**Build ASP.NET Core Web API Project From Scratch**

**How to Build ASP.NET Core Web API Project from Scratch?**

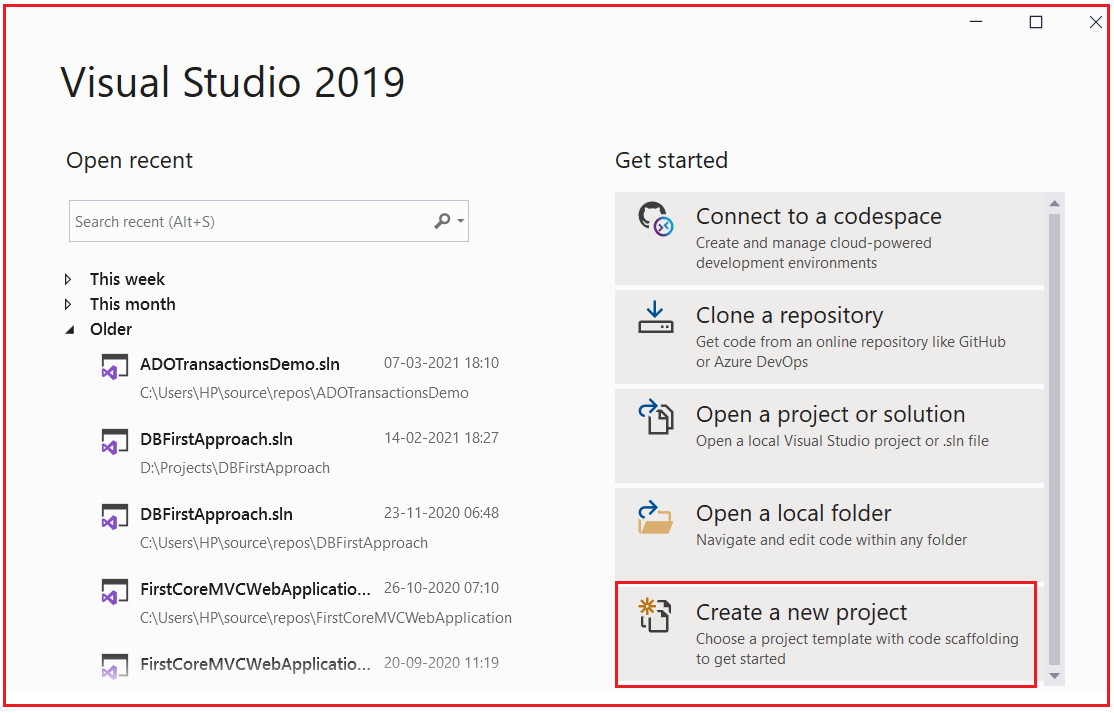
In this article, I am going to discuss **How to Build ASP.NET Core Web API Project from Scratch** i.e. how to convert a console application to a Web API Application. In our [**previous article**](https://dotnettutorials.net/lesson/asp-net-core-web-api-project-in-visual-studio-2019/), we discussed the default Web API template and, in this article, we are going to discuss what are the component and how they are working behind the scene. So, what we will do here in this article, we will create an ASP.NET Core Console Application, and then we will convert that console application into ASP.NET Core Web API Application step by step.

**Converting Console Application to Web API Application:**

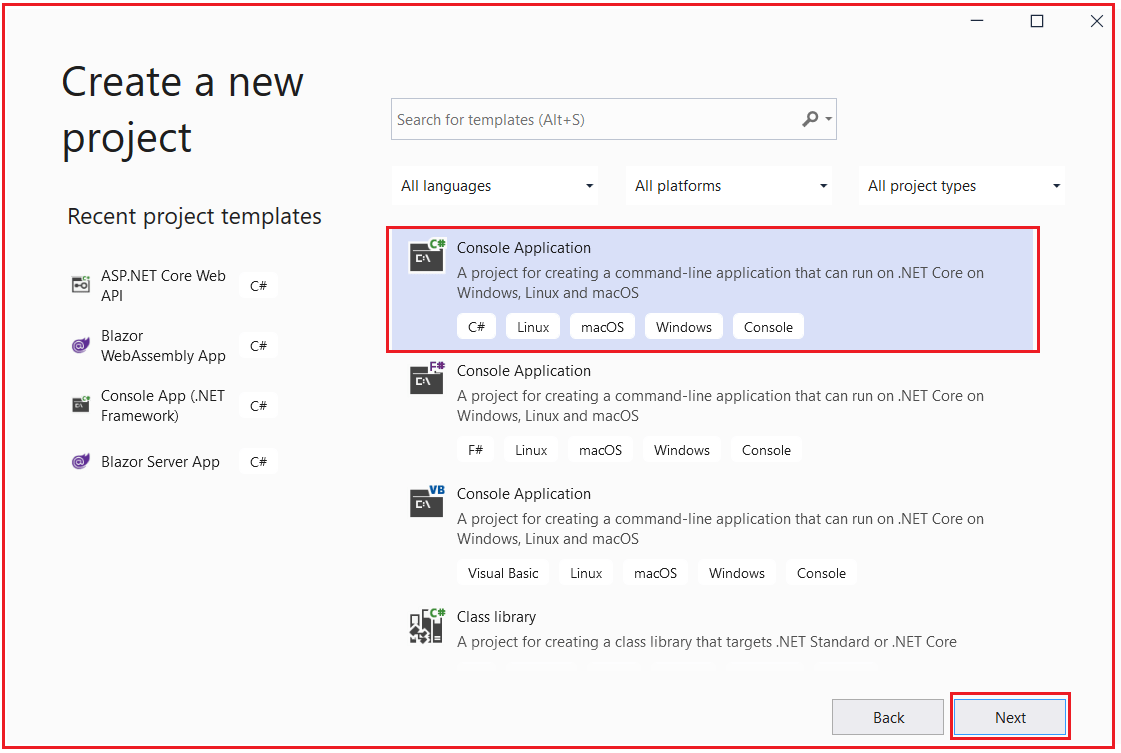
In the case of ASP.NET Core, by default, each .NET Core Application is a Console Application. This is the reason why you can see by default the Program.cs class file with the Main method in each .NET Core Web App. But then we need to add some services (specific to application type) and also need to do some changes to convert the console application to a Web API Project.

