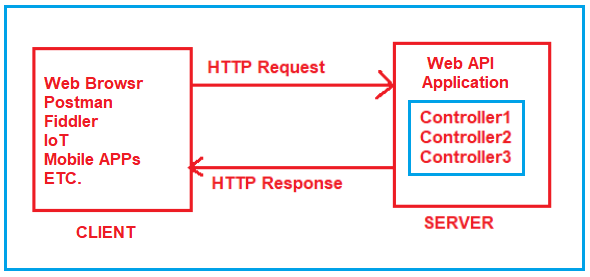
**Routing in ASP.NET Core Web API**

**Routing in ASP.NET Core Web API Application**

In this article, I am going to discuss **Routing in ASP.NET Core Web API** Application. Please read our previous article where we discussed [**Middleware Components in ASP.NET Core**](https://dotnettutorials.net/lesson/middleware-in-asp-net-core-web-api/) Application. Routing is one of the Core concepts in any type of ASP.NET Core Web Application.

**What is Routing in ASP.NET Core?**

Let us understand Routing with an example. Please have a look at the following image. On the right-hand side, we have the server and within the server, we have deployed our ASP.NET Core Web API application. And inside the ASP.NET Core Web API Application, assume that we have three controllers and each controller contains some action method. On the left-hand side, we have the client. The client can be a Web Browser, Postman, Fiddler, Swagger, IoTs, Mobile APP, etc. Suppose we are sending a request from the client to the browser. As we already discussed, first, the request has to go through the request processing pipeline and if everything is fine, then the ASP.NET Core Framework navigates that to the controller action method, and based on the processing of that action method, the respective client will get the response.



Here, the important that we need to understand is, how the application will come to know which request will be mapped to which controller action method. Basically, the mapping between the URL and resource is nothing but the concept or Routing.

**How does the Routing work in ASP.NET Core Web API?**

Routing in ASP.NET Core Web API application is the process of mapping the incoming HTTP Request (URL) to a particular resource i.e. controller action method.

For the Routing Concept in ASP.NET Core Web API, we generally set some URLs for each resource. When we run the application, then it will create the Route table and the Route table will contain the mapping information between the URL and the Resource. So, when we are sending a request from the client to the server, then the application will check the URL in the Route table and if it found an exact, then the application will forward the request to that particular resource else it will throw an error saying resource not found.

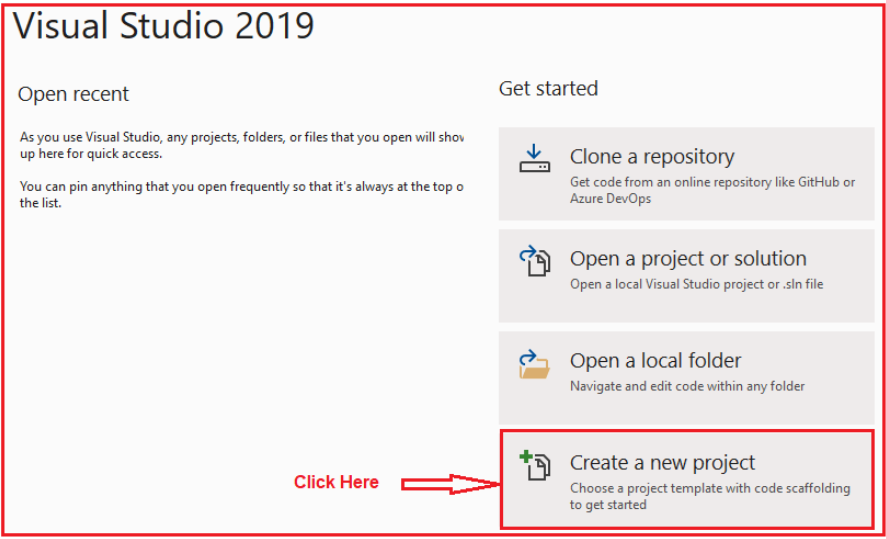
We can access any resource using a unique URL in ASP.NET Core Web API Application. It is also possible that a resource can have multiple unique URLs. But multiple resources can not have the same URL and if you do so, then the application gets confused to invoke which action method and as a result, you will get an ambiguity error.

So, the ASP.NET Core Framework maps the incoming HTTP Requests i.e. URLs to the action methods of Controllers based on the routes that are configured for your application. In ASP.NET Core, it is also possible to configure multiple routes, and also it is possible to set some specific configurations such as default values, constraints, message handlers, etc for each route. If this is not clear at the moment then don’t worry, we will discuss each and everything with examples.

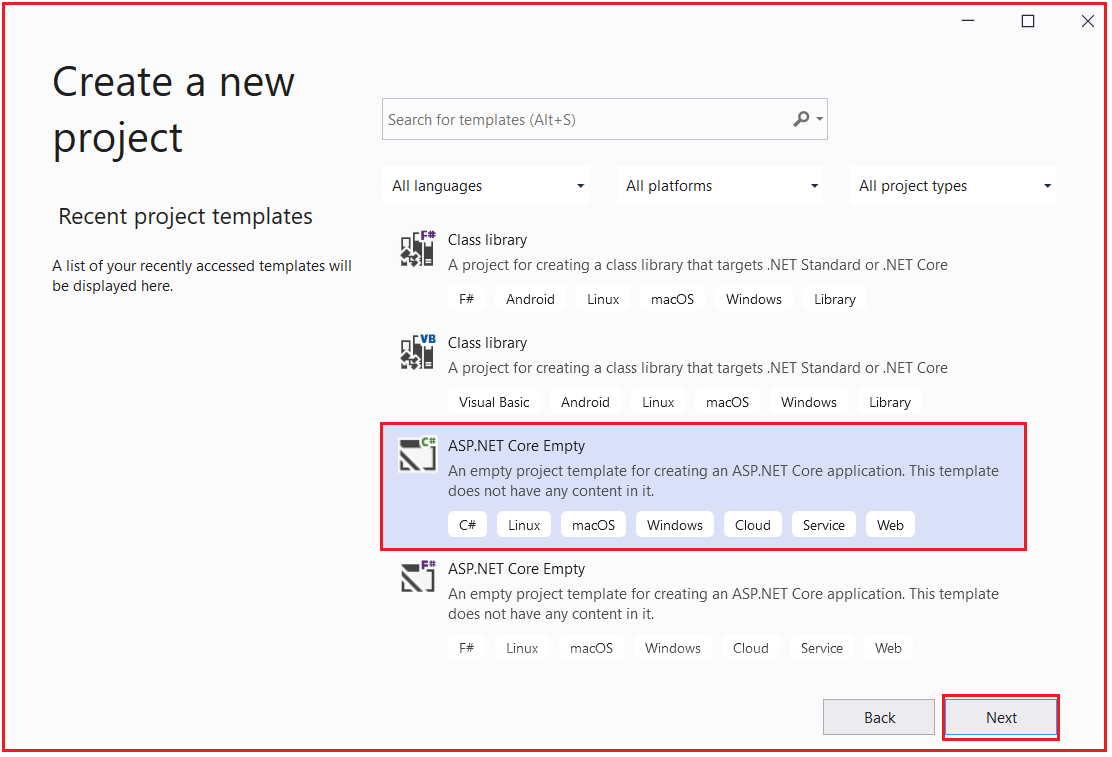
**How to enable Routing in ASP.NET Core Web API?**

Let us understand How to enable Routing in ASP.NET Core Web API Application with an example. Here, we are going to explain everything from scratch. So, create an empty ASP.NET Core Application by following the below steps.

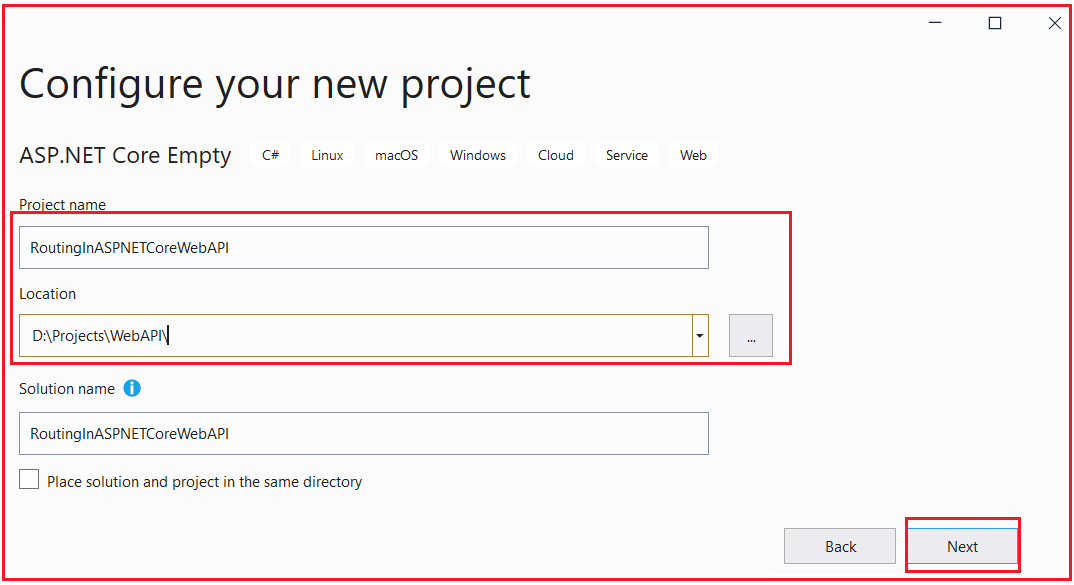
First, open Visual Studio 2019 and then click on the Create a new project option as shown in the below image.



