com/EEGDemoConstruction/wp-content/uploads/2013/01/asthma-endpoint\\_shutterstock\\_89266522\\_small.jpg](http://www.guybutlerphotography.com/EEGDemoConstruction/wp-content/uploads/2013/01/asthma-endpoint_shutterstock_89266522_small.jpg)

## Conclusion

We now know how to define our custom endpoints and what its benefits are. Just to share that this library was introduced by Tim Call, author of <http://attributerouting.net> and Microsoft has included this in WebAPI 2 by default. My next article will explain token based authentication using ActionFilters in WepAPI. Till then Happy Coding 😊 you can also download the complete source code from [GitHub](#). Add the required packages, if they are missing in the source code.

Click [Github Repository](#) to browse for complete source code.

## References

<http://blogs.msdn.com/b/webdev/archive/2013/10/17/attribute-routing-in-asp-net-mvc-5.aspx>  
<https://github.com/mccalltd/AttributeRouting>

## About Author

Akhil Mittal works as a Sr. Analyst in [Magic Software](#) and have an experience of more than 8 years in C#.Net. He is a [codeproject](#) and a [c-sharpcorner](#) MVP (Most Valuable Professional). Akhil is a B.Tech (Bachelor of Technology) in Computer Science and holds a diploma in Information Security and Application Development from CDAC. His work experience includes Development of Enterprise Applications using C#, .Net and Sql Server, Analysis as well as Research and Development. His expertise is in web application development. He is a MCP (Microsoft Certified Professional) in Web Applications (MCTS-70-528, MCTS-70-515) and .Net Framework 2.0 (MCTS-70-536).