**Create a new Console Application**

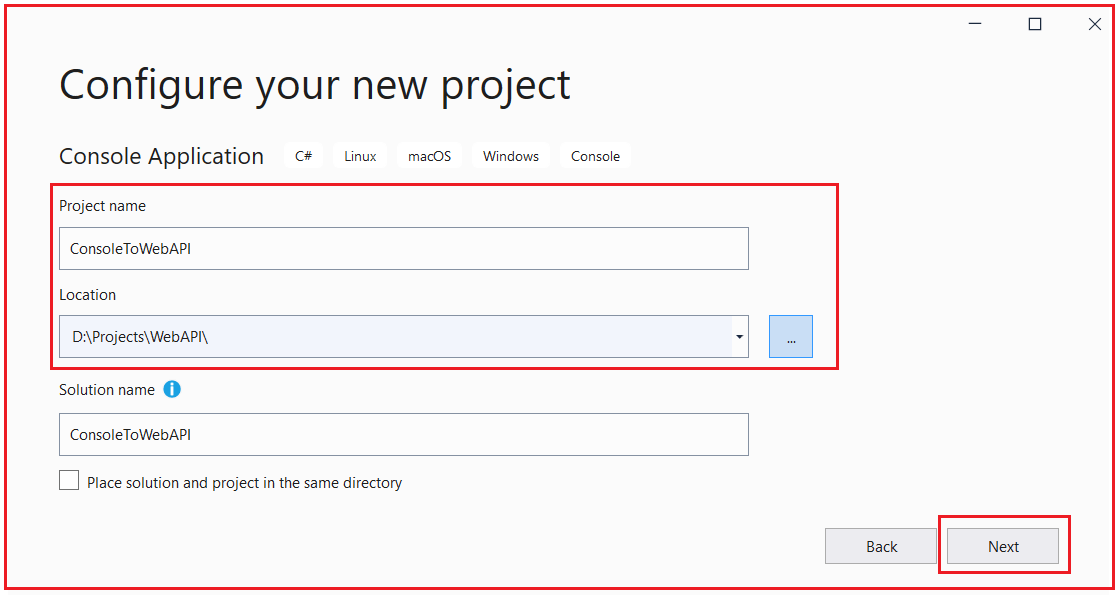
Open Visual Studio and then click on the Create a new Project option as shown in the below image.



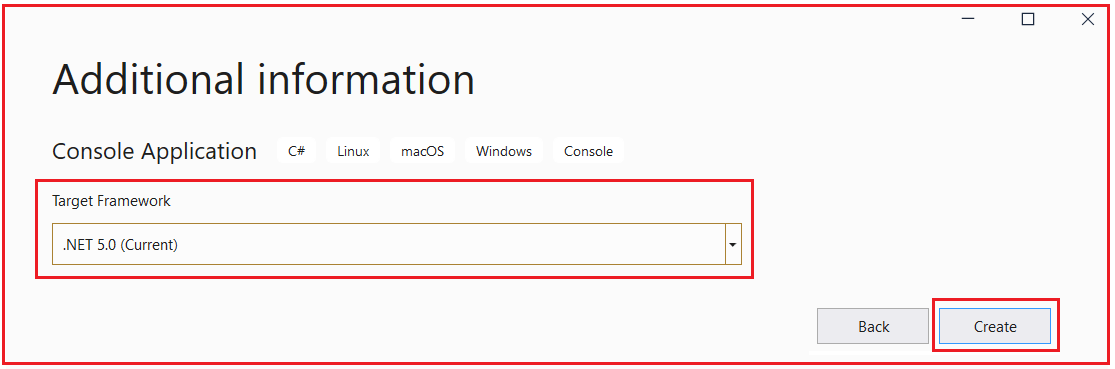
Then from the next Create a Project window, select Console Application that targets .NET Core Framework and click on the Next button as shown in the below image.



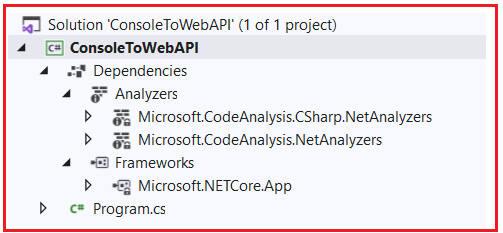
From the next Configure your new project window, give a meaningful name to your project (I am giving the name ConsoleToWebAPI), and select the location where we want to create the console application. Then if you want you can give a different name to your solution but I am going with the default one. Finally, click on the Next button as shown in the below image.



From the next Additional Information window, select the target .NET Core Framework. As we are working with .NET 5, so select .NET 5 and click on the Create button as shown in the below image.



Once you click on the Create button, it will create the Console Application with the following structure. As you can see, it contains Program.cs class file and Dependencies Folder. Inside the Dependencies folder, it also has the Analyzers and Frameworks folder. At this moment the Frameworks folder contains only Microsoft.NETCore.App. Also, notice the project type as C#.



Now open the Program.cs class file and you should the following in it which includes the Main method which is the entry point to this console application.

**using** *System;*

**namespace** *ConsoleToWebAPI*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

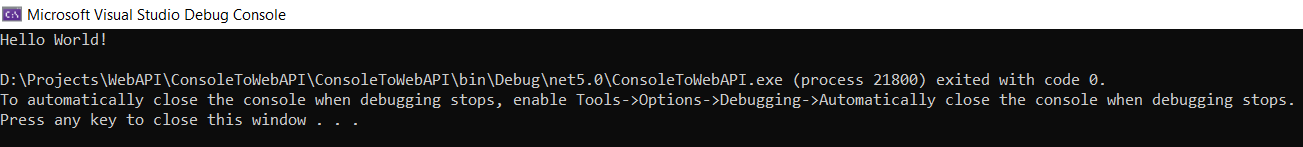
Console.WriteLine**(**"Hello World!"**)**;

**}**

**}**

**}**

Now if you run the application, then you will get the following output in the console window. That means our console application is working fine.