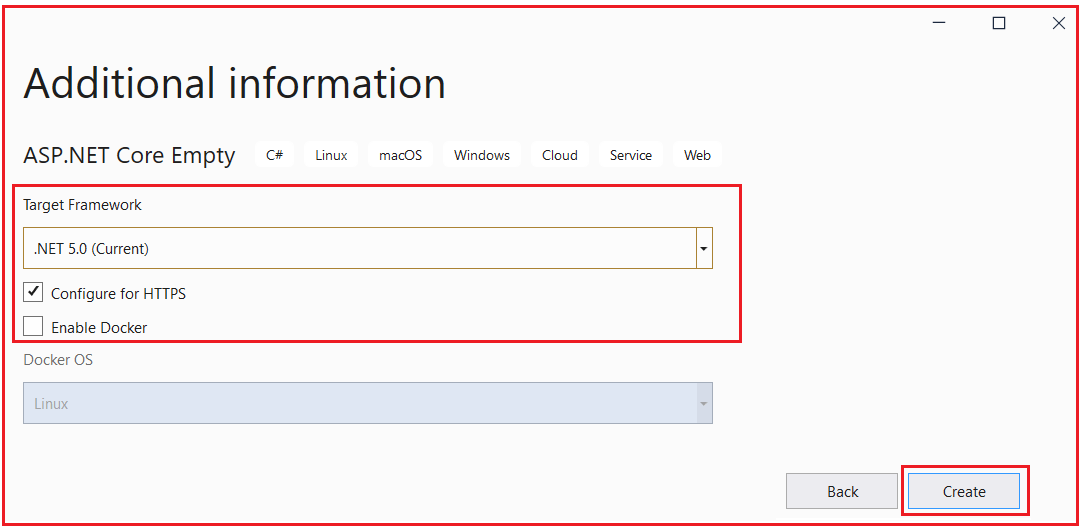
Once you click on the Create a new project option, it will open the Create a new project window. From this window select the ASP.NET Core Empty Project template which uses the programming language as C# and finally, click on the Next button as shown in the below image.



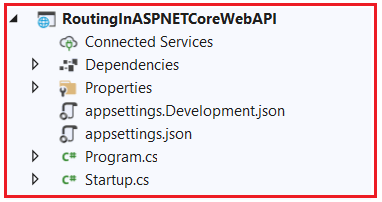
Once you click on the Next button, it will open configure your new project window. Here, you need to specify the Project name (RoutingInASPNETCoreWebAPI) and the location where you want to create the project. And then click on the Next button as shown in the below image.



Once you click on the Next button, it will open the Additional Information window. Here, I am going with the default configuration, and please make sure to select the Target Framework as .NET 5.0 and click on the Create button as shown in the below image.



Once you click on the Create button, it will create the ASP.NET Core Empty Project with the following file and folder structure.



**Adding ASP.NET Core Web API Service:**

Now, we have created an Empty ASP.NET Core Application. Let us add the ASP.NET Core Web API services so that our application will support the ASP.NET Core Web API Features. To do so we need to inject the ASP.NET Core Web API Service to the built-in IoC Container as **services.AddControllers();** You can do the same using the ConfigureService method of the Startup class as shown in the below code.

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddControllers**()**;

**}**

**Enabling Routing in ASP.NET Core Web API Application:**

In ASP.NET Core Web API Application, we can enable the Routing through Middleware. In order to enable Routing in ASP.NET Core, we need to add the following two middleware components to the HTTP Request processing Pipeline.

1. **UseRouting():** The UseRouting Middleware only enables the Routing for your application. This will not map any URL to any resource.
2. **UseEndpoints():** This middleware will map the URL to the resource. But the most important point that you need to remember is, the action methods are not only the resource that you can map. You can also map static file resources to a URL. But here we are only focusing on ASP.NET Core Web API and hence we are going to map the URL to action methods.

**Configuring the Routing Middlewares in ASP.NET Core:**

As we already discussed, if we want to configure any middleware then we need to configure the same inside the Configure method of the Startup class. So, let us modify the Configure method of the Startup class as shown below to configure the UseRouting and UseEndpoints Middlewares which will enable Routing as well as the mapping between the URL and Resource. As you can see in the below code within the UseEndpoints Middleware we write MapControllers which will actually set the mapping between URL and Controller action method. So, the Map controller will act as a bridge between the HTTP Request and the Controllers.

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapControllers**()**;

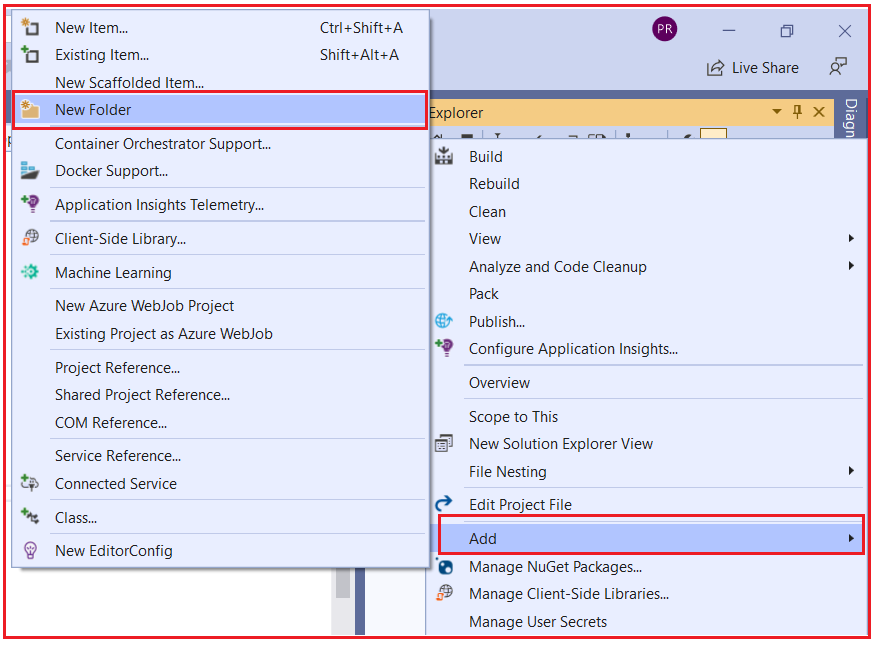
**})**;

**}**

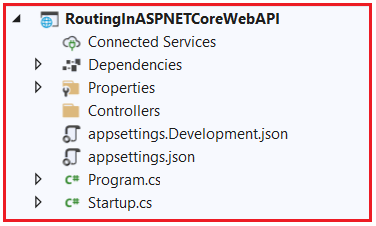
In ASP.NET Core Application, we have two ways to define the Routing i.e. Conventional Based and Attribute-Based Routing. And Attribute Routing is the most preferred way of defining routes in ASP.NET Core Web API Application and hence, we are only going to focus on Attribute Routing in this course.

**Adding Controller:**

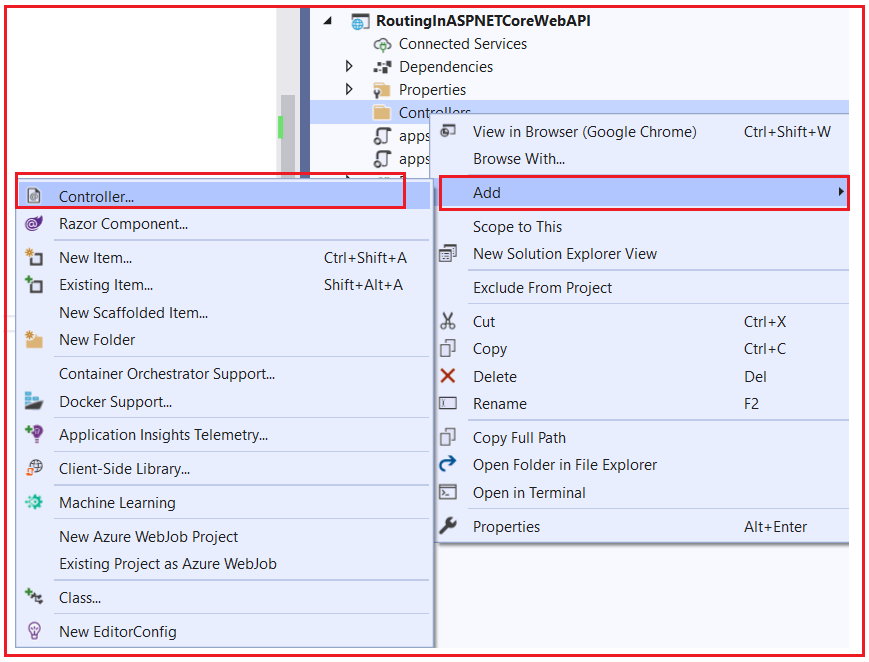
As of now, we have added the ASP.NET Core Web API Service and set the Routing for our application. Now let us add a controller i.e. HomeController to our application. The Controllers in the ASP.NET Core Application should be added inside the Controllers folder in your project. So, let us first add the Controllers folder to the project root directory. To do so, right-click on your project and then select **Add => New Folder** option from the context menu as shown in the below image.



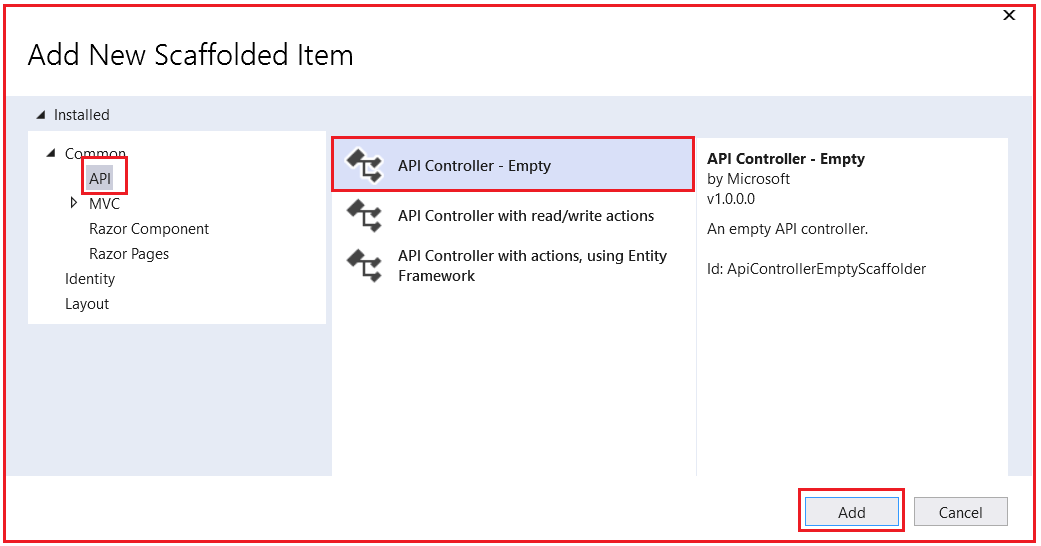
Then rename the folder name as Controllers. Once the controller folder is added your project structure should looks as shown in the below image.