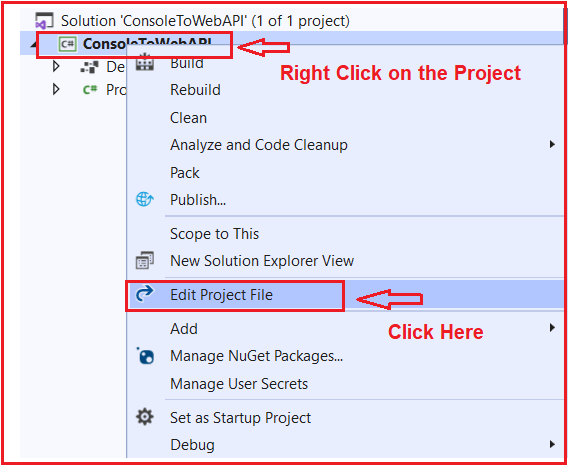
Now we will do some changes to this ASP.NET Core Console Application in order to make it an ASP.NET Core Web API Application.

**Changing in CSPROJ file (Project file)**

In order to make the console application a Web API Application, first of all, we need to make the following changed to the csproj file.

1. Change the Project SDK to Web
2. Remove the Project Output Type
3. Verify the Target Framework Moniker (TFM)

In order to open the csproj file of this console application, we have two options either by double-clicking on the project or by right-clicking on the project and select the **Edit Project file** option as shown in the below image.



Once you open the Project file, you will get the following configuration inside the project file. As you can see in the below code, the Project SDK is Microsoft.NET.Sdk and the OutputType is Exe and the TargerFramework is net5.0.

**<Project** Sdk="Microsoft.NET.Sdk"**>**

**<PropertyGroup>**

**<OutputType>**Exe**</OutputType>**

**<TargetFramework>**net5.0**</TargetFramework>**

**</PropertyGroup>**

**</Project>**

So, we need to do some changes to the above CSPROJ file in order to make this project a Web API Project. As we are going to deal with ASP.NET Core Web Application, so we need to set the SDK type as Microsoft.NET.Sdk.Web. Again, as it is no longer a console application, so we need to remove the OutputType which is set to Exe from the Project file. Then we need to set the TargetFranmework. As already it is set to net5.0, which means it is dealing with .NET 5, so we don’t require to do any changes. In case if it is target to some lower version of the .NET Framework, then change it to net5.0. So, with the above changes in place, your CSPROJ file should looks as shown below.

**<Project** Sdk="Microsoft.NET.Sdk.Web"**>**

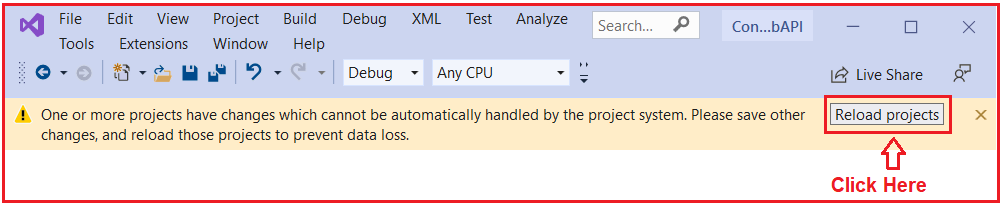
**<PropertyGroup>**

**<TargetFramework>**net5.0**</TargetFramework>**

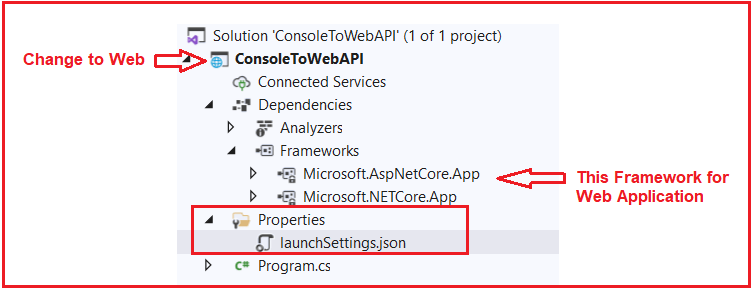
**</PropertyGroup>**

**</Project>**

Once you save the changes, it will give you one warning that one or more projects have changes that cannot be automatically handled by the project system. Please save other changes and reload those projects to prevent data loss. Simply click on the Reload Projects button as shown in the below image.



Once you click on the Reload Projects button, the warning gone and you can also observe the project structure also has been changed. Now, the project type is changed to the web, it also adds **Microsoft.AspNetCore.App** inside the Frameworks folder which provides support for ASP.NET Core Web App as well as it also adds **Properties** folder with **launchSettings.json** file as shown in the below image.



The launchSettings.json file contains all the settings that are required to run this application in the Development environment. You will find the following settings in the launchSettings.json file.

**{**

"iisSettings": **{**

"windowsAuthentication": **false,**

"anonymousAuthentication": **true,**

"iisExpress": **{**

"applicationUrl": "http://localhost:60654/"**,**

"sslPort": 44337

**}**

**},**

"profiles": **{**

"IIS Express": **{**

"commandName": "IISExpress"**,**

"launchBrowser": **true,**

"environmentVariables": **{**

"ASPNETCORE\_ENVIRONMENT": "Development"

**}**

**},**

"ConsoleToWebAPI": **{**

"commandName": "Project"**,**

"launchBrowser": **true,**

"environmentVariables": **{**

"ASPNETCORE\_ENVIRONMENT": "Development"

**},**

"applicationUrl": "https://localhost:5001;http://localhost:5000"

**}**

**}**

**}**

These are the changes required in the project file. But this is not enough to run this application at this moment. We need to do some changes to the Program.cs file i.e. we need to [**add Web Host Builder**](https://dotnettutorials.net/lesson/adding-web-host-builder/). That we will discuss in our next article. Here, in this article, I try to explain the changes that are required in the csproj file to convert a console application to a web application.

**Adding Web Host Builder**

**Adding Web Host Builder in ASP.NET Core**

In this article, I am going to discuss how to modify the Program class in order to add the Web Host Builder in ASP.NET Core Application. This is a continuation part to our previous article where we discussed [**how to modify the project file to convert a console application to an ASP.NET Core Web API Application**](https://dotnettutorials.net/lesson/build-asp-net-core-web-api-project/). So, please read our previous article before proceeding to this article.