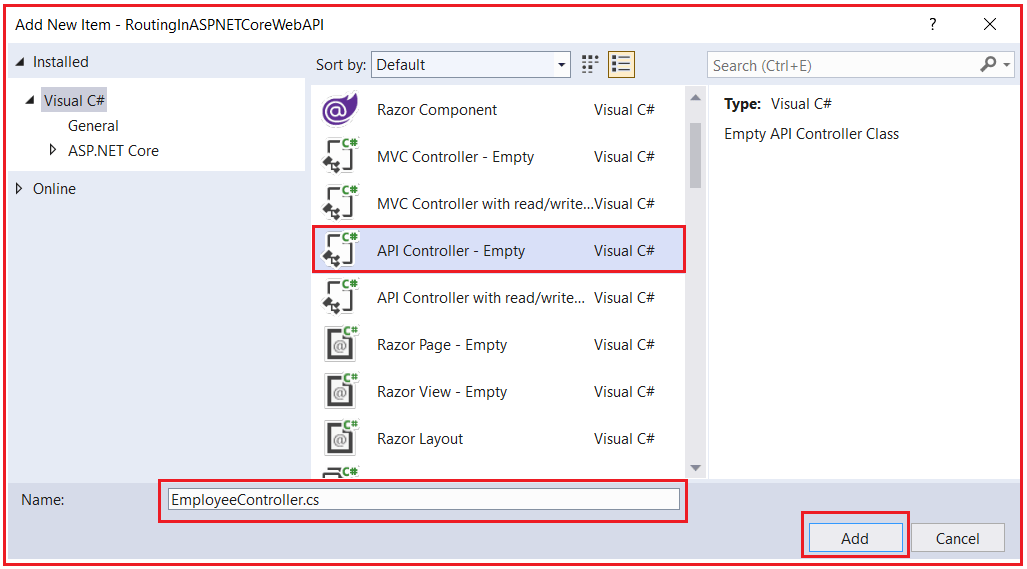
Let’s add the Home controller within the Controllers folder. To do so, right-click on the Controllers folder and then select Add => Controller from the context menu as shown in the below image.



It will open the following Add New Scaffolded item window. Here, from the left side, select API, and from the middle pane select API Controller – Empty and click on the Add button as shown in the below image. As we are going to do everything from scratch, so here we are selecting the Empty Controller template. In our upcoming articles, we are also going to discuss the other two templates.



From the next window, provide the controller name as EmployeeController and click on the Add button as shown in the below image.



Once you click on the Add button, it will add a class file with the name EmployeeController.cs within the Controllers folder. Now, you can find the following default code inside the EmployeeController.cs class file.

**using** *Microsoft.AspNetCore.Http;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**using** *System.Threading.Tasks;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**Route**(**"api/[controller]"**)]**

**[**ApiController**]**

**public** **class** EmployeeController : ControllerBase

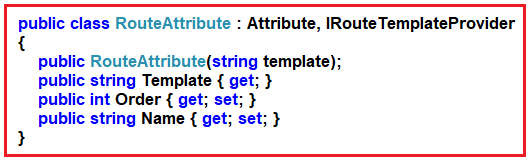
**{**

**}**

**}**

**Understanding Route Attribute in ASP.NET Core:**

Using the Route Attribute, we can define the Routes in ASP.NET Core Application either at the Controller level or at the action method level. Before using the Route Attribute lets us have a look at the signature of the Route Attribute. The following image shows the definition of the Route Attribute.



As you can see in the above image, Route Attribute is basically a class inherited from the Attribute class and IRouteTemplateProvider interface. The Constructor of the RouteAttribute class takes the template as an input parameter which is nothing but the URL that you are excepting from the client and it cannot be null.

**Adding Attribute Routing in ASP.NET Core Web Application:**

Now let us add two action methods within the EmployeeController class. Now, don’t concentrate on the return type and the data that we are returning from the action method, rather concentrate on the Routing concept. We want to invoke the GetAllEmployees method with the URL **/Emp/All** and GetEmployeeById method with the URL **/Emp/ById**. To achieve this, we can use the Route Attribute and decorate the action GetAllEmployees and GetEmployeeById method as [Route(“Emp/All”)] and [Route(“Emp/ById”)] respectively. So, modify the EmployeeController class as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"Emp/All"**)]**

**public** string GetAllEmployees**()**

**{**

**return** "Response from GetAllEmployees Method";

**}**

**[**Route**(**"Emp/ById"**)]**

**public** string GetEmployeeById**()**

**{**

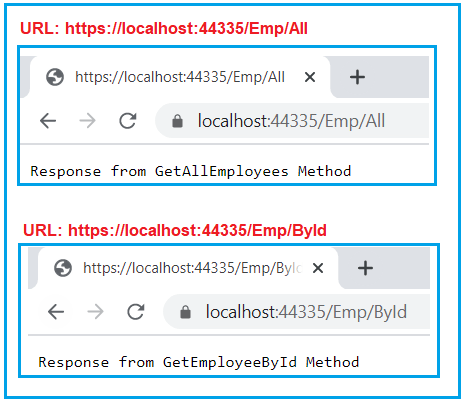
**return** "Response from GetEmployeeById Method";

**}**

**}**

**}**

Now everything is ready. So, run the application and access the action method using the URL we configured as shown in the below image.



**Variables and Query Strings in Routing**

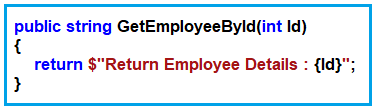
**Variables and Query Strings in ASP.NET Core Web API Routing**

In this article, I am going to discuss **how to work with variables and Query Strings in ASP.NET Core Web API Routing** with Examples. Please read our previous article where we discussed [**what is Routing**,](https://dotnettutorials.net/lesson/routing-in-asp-net-core-web-api/) how does routing works, and how to configure and use Routing in ASP.NET Core Web API Application. In fact, we are also going to work with the same example that we created in our previous article.

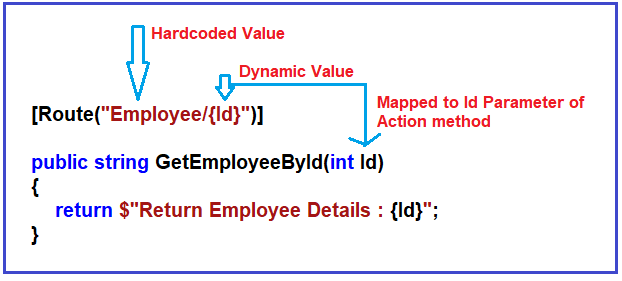
**Working with Variables in ASP.NET Core Web API Routing:**

When working with Real-time Restful services, then we need to deal with dynamic values like getting Order Details by Id, Get Employee details by Id, Get books by Author, etc. Let us see how we can handle the variables in the Routes in ASP.NET Core Web API Application.

For example, we have multiple employees and we want to fetch one employee detail by its Id. Then how we can get the Id values? The URL is the only place that is going to give us the Id value. So, we need to define one parameter to take the Id value within the method signature as shown in the below image

 .

In ASP.NET Core Web Application, if you want to pass anything as a variable then you need to use curly braces {} and inside the curly braces, you need to give the name of the parameter your method accepting. In our example, the GetEmployeeById method takes the Id parameter, so we need to pass the Id within the curly braces of the Route attribute as shown in the below image.



So, modify the EmployeeController class as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"Employee/{Id}"**)]**

**public** string GetEmployeeById**(int** Id**)**

**{**

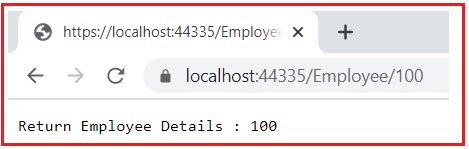
**return** $"Return Employee Details : {Id}";

**}**

**}**

**}**

Now you can access the GetEmployeeById action method using the URL: **Domainname/Employee/100** (instead of 100 you can pass any dynamic value i.e. the Employee Id which information you want to retrieve). So run the above application and see the output as shown in the below image.



This is how we pass dynamic values using routing.

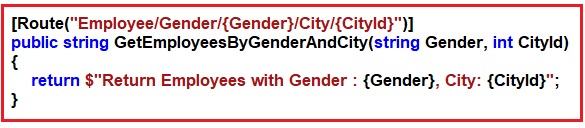
**Passing Multiple dynamic Values in ASP.NET Core Web API Routing:**

Now let us understand how to pass multiple dynamic values using the Route attribute. Now we need to fetch all the employees by Gender and City. Here, we want the Gender to be Male or Female i.e. as a string value and intentionally, we want to City Id value i.e. an integer value. So, we will create one action method which takes two parameters, one of which is Gender, and the order one is CityId as shown in the below image.



Now we want to access the above GetEmployeesByGenderAndCity method using the URL: **Employee/Gender/Male/City/10**

Here, Male and 10 are the dynamic values. So, we need to decorate the GetEmployeesByGenderAndCity method with the Route Attribute as shown in the below image. Here, we are passing the Gender and CityId parameter in curly braces.



So, modify the EmployeeController class as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"Employee/{Id}"**)]**

**public** string GetEmployeeById**(int** Id**)**

**{**

**return** $"Return Employee Details : {Id}";

**}**

**[**Route**(**"Employee/Gender/{Gender}/City/{CityId}"**)]**

**public** string GetEmployeesByGenderAndCity**(**string Gender, **int** CityId**)**

**{**

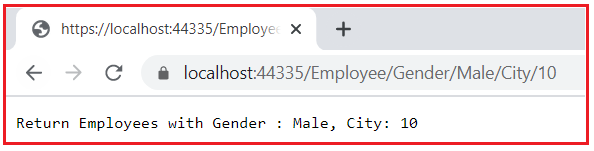
**return** $"Return Employees with Gender : {Gender}, City: {CityId}";

**}**

**}**

**}**

Now run the application and navigate to the URL Employee/Gender/Male/City/10 and you should get the message as expected as shown in the below image.

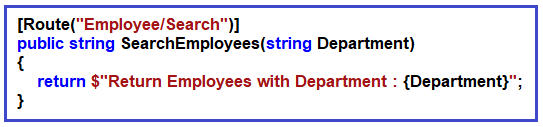


This is how you can pass multiple dynamic values in your routing.

**Working with Query Strings in ASP.NET Core Web API Routing:**

Let us understand how to work with Query Strings in ASP.NET Core Web API Routing. Query Strings are nothing but key-value pairs that you need to pass as of the URL. Again, you pass multiple query strings (i.e. multiple key-value pairs) separated by &. Further, the most important point that you need to remember is before the first query string or after the domain name you need to use a question mark (?). The question mark (?) in the URL indicates that the query string is started.