**Changes in Program.cs class:**

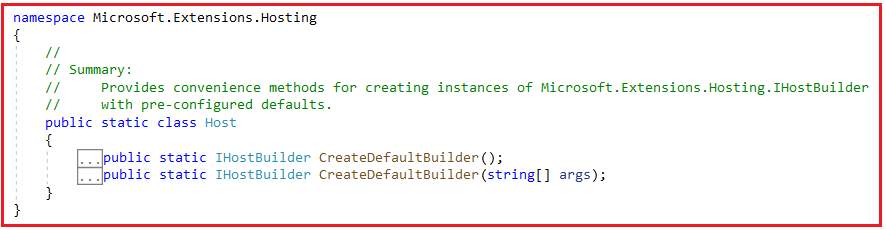
In order to make our console application a Web API Application, we also need to do some changes in our Program.cs class file which is the second step. We are going to do the following changes in our Program.cs class file

1. Adding the Web Host Builder
2. Configuring the Startup class.

Let us first understand what is Host Builder and then we will see how to add it.

**What is Host Builder?**

Host Builder is a static class that provides two methods and when we call these methods, they will add some features into ASP.NET Core Applications. The Host Builder Provides two convenient methods for creating instances of IHostBuilder with pre-configured defaults. If you go to the definition of Host class, then you will find two overloaded versions of the CreateDefaultBuilder method as shown in the below image.



**Note:** One method without parameter and the other method take string array as an argument. Let us discuss what pre-configured settings these two methods provide.

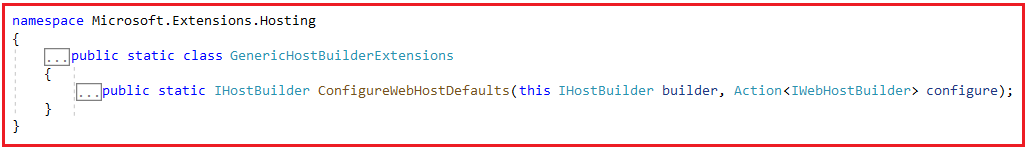
**CreateDefaultBuilder:**

The following defaults are applied to the returned Microsoft.Extensions.Hosting.HostBuilder:

1. By using the CreateDefaultBuilder method, we will get inbuilt support for Dependency Injection. This is one of the most popular design patterns used in almost all modern web applications and you get this by default when you add the CreateDefaultBuilder method. Also, enables scope validation on the dependency injection container when EnvironmentName is ‘Development’.
2. It load app Configurations from ‘appsettings.json’ and ‘appsettings.[EnvironmentName].json’. Configurations are basically the settings that our application uses.
3. The CreateDefaultBuilder method also set the ContentRootPath to the result of GetCurrentDirectory() method. The GetCurrentDirectory method belongs to the System.IO.Directory namespace.
4. The CreateDefaultBuilder method also configures the logging factory to log to the console, debug, and event source output. Nowadays, logging is mandatory for any kind of application. So, by using the CreateDefaultBuilder method, we also get logging support by default.
5. It also loads application Configuration from User Secrets when EnvironmentName is ‘Development’ using the entry assembly as well as load Configuration from environment variables.
6. It also loads host Configuration from supplied command line args. This is only applicable with the version which takes string array as an argument.

**ConfigureWebHostDefaults:**

Along with the CreateDefaultBuilder method, we also need to add the ConfigureWebHostDefaults method. This method Configures a HostBuilder with defaults for hosting a web app. Following is the signature of this method.



**Parameter**:

1. builder: The IHostBuilder instance to configure.
2. configure: The configure callback

**Returns**: A reference to the builder after the operation has completed.

The ConfigureWebHostDefaults method provides the following support.

1. Sets Kestrel Server as the Web Server and configure it using the application’s configuration providers. When we run the application using CLI, the kestrel web server is used.
2. Enables IIS Integration, etc.
3. It adds the Host Filtering middleware
4. Also adds the ForwardedHeaders middleware if ASPNETCORE\_FORWARDEDHEADERS\_ENABLED=true
5. It configures WebRootFileProvider to include static web assets from projects referenced by the entry assembly during the development

**Adding CreateHostBuilder Method in Program class:**

Before adding the CreateHostBuilder method, let us first add a class file with the name **Startup.cs** into our project. In our next article, we will discuss the use of Startup class. For now, simply create the Startup.cs class file. Once you add the Startup class, then modify the Program.cs class as shown in the below code. The IHostBuilder and Host class belong to Microsoft.Extensions.Hosting and the UseStartup class belongs to Microsoft.AspNetCore.Hosting namespace.

**using** *Microsoft.Extensions.Hosting;*

**using** *Microsoft.AspNetCore.Hosting;*

**using** *System;*

**namespace** *ConsoleToWebAPI*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

CreateHostBuilder**(**args**)**.Build**()**.Run**()**;

**}**

**public** **static** IHostBuilder CreateHostBuilder**(**string**[]** args**)** =**>**

Host.CreateDefaultBuilder**(**args**)**

.ConfigureWebHostDefaults**(**webBuilder =**>**

**{**

webBuilder.UseStartup**<**Startup**>()**;

**})**;

**}**

**}**

As you can see in the above code, we have created a method called CreateHostBuilder which takes string array as a parameter. Then we call the CreateDefaultBuilder method on the Host class bypassing the input array as an argument. Even you can also the other overloaded version of the CreateDefaultBuilder method which does not take any parameter. Then on the instance of the CreateDefaultBuilder method, we call the ConfigureWebHostDefaults method and also provide the Startup class name. In the Main method, first, we call the CreateHostBuilder method and then call the Build Method followed by the Run method.

**Build Method:**The Build Method runs the given actions to initialize the host. This can only be called once. It returns an initialized Microsoft.Extensions.Hosting.IHost object.

**Run Method:**It Runs the application and blocks the calling thread until the host shutdown.

Now if you compare the above code with the Program class auto-generated by ASP.NET Core Web API Application, then you will find everything as the same.

**Can I give some different names for the CreateHostBuilder method like MyCreateHostBuilder?**

If you are using some basic features of ASP.NET Core, then you will not face any problems. But if you use some features like Entity Framework core, then it will not work. The reason is if you are using EF Core, then EF Core will search for this CreateHostBuilder method to add some default configurations.

**Configuring Startup Class in ASP.NET Core**

**Configuring Startup Class in ASP.NET Core**

In this article, I am going to discuss **Configuring the Startup Class in ASP.NET Core** Application i.e. How to Configure ConfigureService and Configure Method of the Startup class. This is step3 to convert a Console Application to a Web API Application. So, please read [**Step1**](https://dotnettutorials.net/lesson/build-asp-net-core-web-api-project/) and [**Step2**](https://dotnettutorials.net/lesson/adding-web-host-builder/) before proceeding to this article.