Let us understand the query string with an example. Now, we need to search employees by department name. But we don’t want the department name to be part of the Route Attribute. Then how we can do this? We can do this by using the query string. So, let us create a method with the name SearchEmployee with one parameter called Department as shown in the below image. Further notice, we have not included that parameter as part of the Route Attribute.



The complete code of EmployeeController is given below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"Employee/Search"**)]**

**public** string SearchEmployees**(**string Department**)**

**{**

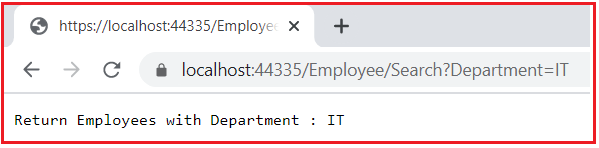
**return** $"Return Employees with Department : {Department}";

**}**

**}**

**}**

The URL to access the above action method is domainname/Employee/Search, but how we can pass the value for the Department parameter. The answer is the query string. So, what we need to do is just append **?Department=IT** to the end of the URL and press enter as shown in the below image.



**How to pass Multiple Query Strings in ASP.NET Core Web API?**

Let us understand how to pass multiple query strings with an example. In a real-time application, when you implement a search functionality, you generally accept multiple search parameters to filter out the data, Let say, we want employee city, gender, and department to filter out the number of employees to be returned. Then, in that case, our action method accepting three parameters. So, modify the SearchEmployees method of the Employee Controller class as shown below.

**[**Route**(**"Employee/Search"**)]**

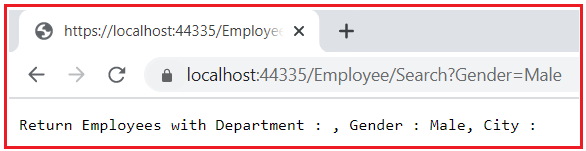
**public** string SearchEmployees**(**string Department, string Gender, string City**)**

**{**

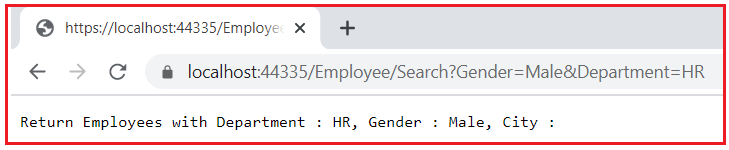
**return** $"Return Employees with Department : {Department}, Gender : {Gender}, City : {City}";

**}**

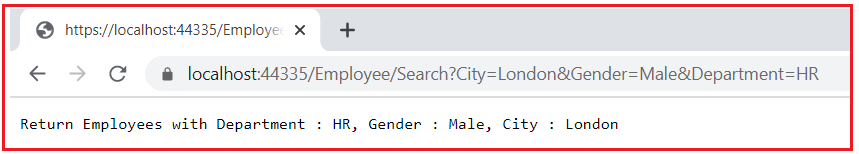
With the above changes in place, now run the application. Now let say we want to filter the employees by Gender only. Then you can only pass the Gender query string in the URL as shown in the below image.



Now, you want to search the employees by Gender and Department. Then you can pass two query strings in the URL as shown in the below image.



Now, you want to search the employees by using all three parameters i.e. Gender, Department, and City. Then you can pass the query strings in the URL as shown in the below image. The point that you need to remember the order of the query does not matter i.e. you can pass the query strings in any order.



This is how query strings work in ASP.NET Core Web API Routing. In the next article, I am going to discuss [**how to set multiple URLs for a single resource in ASP.NET Core Web API**](https://dotnettutorials.net/lesson/multiple-urls-for-a-single-resource-using-routing/) Application using Routing. Here, in this article, I try to explain how to work with **Variables and Query Strings in ASP.NET Core Web API Routing** with Examples. I hope you enjoy variables and Query Strings in the ASP.NET Core Web API Routing article.

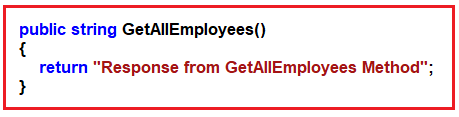
**Multiple URLs for a Single Resource using Routing**

**How to set up Multiple URLs for a Single Resource in ASP.NET Core Web API**

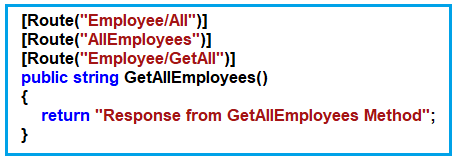
In this article, I am going to discuss **How to set up Multiple URLs for a Single Resource in ASP.NET Core Web API** Application with Examples. Please read our previous article, where we discussed how to work with [**Variables and Query Strings in ASP.NET Core Web API**](https://dotnettutorials.net/lesson/variables-and-query-strings-in-routing/). We are going to work with the same application that we created in our [**Routing in ASP.NET Core Web API**](https://dotnettutorials.net/lesson/routing-in-asp-net-core-web-api/) article.

**How to access a Single Resource with Multiple URLs in ASP.NET Core Web API:**

Let us understand how to access a single resource with Multiple URLs with an example. Suppose, we have the following resource available in our Employee Controller.



Now we want to access the above resource with three different URLs such as Employee/All, AllEmployees, Employee/GetAll, then how we can do this? If this is our requirement, then we need to decorate the GetAllEmployees action method with three different Route attributes as shown in the below image.



With this, now you can access the GetAllEmployees resource with three different URLs in the ASP.NET Core Web API Application. Let us prove this. First, modify the EmployeeController class as shown in the below code.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"Employee/All"**)]**

**[**Route**(**"AllEmployees"**)]**

**[**Route**(**"Employee/GetAll"**)]**

**public** string GetAllEmployees**()**

**{**

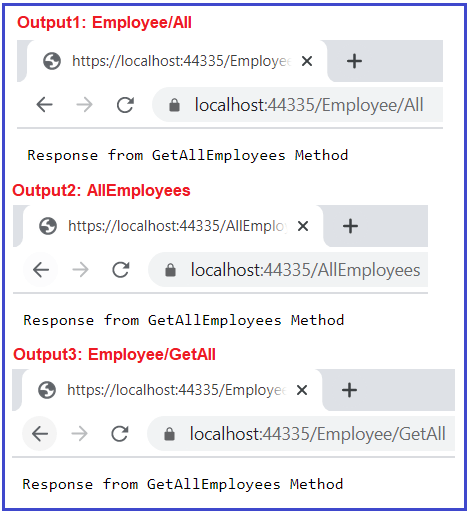
**return** "Response from GetAllEmployees Method";

**}**

**}**

**}**

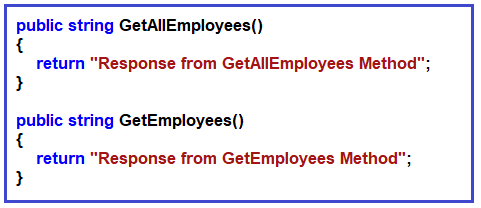
Now run the application and access the above resource using the different URLs as shown in the below image.



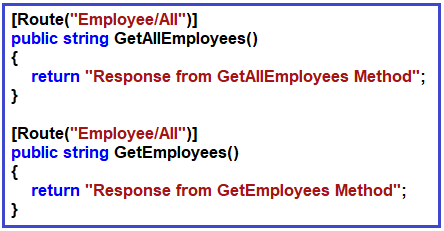
As you can see in the above image, we are now able to access the same resource using three different URLs in ASP.NET Core Web API Application. As you can see each URL is unique. So, as long as the URLs are unique, you can access a particular resource using different URLs.

**What happens if we use the same URL for multiple resources?**

This is not accepted in ASP.NET Core Web API. Let us prove this with an example. Let say we have the following two resources.



Now, let us try to access both the above resources using the URL Employee/All, so what we will do is, we will decorate both the resources with the same Route Attribute as shown in the below image.



Let us modify the EmployeeController class as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"Employee/All"**)]**

**public** string GetAllEmployees**()**

**{**

**return** "Response from GetAllEmployees Method";

**}**

**[**Route**(**"Employee/All"**)]**

**public** string GetEmployees**()**

**{**

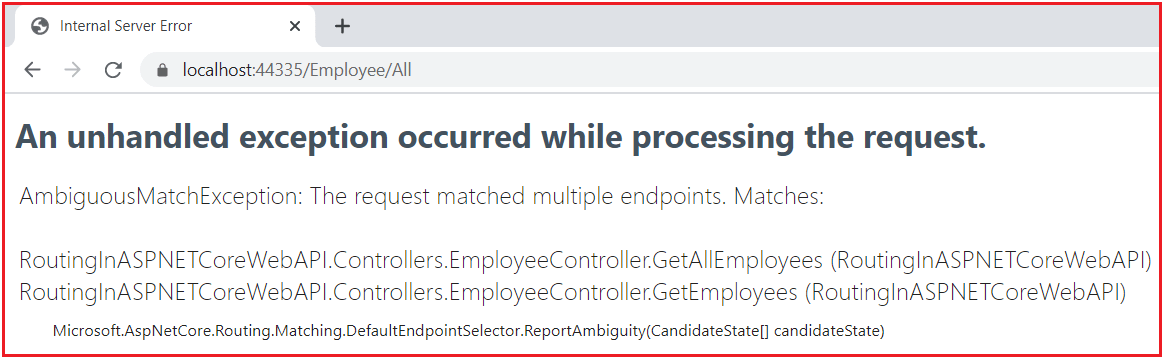
**return** "Response from GetEmployees Method";

**}**

**}**

**}**

With the above changes in place, now run the application and navigate to **Employee/All** URL as shown in the below image.



As you can see in the above image, it is throwing an Internal Server Error saying AmbiguousMatchException: The request matched multiple endpoints. This is because the application finds to resource for the same URL and gets confused about who is going to handle the request and hence throwing the AmbiguousMatchException.

**Note:** So, the point that you need to remember is, each resource must have a unique URL, and also it is possible that a resource can be accessed using multiple URLs as long as all the URLs are unique. But it is not possible to access two or more different resources using a single URL in ASP.NET Core Web API Application.

Now change the Route Attribute of both the resources as shown in the below code to give different URLs.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"Employee/GetAll"**)]**

**[**Route**(**"EmployeeAll"**)]**

**public** string GetAllEmployees**()**

**{**

**return** "Response from GetAllEmployees Method";

**}**

**[**Route**(**"Employee/All"**)]**

**public** string GetEmployees**()**

**{**

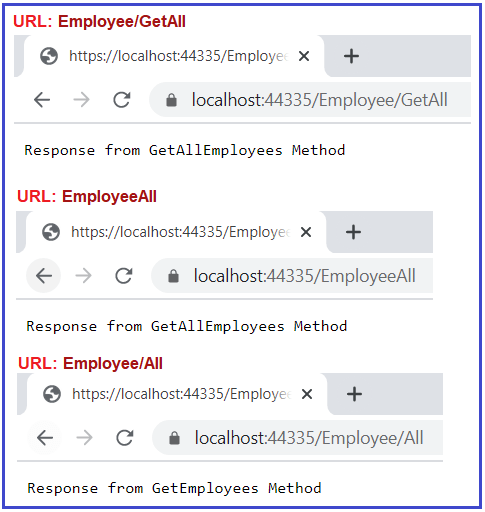
**return** "Response from GetEmployees Method";

**}**

**}**

**}**

With the above changes in place, now you can access the **GetAllEmployees** resource using two URLs i.e. **Employee/GetAll** and **EmployeeAll**. On the other hand, you can access the **GetEmployees** resource using the URL **Employee/All** as shown in the below image.



In the next article, I am going to discuss [**Token Replacement in ASP.NET Core Web API Attribute Routing**](https://dotnettutorials.net/lesson/token-replacement-asp-net-core-routing/)with Examples. Here, in this article, I try to explain **How to set up Multiple URLs for a Single Resource in ASP.NET Core Web API** Application with Examples. I hope you enjoy Multiple URLs for a Single Resource article.

**Token Replacement in ASP.NET Core Routing**

**Token Replacement in ASP.NET Core Web API Routing**

In this article, I am going to discuss **Token Replacement in ASP.NET Core Web API Routing** with Examples. Please read our previous article, where we discussed [**How to set up Multiple URLs for a Single Resource in the ASP.NET Core Web API**](https://dotnettutorials.net/lesson/multiple-urls-for-a-single-resource-using-routing/) Application. We are also going to work with the same application that we created in our [**Routing in ASP.NET Core Web API**](https://dotnettutorials.net/lesson/routing-in-asp-net-core-web-api/) article.

**What are Tokens in ASP.NET Core Attribute Routing?**

Token Replacement is a new feature available in ASP.NET Core and it was not available in .NET Frameworks like ASP.NET MVC and ASP.NET Web API. The meaning of token replacement is, we can replace the value of the controller and action method dynamically.

In ASP.NET Core Web API Application, the Route Attribute support token replacement. It means we can enclose the token (i.e. controller and action) within a pair of square braces ([]). The tokens (i.e. [controller] and [action]) are then replaced with the values of controller and action method name where the route is defined.

**Example: without using Token Replacement in ASP.NET Core Web API**

Before understanding the need and use of token replacement, let us first understand an example without using token replacement. Suppose we have two resources in our Employee Controller and we want to access the two resources using the controller/action method name. Then we can do the same without using token replacement as shown in the below code.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"Employee/GetAllEmployees"**)]**

**public** string GetAllEmployees**()**

**{**

**return** "Response from GetAllEmployees Method";

**}**

**[**Route**(**"Employee/GetAllDepartment"**)]**

**public** string GetAllDepartment**()**

**{**

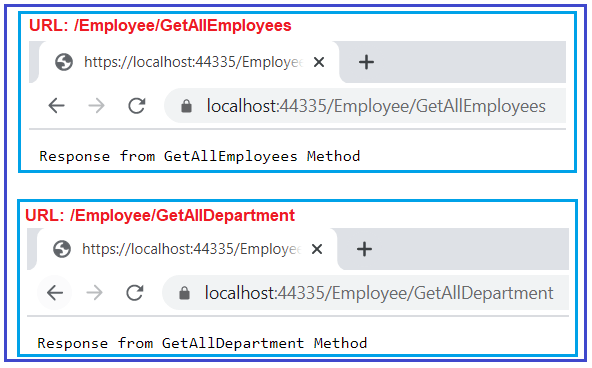
**return** "Response from GetAllDepartment Method";

**}**

**}**

**}**

Now run the application and you can access both the resource using the controller and action method name as shown in the below image.



**Example: Token Replacement in ASP.NET Core Web API Application**

Let us understand how the token replacement work in ASP.NET Core Web API Application with an example. Please modify the Employee Controller class that we have been working so far as shown below. As you can see, here we are applying the token **[controller]** on the EmployeeController and at the same time, we are also applying the token **[action]** on all the action methods of the Employee Controller.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**[**Route**(**"[controller]"**)]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"[action]"**)]**

**public** string GetAllEmployees**()**

**{**

**return** "Response from GetAllEmployees Method";

**}**

**[**Route**(**"[action]"**)]**

**public** string GetAllDepartment**()**

**{**

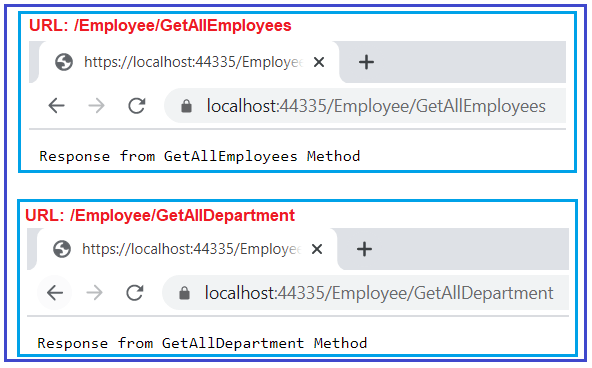
**return** "Response from GetAllDepartment Method";

**}**

**}**

**}**

Now, with the above changes in place, you can access the **GetAllEmployees** method with the URL: **/Employee/GetAllEmployees** and **GetAllDepartment** method with the URL: **/Employee/GetAllDepartment** as shown in the below image. This is because at the run time, the token **[controller]** will be replaced by Employee and the token **[action]** will be replaced by the respective action method of the controller.



**Advantages of Tokens in Attribute Routing:**

The main advantage of using Token Replacement in ASP.NET Core Application is that, if you rename the controller’s name or the action method name then you do not have to change the route templates. The application is going to works with the new controller and action method names.

**Do we need to write the action token on each action method?**

Not Really. If you want all your action methods of the controller to apply an action token, then instead of including the [action] token on each and every action method, you can apply it only once on the controller.

Let us understand this with an example. Please modify the Employee Controller class as shown in the below code. As you can see in the below code, we have removed the [Route(“[action]”)] attribute from the action method and modify the Route attribute as [Route(“[controller]/[action]”)] which is applied at the controller level.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**[**Route**(**"[controller]/[action]"**)]**

**public** **class** EmployeeController : ControllerBase

**{**

**public** string GetAllEmployees**()**

**{**

**return** "Response from GetAllEmployees Method";

**}**

**public** string GetAllDepartment**()**

**{**

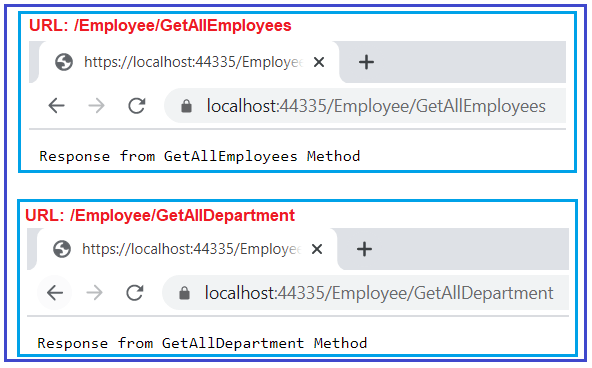
**return** "Response from GetAllDepartment Method";

**}**

**}**

**}**

Now, with the above changes in place, you can also access the **GetAllEmployees** method with the URL: **/Employee/GetAllEmployees** and **GetAllDepartment** method with the URL: **/Employee/GetAllDepartment** as shown in the below image.



**Token Replacement with dynamic Values:**

Now let us see an example of token replacement with dynamic values. Our requirement is to fetch employee information by employee id. Here, the employee id is the dynamic value and this value will come as part of the URL. Let us see how we can achieve this. Here, we need to decorate the Route attribute at the action method as shown below.

**[**Route**(**"{Id}"**)]**

**public** string GetEmployeeById**(int** Id**)**

**{**

**return** $"Response from GetEmployeeById Method, Id : {Id}";

**}**

So, at the run time, the [Route(“[controller]/[action]”)] will evaluate first and then [Route(“{Id}”)] will be added to the route which will form the URL as **Employee/GetEmployeeById/10**. So, modify the Employee Controller class as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**[**Route**(**"[controller]/[action]"**)]**

**public** **class** EmployeeController : ControllerBase

**{**

**public** string GetAllEmployees**()**

**{**

**return** "Response from GetAllEmployees Method";

**}**

**public** string GetAllDepartment**()**

**{**

**return** "Response from GetAllDepartment Method";

**}**

**[**Route**(**"{Id}"**)]**

**public** string GetEmployeeById**(int** Id**)**

**{**

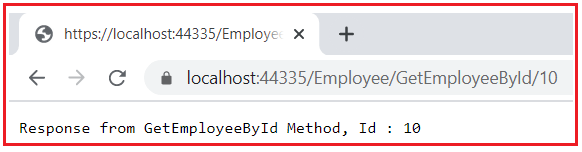
**return** $"Response from GetEmployeeById Method, Id : {Id}";

**}**

**}**

**}**

Now save the changes, run the application and access the **GetEmployeeById** as shown in the below image and you should the response as expected.



**Example: controller and action token applied to the action method**

In the below example, we are applying the Route Attribute only at the action method level and it is also going to work as expected.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"[controller]/[action]"**)]**

**public** string GetAllEmployees**()**

**{**

**return** "Response from GetAllEmployees Method";

**}**

**[**Route**(**"[controller]/[action]"**)]**

**public** string GetAllDepartment**()**

**{**

**return** "Response from GetAllDepartment Method";

**}**

**[**Route**(**"[controller]/[action]/{Id}"**)]**

**public** string GetEmployeeById**(int** Id**)**

**{**

**return** $"Response from GetEmployeeById Method, Id : {Id}";

**}**

**}**

**}**

It is always recommended to use the common part of the Route at the Controller level. In fact, in the next article, I am going to discuss [**How to set the common or base route at the controller level in ASP.NET Core Web API Routing**](https://dotnettutorials.net/lesson/base-route-asp-net-core-web-api-routing/) with Examples. Here, in this article, I try to explain **Token Replacement in ASP.NET Core Web API Attribute Routing** with Examples. I hope you enjoy this Token Replacement in the ASP.NET Core Web API Attribute Routing article.