**Configuring the Startup Class in ASP.NET Core**

The Startup class is a very important class for any type of ASP.NET Core Web Application. This class provides two important methods are as follows:

1. **ConfigureService**
2. **Configure**

**ConfigureService Method:**

The ConfigureService method is used to configure all the services that you want to use in this application. The ConfigureService method takes IServiceCollection as an input parameter. The IServiceCollection interface belongs to Microsoft.Extensions.DependencyInjection namespace. So, whenever we inject any kind of method to this IServiceCollection object that means we are injecting the service to the built-in dependency injection.

**Configure Method:**

The Configure method is used to configure the HTTP request processing pipeline of the application. In other words, we can say that it will configure all the middleware that you want to use in your application. The Configure method takes two parameters i.e. IApplicationBuilder and IWebHostEnvironment instance. The IApplicationBuilder interface belongs to Microsoft.AspNetCore.Builder namespace and IWebHostEnvironment interface belong to Microsoft.AspNetCore.Hosting namespace.

The IApplicationBuilder Defines a class that provides the mechanisms to configure an application’s request pipeline. On the other hand, the IWebHostEnvironment Provides information about the web hosting environment an application is running in.

**Modifying Startup class:**

Please modify the Startup class as shown below. As you can see here, we just define the two methods ConfigureService and Configure and also include the respective namespaces.

**using** *Microsoft.AspNetCore.Builder;*

**using** *Microsoft.AspNetCore.Hosting;*

**using** *Microsoft.Extensions.DependencyInjection;*

**namespace** *ConsoleToWebAPI*

**{**

**public** **class** Startup

**{**

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

**{**

**}**

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

**{**

**}**

**}**

**}**

**Add Default Route:**

Once we define the ConfigureService and Configure methods, then we need to add the Route. Routing is basically a mechanism to access the resources by using a URI (Unique Resource Identifier). Each resource is uniquely identified. So, whenever you type the URL and press the enter button in the URL, then that request is mapped to a particular resource. In ASP.NET Core, if we want to add any kind of functionalities then we need to use Middleware. And as discussed, we need to configure the Middleware inside the Configure method of the Startup class.

So, we need to add a Middleware for Routing. By calling the UseRouting() method on the IApplicationBuilder object (i.e. app) we are configuring the Routing middleware in our application. By calling app.UseRouting() method we are just enabling the routing for our application. But we are not telling any kind of mapping i.e. when the request comes who is going to handle that request. Or in other words, we can say which resource is going to be executed when the request comes from the client.

So, we need to tell the mapping between a URL and a resource. And we can do the same by using the UseEndpoints middleware. Inside the UseEndpoints method, we need to call the MapGet method. And inside the MapGet method, we need to pass the URL. Along with we also need to set the content that we need to return. So, modify the Configure method as shown below.

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

**{**

app.UseRouting**()**;

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

**{**

endpoints.MapGet**(**"/", **async** context =**>** **{**

**await** context.Response.WriteAsync**(**"Hello From ASP.NET Core Web API"**)**;

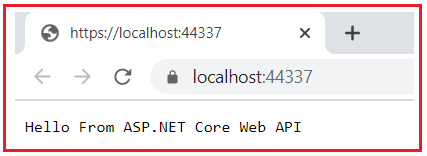
**})**;

**})**;

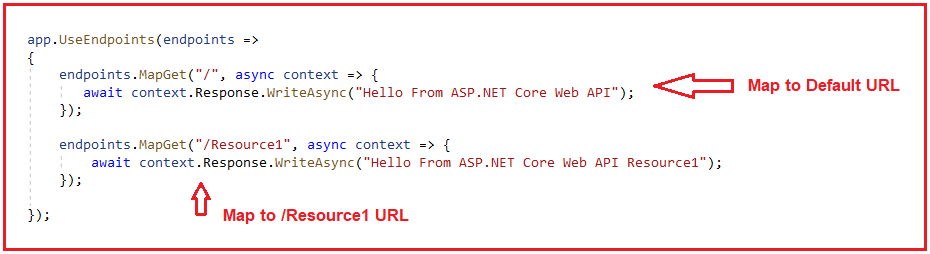
**}**

**Note:**Also include the using Microsoft.AspNetCore.Http namespace as we are using the WriteAsync method.

With the above changes in place, now run the application and you should the message on the browser as shown in the below image.



Let us add another endpoint to our application. Here, we are specifying the URL as /Resource1 as shown in the below image. With this, we have configured two URLs.



The Complete code of the Startup.cs class is given below.

**using** *Microsoft.AspNetCore.Builder;*

**using** *Microsoft.AspNetCore.Hosting;*

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

**using** *Microsoft.Extensions.DependencyInjection;*

**namespace** *ConsoleToWebAPI*

**{**

**public** **class** Startup

**{**

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

**{**

**}**

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

**{**

app.UseRouting**()**;

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

**{**

endpoints.MapGet**(**"/", **async** context =**>** **{**

**await** context.Response.WriteAsync**(**"Hello From ASP.NET Core Web API"**)**;

**})**;

endpoints.MapGet**(**"/Resource1", **async** context =**>** **{**

**await** context.Response.WriteAsync**(**"Hello From ASP.NET Core Web API Resource1"**)**;

**})**;

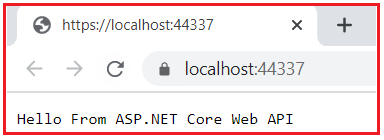
**})**;

**}**

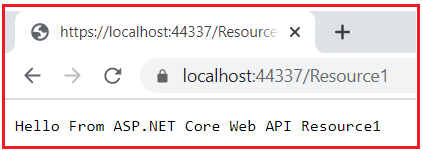
**}**

**}**

Now run the application and see the output. With the default URL, you will get the below output.



When we add /Resource1 to the URL, you will get the below output as expected.



This is what you will get when you create an empty ASP.NET Core Web Application. Again we need to add some features in order to make this application an ASP.NET Web API application

**How to use Web API Service?**

In ASP.NET Core, we can create different types of Web Applications such as MVC, Razor Pages, and Web API Applications. And for each application type, ASP.NET Core provided some set of services that should be included in the application.