**Base Route in ASP.NET Core Web API Routing**

**Base Route in ASP.NET Core Web API Routing**

In this article, I am going to discuss **How to set the common route or Base Route in ASP.NET Core Web API Routing** with Examples. Please read our previous article, where we discussed [**Token Replacement in ASP.NET Core Web AP**I](https://dotnettutorials.net/lesson/token-replacement-asp-net-core-routing/) Routing. We are also going to work with the same application that we created in our [**Routing in ASP.NET Core Web API**](https://dotnettutorials.net/lesson/routing-in-asp-net-core-web-api/) article.

**Base Route in ASP.NET Core Web API Routing**

Let’s understand the need and use of Base Route in ASP.NET Core Web API Routing with an Example. Let us first modify the EmployeeController class as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"employee/all"**)]**

**public** string GetAllEmployees**()**

**{**

**return** "Response from GetAllEmployees Method";

**}**

**[**Route**(**"employee/{Id}"**)]**

**public** string GetEmployeeById**(int** Id**)**

**{**

**return** $"Response from GetEmployeeById Method, Id : {Id}";

**}**

**[**Route**(**"employee/department/{Department}"**)]**

**public** string GetDepartmentEmployees**(**string Department**)**

**{**

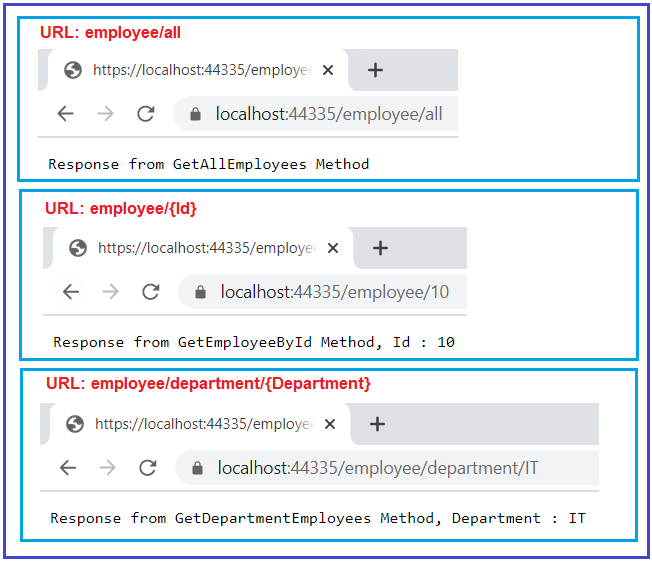
**return** $"Response from GetDepartmentEmployees Method, Department : {Department}";

**}**

**}**

**}**

Now you can access the above three methods as shown in the below image.



As you can see in the above example, we are using the Route attributes at the action level to define the routes, and furthermore, all the routes in the **EmployeeController** are starts with the same prefix – employee. That means the employee is the common prefix that is used for all the routes available in the Employee Controller at the moment.

Is not it good enough, if you can move the common attribute prefix to the controller level? Yes, we can. So, basically, the prefix which is common for all the routes should be placed at the controller level and the route which is specific to the action method should be placed at the action method level. So, in our example, the common prefix i.e. employee should be placed at the controller level and the rest should be placed at the action method level as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**[**Route**(**"employee"**)]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"all"**)]**

**public** string GetAllEmployees**()**

**{**

**return** "Response from GetAllEmployees Method";

**}**

**[**Route**(**"{Id}"**)]**

**public** string GetEmployeeById**(int** Id**)**

**{**

**return** $"Response from GetEmployeeById Method, Id : {Id}";

**}**

**[**Route**(**"department/{Department}"**)]**

**public** string GetDepartmentEmployees**(**string Department**)**

**{**

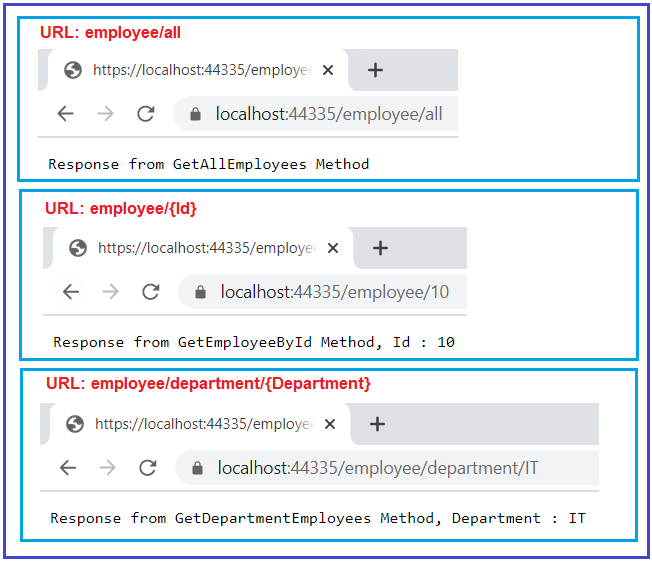
**return** $"Response from GetDepartmentEmployees Method, Department : {Department}";

**}**

**}**

**}**

Now with the above changes in place, you can access the above three resources in the same way that we access in our previous example as shown in the below image.



With the above changes in place, we eliminate the need to repeat the common prefix “employee” on each and every controller action method. However, sometimes we may need to override the common route prefix attribute.

**How to override the Base Route in ASP.NET Core Web API Attribute Routing?**

Let us understand how to override the common route prefix or base route in ASP.NET Core Web API Attribute Routing with an example. At this moment our Employee Controller class contains three action methods and all these action methods start with the same Route Prefix i.e. employee.

Now, we need to add one action method within the employee controller for returning all departments. And we want to access this resource using the URL: department/all. So, let us first add the following GetAllDepartment method and decorate it with [Route(“department/all”)] Attribute as shown below within the EmployeeController.

**[**Route**(**"department/all"**)]**

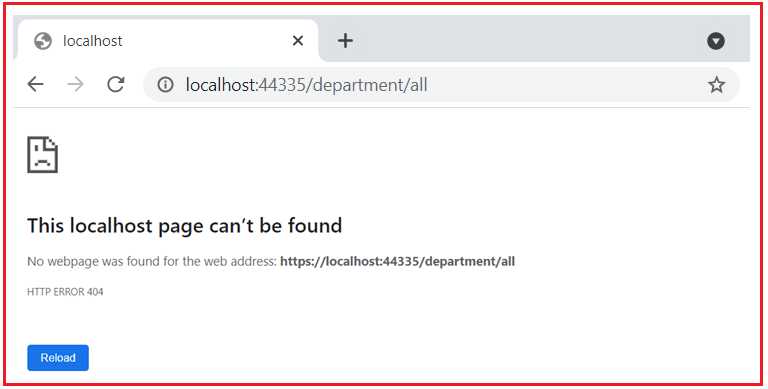
**public** string GetAllDepartment**()**

**{**

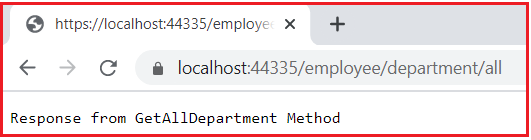
**return** "Response from GetAllDepartment Method";

**}**

With the above **[Route(“department/all”)]**attribute on GetAllDepartment() method and when we navigate to **department/all**, we will get the following error.



But if we navigate to **/employee/department/all** then we will get the output as expected as shown in the below image. This is because the [Route(“employee”)] attribute that is defined at the Employee Controller.



Now the question that should come to your mind is how to override the Route attribute used in the EmployeeController i.e. Route Attribute defined at the controller level.

**How to override the Controller level Route Attribute at the action method level?**

In ASP.NET Core Application, you can override the Controller level Route Attribute at the action method level by using the **~** (tilde) symbol. So, modify the GetAllDepartment action method as shown below to use the tilde symbol to override the route defined at the employee controller.

**[**Route**(**"~/department/all"**)]**

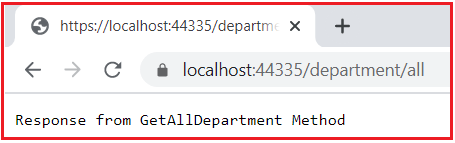
**public** string GetAllDepartment**()**

**{**

**return** "Response from GetAllDepartment Method";

**}**

With the above change in place, now the **GetAllDepartment**() action method is mapped to URI “**/department/all**” as expected as shown in the below image.



# Route Constraints in ASP.NET Core Web API

**Route Constraints in ASP.NET Core Web API**

In this article, I am going to discuss **Route Constraints in ASP.NET Core Web API** Application with Examples. Please read our previous article, where we discussed How to set the [**Common Route or Base Route in ASP.NET Core Web API**](https://dotnettutorials.net/lesson/base-route-asp-net-core-web-api-routing/) Routing. We are also going to work with the same application that we created in our [**Routing in ASP.NET Core Web API**](https://dotnettutorials.net/lesson/routing-in-asp-net-core-web-api/)article. As part of this article, we are going to discuss the following Route Constraints.

1. **Type: int, double, bool, float, datetime, etc**
2. **Min: min(number)**
3. **Max: max(number)**
4. **Range: range(10. 15)**
5. **Alpha: alpha**
6. **MinLength: minlength(5)**
7. **MaxLength: maxlength(10)**
8. **Length: length(10)**
9. **Required: required**
10. **Regex: regex(expression)**

**ASP.NET Core Web API Attribute Routing with Route Constraints**

The Route Constraints in ASP.NET Core Web API Attribute Routing are nothing but a set of rules that can be applied to routing parameters to restrict how the parameters in the route template are matched. The syntax to use Route Constraints is: **{parameter:constraint}**

**Examples to Understand Route Constraints in ASP.NET Core Web API**

Let us understand ASP.NET Core Web API Attribute Routing Route Constraints with some examples. Please modify the Employee Controller class as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**[**Route**(**"api/[controller]"**)]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"{EmployeeId}"**)]**

**public** string GetEmployeeDetails**(int** EmployeeId**)**

**{**

**return** $"Response from GetEmployeeDetails Method, EmployeeId : {EmployeeId}";

**}**

**[**Route**(**"{EmployeeName}"**)]**

**public** string GetEmployeeDetails**(**string EmployeeName**)**

**{**

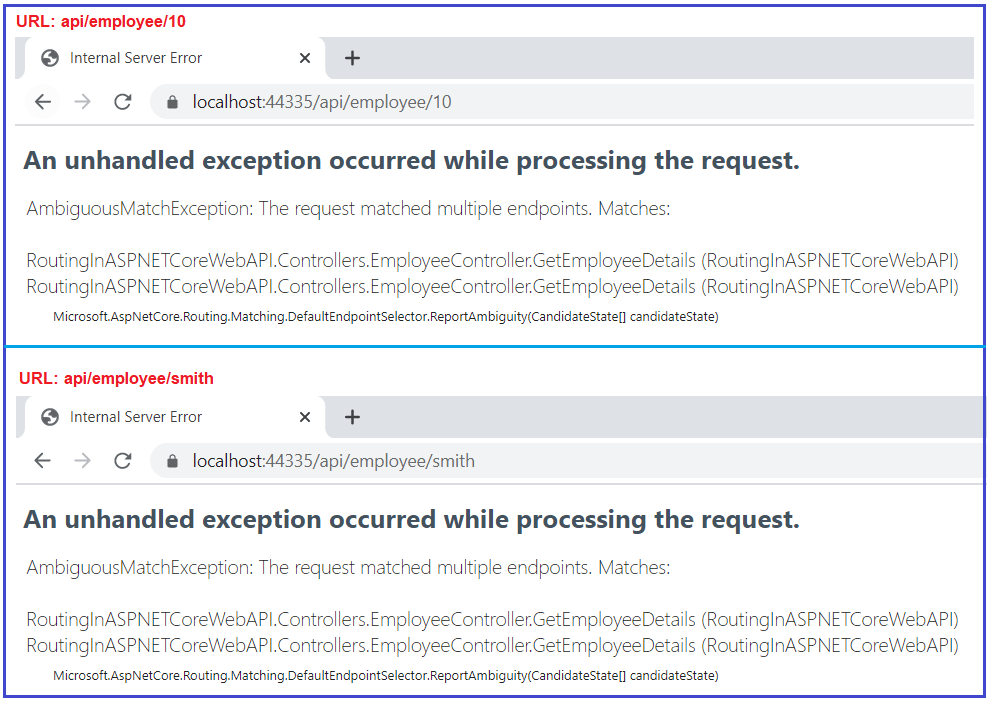
**return** $"Response from GetEmployeeDetails Method, EmployeeName : {EmployeeName}";

**}**

**}**

**}**

With the above changes in place, now the run application and navigate to the URL **api/employee/10** and **api/employee/smith**, and in both cases you will get the following error.



This is because, when the request comes, the application does not identify which version of the GetEmployeeDetails() method to use, and hence it gives an Ambiguous Match Exception as two end pints match the same request. This is the situation where the Route Constraints come into the picture in ASP.NET Core Web API.

**How to use Route Constraint in ASP.NET Core Web API?**

What we want to achieve is, if an integer is specified in the URL like **api/employee/10**, then we need to execute the **GetEmployeeDetails(int EmployeeId)** method which takes an integer parameter whereas if a string is specified in the URL like **api/employee/smith**, then we need to execute the **GetEmployeeDetails(string EmployeeName)** method of the Employee Controller which takes the string as a parameter.

This can be very easily achieved in ASP.NET Core Web API Application using Route Constraints. We need to use the following syntax to specify the route constraint,

**{parameter:constraint}**

**Type Route Constraint in ASP.NET Core Web API:**

We can use the type constraint to specify the parameter type. The different types of type constraints supported in ASP.NET Core Web Application are int, decimal, float, long, double, bool, etc. So, with the type constraint in place, if an integer is specified in the URL, then the GetEmployeeDetails(int EmployeeId) action method is invoked and if a string is specified in the URL then the GetEmployeeDetails(string EmployeeName) method is invoked.

**Example: int type constraint:**

If you want any parameter to accept only integer values then you need to specify the int type constraint. So, let us decorate the GetEmployeeDetails method which takes an integer parameter with the following Route Attribute.

**[Route(“{EmployeeId:int}”)]**

**Note:** we don’t need to specify anything for the string parameter as by default all the parameters in ASP.NET Core Web Application are string only.

**Let’s modify the Employee Controller to use the above-discussed int type Route Constraints as shown below.**

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *RoutingInASPNETCoreWebAPI.Controllers*

**{**

**[**ApiController**]**

**[**Route**(**"api/[controller]"**)]**

**public** **class** EmployeeController : ControllerBase

**{**

**[**Route**(**"{EmployeeId:int}"**)]**

**public** string GetEmployeeDetails**(int** EmployeeId**)**

**{**

**return** $"Response from GetEmployeeDetails Method, EmployeeId : {EmployeeId}";

**}**

**[**Route**(**"{EmployeeName}"**)]**

**public** string GetEmployeeDetails**(**string EmployeeName**)**

**{**

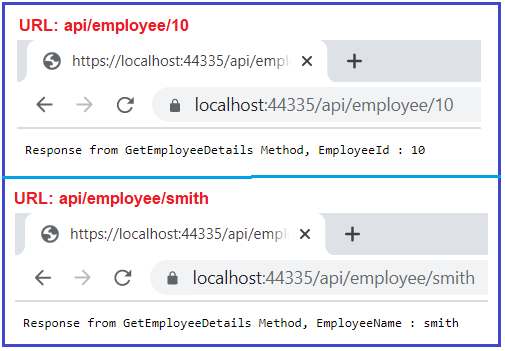
**return** $"Response from GetEmployeeDetails Method, EmployeeName : {EmployeeName}";

**}**

**}**

**}**

With the above changes in place, now run the application and navigate to **api/employee/10** and **api/employee/smith** URLs and you should get the following output.



**Min(number) constraint in ASP.NET Core:**

If you want to apply some minimum value constraint for any parameter then you can use the Min constraint. The min constraint takes one parameter i.e. the minimum value to be applied on the parameter. For example, if you want the GetEmployeeDetails(int EmployeeId) action method to be invoked only if the EmployeeId is a number greater than 1000, then you can use the min(1000) constraint in ASP.NET Core Web API as shown below.

**[**Route**(**"{EmployeeId:int:min(1000)}"**)]**

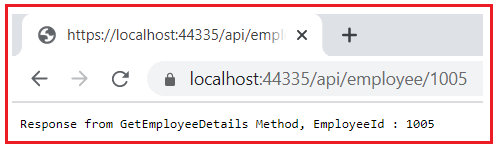
**public** string GetEmployeeDetails**(int** EmployeeId**)**

**{**

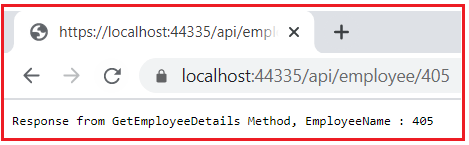
**return** $"Response from GetEmployeeDetails Method, EmployeeId : {EmployeeId}";

**}**

Now run the application and navigate to the URL by passing a value greater than 1000 and you should get the following output.



Now, if you pass a value less than 1000, then the other GetEmployeeDetails method which takes string parameter is executed as shown in the below image.



This is because the second GetEmployeeDetails method takes a string parameter and here it treats the value 405 as a string and executes that method.

**Alpha constraint in ASP.NET Core Web API:**

If you want any parameter to accept only alphabets (a to z characters) values then you need to specify the alpha constraint. So, let us decorate the GetEmployeeDetails method which takes string parameters with the following Route Attribute.

**[Route(“{EmployeeName:alpha}”)]**

Here, alpha stands for uppercase or lowercase alphabet characters. So, changes the GetEmployeeDetails method which takes the string parameter as shown below.

**[**Route**(**"{EmployeeName:alpha}"**)]**

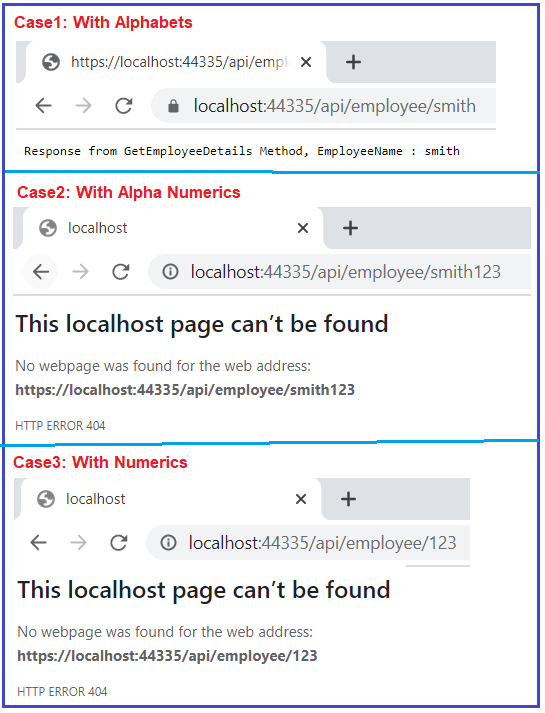
**public** string GetEmployeeDetails**(**string EmployeeName**)**

**{**

**return** $"Response from GetEmployeeDetails Method, EmployeeName : {EmployeeName}";

**}**

With the above changes in place, now run the application and test the below three cases. Case1: with only alphabets and it should work. Case2: with alphanumeric and should not work. Case3: with an integer value less than 1000 and it should not work. All these three cases are shown in the below image.



**Max(Number) constraint in ASP.NET Core Web API:**

Along with the min constraint, you can also specify the max constraint in ASP.NET Core Web API. The max constraint also takes one parameter which is used to specify the max value that can be applied to the parameter. For example, if you want the EmployeeId in the URL should not to be greater than 1000, then you can use the max constraint as shown below.