So, the service that you want in your application should be injected in the ConfigureService method of the Startup class. We have different service like AddMvc(), AddControllers(), AddRazorPages(), and AddControllersWithViews() that we can inject into the dependency injection container. Now the question is which one we need to use or what are the differences between AddMvc, AddControllers, AddRazorPages, and AddControllersWithViews.

**Differences between AddMvc, AddControllers, AddRazorPages, and AddControllersWithViews:**

1. **AddControllers**: The AddControllers method is used only for Web APIs. That means if you want to create a Web API Application where there are no views, then you need to use the AddController extension method.
2. **AddControllersWithViews**: The AddControllersWithViews method is used to support MVC Based Applications. That means If you want to develop a Model View Controller (i.e. MVC) application then you need to use AddControllersWithViews() extension method.
3. **AddRazorPages**: The AddRazorPages method is used for the Razor Pages application. That means if you want to work with the Razor Page application, then you need to use the AddRazorPages() extension method
4. **AddMvc:**The AddMvc method adds all the services that are required for developing any type of application. That means we can say if we are using the AddMvc method, then we can develop Web API, MVC, and Razor Pages applications.

**Note:** Adding AddMvc() method will add extra features to your application even though they are not required which might impact the performance of the application.

**Adding Web API Service:**

As we are going to deal with ASP.NET Core Web API, so, we need to configure the **AddControllers** extension method to the built-in dependency injection (services) in the ConfigureService method of the Startup class. So, please modify the ConfigureService method of the Startup class as shown in the below code.

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

**{**

services.AddControllers**()**;

**}**

# Adding Controller in ASP.NET Core

## ****Adding Controller Class in ASP.NET Core****

In this article, I am going to discuss **How to Add a Controller to ASP.NET Core Web API Application**. This is the last step of converting ASP.NET Core Console Application to ASP.NET Core Web API Application. So, before proceeding to this article, please read [**Step1**](https://dotnettutorials.net/lesson/build-asp-net-core-web-api-project/), [**Step2**](https://dotnettutorials.net/lesson/adding-web-host-builder/), and [**Step3**](https://dotnettutorials.net/lesson/configuring-startup-class/) articles.

##### ****Adding Controller Class in ASP.NET Core Web API****

As of now, we have done the very basic configuration that is required to convert a console application to a Web API Application. Now we are going to discuss the most important concept of Web API i.e. Controllers.

###### **Points to Remember while working with Controller class in ASP.NET Core Web API:**

1. The Controller class in ASP.NET Core Web API must have a “Controller” suffix. For example, if you want to add a controller with the name Home, then the name of the controller should be HomeController. similarly, if you want to add a controller for Student, then the name should be StudentController.
2. The Controller class must be inherited from the ControllerBase class. If you are coming from an ASP.NET Core MVC background, then, in that case, the Controller class is inherited from the Controller class.
3. We need to decorate the controller class with the ApiController attribute.
4. We also need to use Attribute Routing to access the resource using URI.

##### ****What is the difference between the Controller and ControllerBase class in ASP.NET Core?****

The Controller class that we used in ASP.NET Core MVC has support Views, ViewBag, ViewData, TempData, etc. But here in ASP.NET Core Web API, we don’t require such concepts. So, we skip controller class and use the ControllerBase class.

The point that you need to remember is the ControllerBase class is also serves as the base class for the Controller class. That means behind the scene the Controller class is inherited from the ControllerBase class.

So, if you are using the Controller class in your ASP.NET Core MVC application, it means you are also using ControllerBase class. But in Web API, we simply need to use the ControllerBase class.

##### ****ControllerBase Class in ASP.NET Core Web API:****

The ControllerBase Class in ASP.NET Core provides many methods and properties to handle HTTP Requests and Responses. For example, if you want 200 Status codes from your action method, this provides one method. Similarly, if you want to return the 201 status code from your controller action method, it is provided with another method.

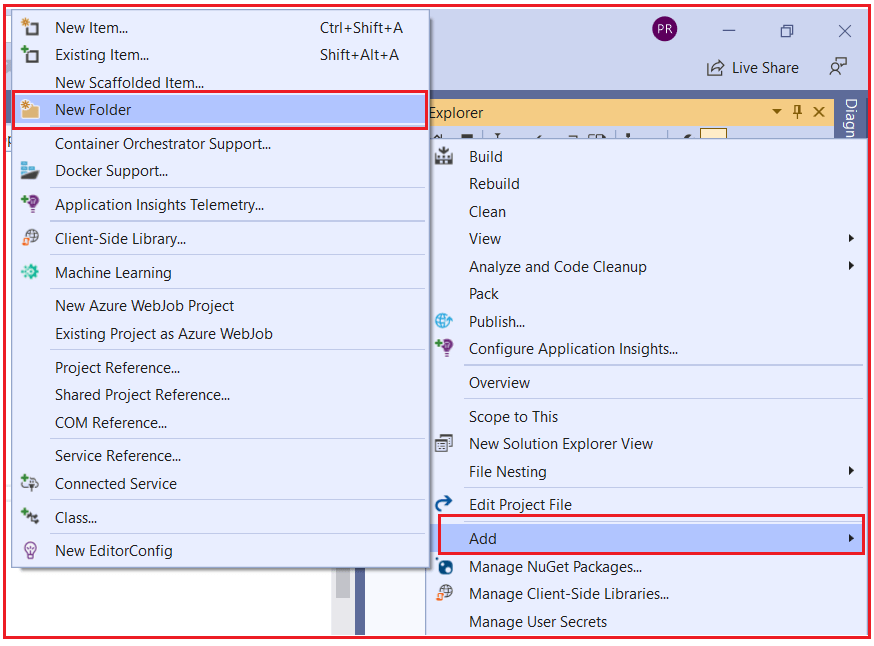
##### ****ApiController Attribute in ASP.NET Core Web API:****

It Indicates that a type and all derived types are used to serve HTTP API responses. Controllers decorated with this attribute are configured with features and behavior targeted at improving the developer experience for building APIs. When decorated on an assembly, all controllers in the assembly will be treated as controllers with API behavior. Some of the features are as follows:

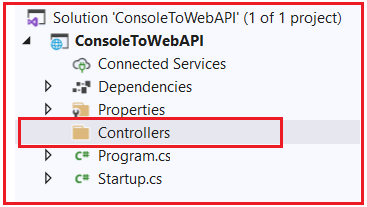
1. Attribute Routing Requirement
2. Handle the client error i.e. 400 HTTP status code.
3. Multipart/form-data request inference
4. Binding the incoming data with the parameters using the Model Binding concepts

##### ****Adding Controller in ASP.NET Core Web API:****

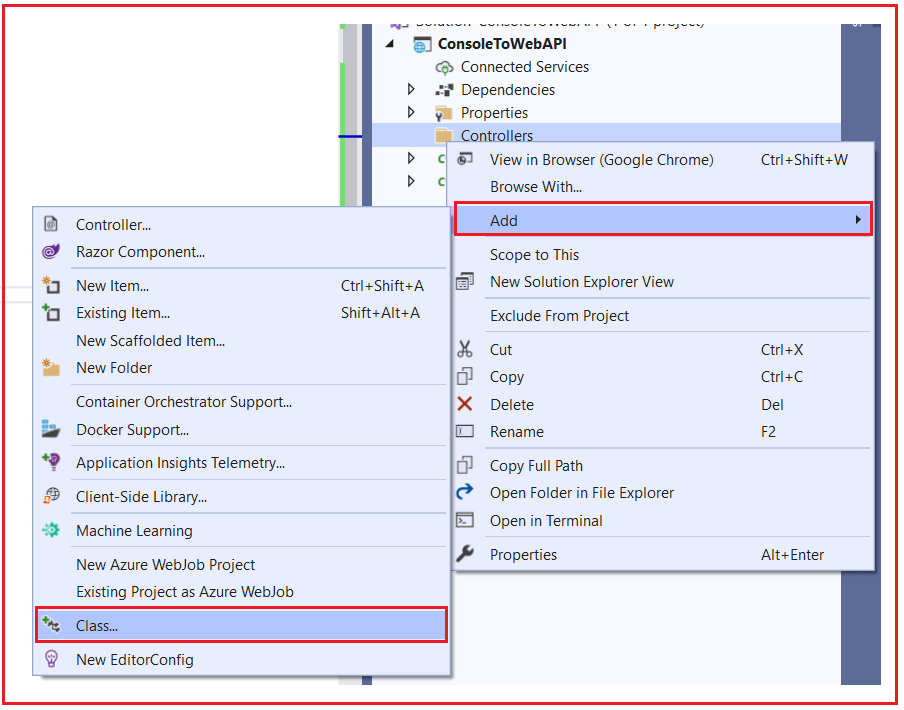
The Controller should be added inside the Controllers folder in your project. So, let us first add the Controllers folder to the project root directory. To do so, right-click on your project and then select Add => New Folder option from the context menu as shown in the below image.



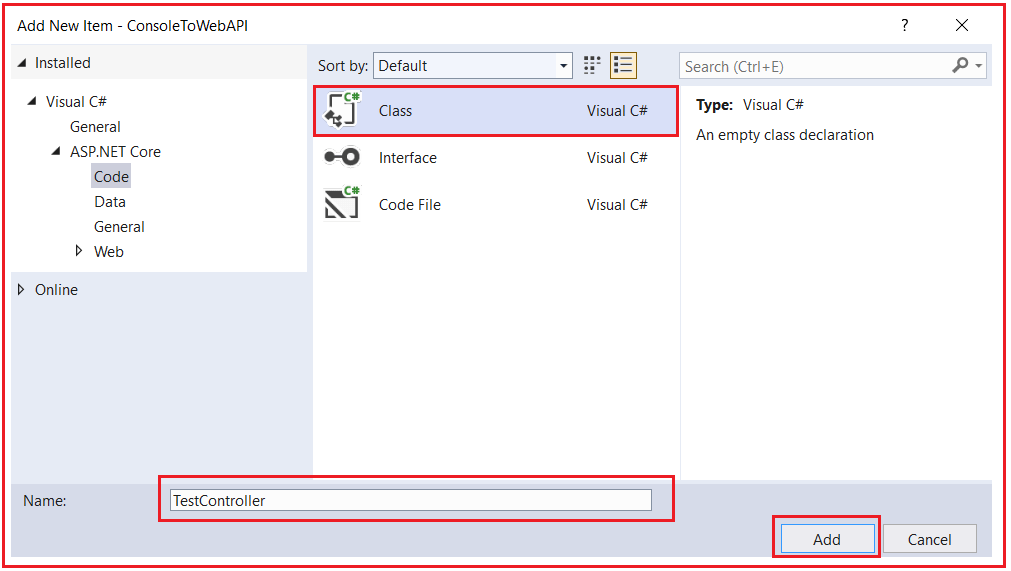
Then rename the folder as Controllers. Once the controller folder is added your project structure should looks as shown in the below image.



Let’s add a controller with the name Test. To do so, right-click on the Controllers folder and then select Add => Class from the context menu as shown in the below image.



From the next Add new item window, select class template and provide the class name as TestController and click on the Add button as shown in the below image which will add a simple class inside the Controllers folder.



##### ****Modifying the TestController class:****

First, we need to inherit the TestController class from the ControllerBase class and ControllerBase class present in Microsoft.AspNetCore.Mvc namespace. Then decorate the Controller with ApiController and Route attribute. A Controller class can have multiple action methods. Let add one action method which simply returns a string. So, modify the TestController class as shown in the below code.

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

**namespace** *ConsoleToWebAPI.Controllers*

**{**

**[**ApiController**]**

**[**Route**(**"test"**)]**

**public** **class** TestController : ControllerBase

**{**

**public** string Get**()**

**{**

**return** "Returning from TestController Get Method";

**}**

**}**

**}**

##### ****Mapping HTTP Request to Web API Controller:****

Now we need to tell the ASP.NET Core Framework to map the incoming HTTP Request to our Web API Controller. To do so, modify the Configure method of the Startup class as shown in the below code. Here, we are using the built-in MapControllers extension method to map the incoming HTTP Request to our Controllers.