**[**Route**(**"{EmployeeId:int:max(1000)}"**)]**

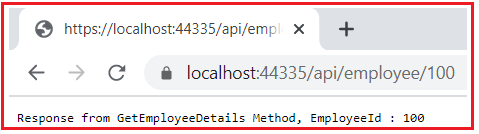
**public** string GetEmployeeDetails**(int** EmployeeId**)**

**{**

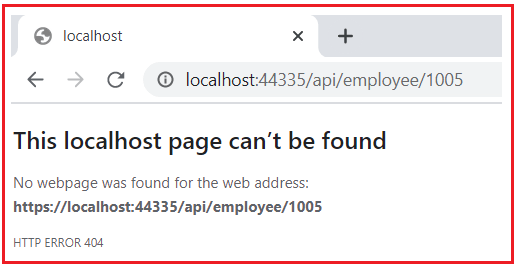
**return** $"Response from GetEmployeeDetails Method, EmployeeId : {EmployeeId}";

**}**

Now save the changes and run the application and navigate to the URL by passing a value less than 1000 and you should get the following output.



Now, if you pass a value greater than 1000, then you will get the following resource not found 404 error page.



It is also possible to use both min and max constraints for a single route. For example, if you want the EmployeeId value in the URL to be between 100 and 1000 inclusive, then we can specify both min and max constraints as shown below.

**[**Route**(**"{EmployeeId:int:min(100):max(1000)}"**)]**

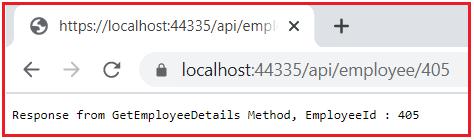
**public** string GetEmployeeDetails**(int** EmployeeId**)**

**{**

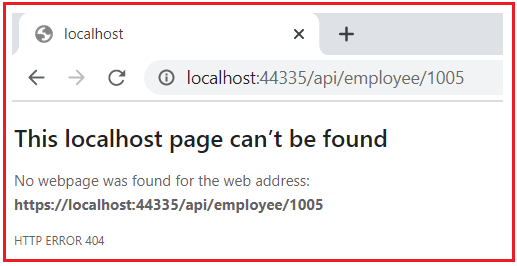
**return** $"Response from GetEmployeeDetails Method, EmployeeId : {EmployeeId}";

**}**

Now save the changes and run the application and navigate to the URL by passing a value between 100 and 1000 and you should get the response as expected as shown in the below image.



Now, if you pass a value greater than 1000 or less than 100, then you will get the following resource not found 404 error page.



**Range Constraint in ASP.NET Core:**

Instead of using the min and max constraint to specify the minimum and maximum value, we can also use the Range constraint. The Range method takes two parameters, the first parameter is the minimum value and the second parameter is the maximum value. Let us rewrite the previous example using the Range method bypassing 100 and 1000 as the two parameters as shown below.

**[**Route**(**"{EmployeeId:int:range(100,1000)}"**)]**

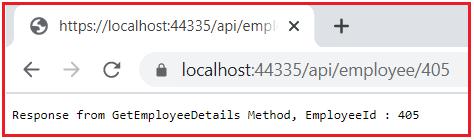
**public** string GetEmployeeDetails**(int** EmployeeId**)**

**{**

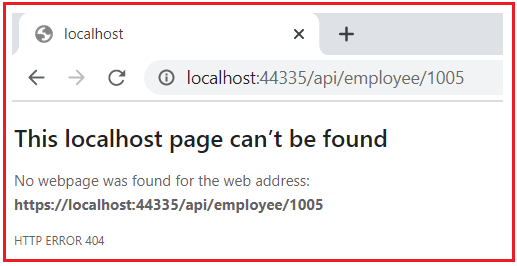
**return** $"Response from GetEmployeeDetails Method, EmployeeId : {EmployeeId}";

**}**

With the above changes in place, run the application and navigate to the URL by passing a value between 100 and 1000 and you should get the response as expected as shown in the below image.



Now, if you pass a value greater than 1000 or less than 100, then you will get the following resource not found 404 error page.



**MinLength Route Constraint in ASP.NET Core Web API:**

The MinLength constraint is used to specify the minimum length constraint on the string parameter. For example, the GetEmployeeDetails(string EmployeeName) method takes one string parameter and we want if the length of the input parameter is greater than 5 characters then only invoke this method. We can achieve this very easily by using the minlength constraint which takes one parameter to specify the minimum length. So, let modify the GetEmployeeDetails(string EmployeeName) action method as shown below.

**[**Route**(**"{EmployeeName:alpha:minlength(5)}"**)]**

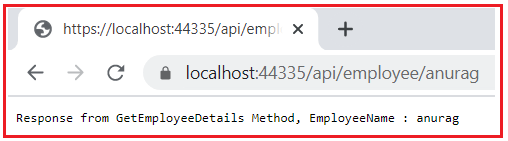
**public** string GetEmployeeDetails**(**string EmployeeName**)**

**{**

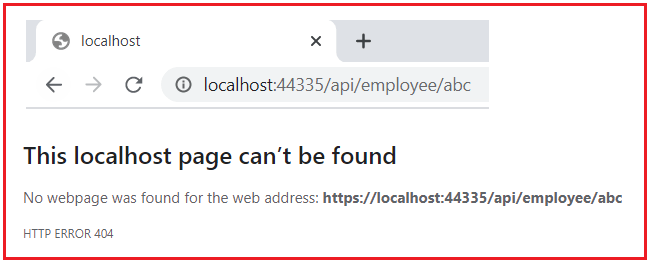
**return** $"Response from GetEmployeeDetails Method, EmployeeName : {EmployeeName}";

**}**

With the above changes in place, run the application and pass a string of more than 5 characters and you should see the following response.



Now, if you pass a string less than 5 characters in URL like **api/employee/abc**, then you will get the resource not found 404 error page as shown in the below image.



**MaxLength Route Constraint in ASP.NET Core Web API:**

Just like the minlength constraint, we can also apply the MinLength constraint to specify the maximum length on the string parameter. For example, the GetEmployeeDetails(string EmployeeName) method takes one string parameter, we want if the length of the input parameter is less than 10 characters then only invoke the method.

We can achieve the above very easily by using the maxlength constraint which takes one parameter to specify the maximum length. So, let modify the GetEmployeeDetails(string EmployeeName) action method as shown below.

**[**Route**(**"{EmployeeName:alpha:maxlength(10)}"**)]**

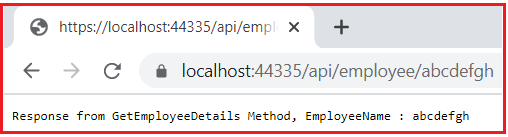
**public** string GetEmployeeDetails**(**string EmployeeName**)**

**{**

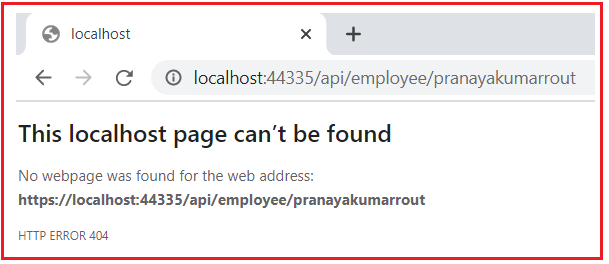
**return** $"Response from GetEmployeeDetails Method, EmployeeName : {EmployeeName}";

**}**

With the above changes in place, now run the application and pass a string of less than 10 characters and it should work and you should get the following response.



Now, if you pass a string greater than 10 characters in URL like **api/employee/pranayakumarrout**, then you will get the following resource not found 404 error page.



It is also possible to apply both minlength and maxlength constraints on a single parameter. For example, if we want to specify the EmployeeName parameter to a minimum of 5 characters and a maximum of 10 characters, then we can apply both minlength and maxlength constraints as shown below.

**[**Route**(**"{EmployeeName:alpha:minlength(5):maxlength(10)}"**)]**

**public** string GetEmployeeDetails**(**string EmployeeName**)**

**{**

**return** $"Response from GetEmployeeDetails Method, EmployeeName : {EmployeeName}";

**}**

Now run the application and test the same by yourself by passing a string between 5 and 10 characters as well as a string less than 5 and greater than 10 characters.

**Length Route Constraint in ASP.NET Core Web API:**

The Length Route Constraint is basically used to specify the exact length of a string. This constraint takes one parameter which specifies the length to be applied for the parameter. For example, we want the GetEmployeeDetails(string EmployeeName) method to be invoked only when the EmployeeName is five characters. Then we can achieve the same very easily by using the length route constraint. So, please modify the GetEmployeeDetails(string EmployeeName) method as shown below.

**[**Route**(**"{EmployeeName:alpha:length(5)}"**)]**

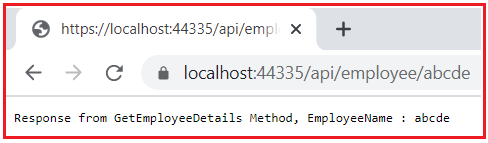
**public** string GetEmployeeDetails**(**string EmployeeName**)**

**{**

**return** $"Response from GetEmployeeDetails Method, EmployeeName : {EmployeeName}";

**}**

Save the above changes and run the application and pass a string with exactly 5 characters in the URL and you should get the following response.



Now, if you pass a string less than 5 like **api/employee/abcd** or a string greater than 5 like **api/employee/abcdef**, then you will get the resource not found 404 error page.

**Regex Route Constraint in ASP.NET Core Web API:**

You can also validate the value of a particular variable by using the regex route constraint. The regex takes one parameter and you can specify an expression or pattern to validate. Let us understand this with an example. Modify the **GetEmployeeDetails(string EmployeeName)** method as shown below. Here, we are adding a simple pattern that will validate if the string starts with the letter a and followed by b or c, then this method is going to be invoked.

**[**Route**(**"{EmployeeName:regex(a(b|c))}"**)]**

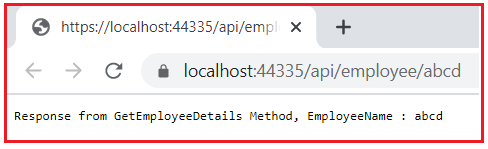
**public** string GetEmployeeDetails**(**string EmployeeName**)**

**{**

**return** $"Response from GetEmployeeDetails Method, EmployeeName : {EmployeeName}";

**}**

Save the above changes and run the application and pass abcd string in the URL and you should get the following response.



Now, if you pass a string that does not start with a or if it starts with a but not followed by b or c character, then you will get the resource not found 404 error page.

In the next article, I am going to discuss the different [**Controller Action Method Return Types in ASP.NET Core Web API**](https://dotnettutorials.net/lesson/controller-action-return-types-core-web-api/) Application with Examples. Here, in this article, I try to explain **Route Constraints in ASP.NET Core Web API** Application with Examples. And I hope you enjoy this article.