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

**{**

app.UseRouting**()**;

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

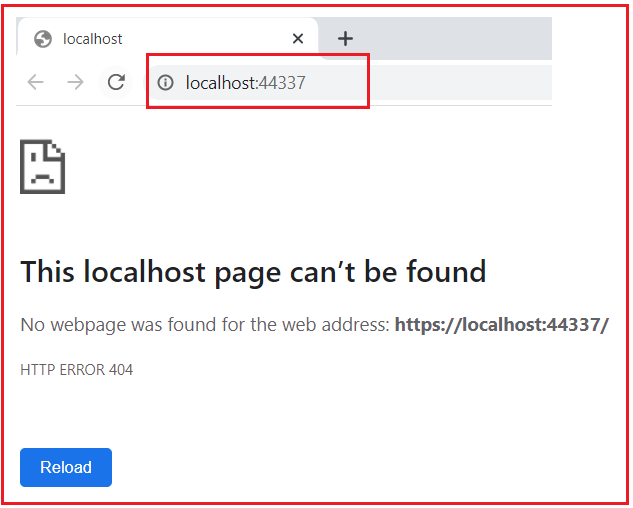
**{**

endpoints.MapControllers**()**;

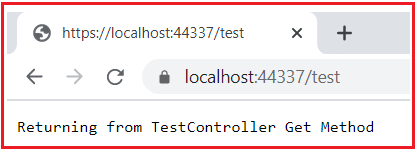
**})**;

**}**

With the above changes in place, now run the application and you will get the following 404 error page. This is because at this point there is no resource for this request



Let’s append /test at the end of the URL and you should get the data as expected as shown in the below image.



##### ****What happens if we add one more resource into our Controller class?****

Here resource means method. Let’s add one method into the Test Controller class as shown in the below code.

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

**namespace** *ConsoleToWebAPI.Controllers*

**{**

**[**ApiController**]**

**[**Route**(**"test"**)]**

**public** **class** TestController : ControllerBase

**{**

**public** string Get**()**

**{**

**return** "Returning from TestController Get Method";

**}**

**public** string Get2**()**

**{**

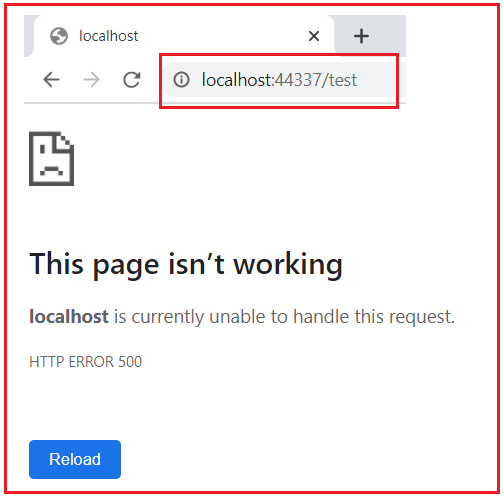
**return** "Returning from TestController Get2 Method";

**}**

**}**

**}**

At this point, if you run the application, then you will get the following.



As you can see in the above image, the actual exception details are not visible to you and the reason for this is we have not added the Developer Exception Page Middleware into the application request processing pipeline. Let us add the Developer Exception Page middleware into our request processing pipeline. So, modify the Configure method of the Startup class as shown in the below code. Here, we checking if the environment is Development, then only show the exception details. Please also include Microsoft.Extensions.Hosting namespace

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

**{**

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

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

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

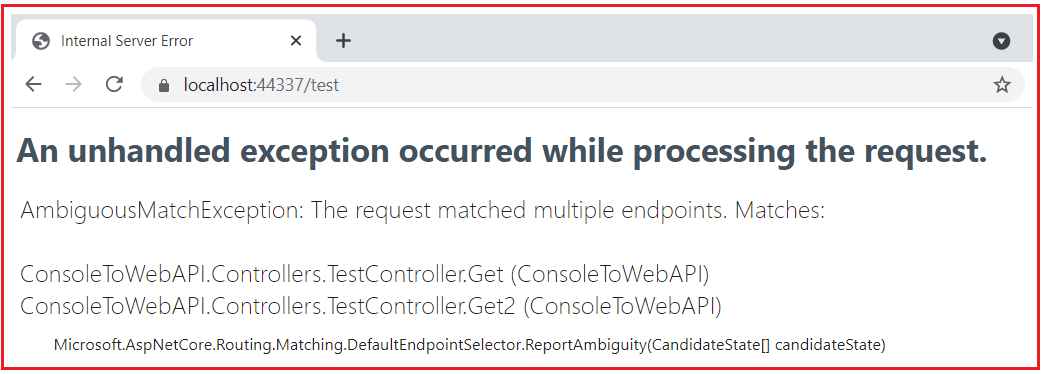
**{**

endpoints.MapControllers**()**;

**})**;

**}**

With the above changes in place, now run the application and navigates to /Test in the URL and you should see the exception details as shown in the below image.



As you can see in the above image, it clearly said that

**AmbiguousMatchException: The request matched multiple endpoints. Matches:**  
**ConsoleToWebAPI.Controllers.TestController.Get (ConsoleToWebAPI)**  
**ConsoleToWebAPI.Controllers.TestController.Get2 (ConsoleToWebAPI)**

So, the application finds two endpoints for the incoming request and gets confused about who is going to serve the request, and hence it gives the above exception.

As per Web API standard, each resource should have a Unique Identifier. Let us do some changes in our Route Attribute so that each request will have a unique URI. In the Route attribute, we specified the action as part of the URI. In our upcoming articles, we will discuss Routing in detail. As of now, just modify the TestController class as shown below.

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

**namespace** *ConsoleToWebAPI.Controllers*

**{**

**[**ApiController**]**

**[**Route**(**"test/{action}"**)]**

**public** **class** TestController : ControllerBase

**{**

**public** string Get**()**

**{**

**return** "Returning from TestController Get Method";

**}**

**public** string Get2**()**

**{**

**return** "Returning from TestController Get2 Method";

**}**

**}**

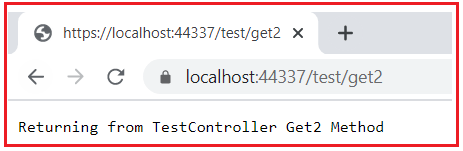
**}**

With the above changes in place, now run the application. Now we have to pass the method name as part of the URL as shown in the below image.

###### **Test/get**



###### **Test/Get2**



That’s it. We have done all the steps. Now we have successfully converted an ASP.NET Core Console Application to an ASP.NET Core Web API Application.