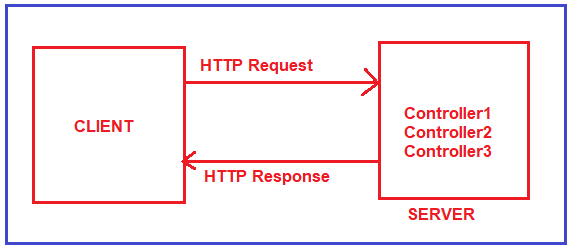
**Middleware in ASP.NET Core Web API**

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

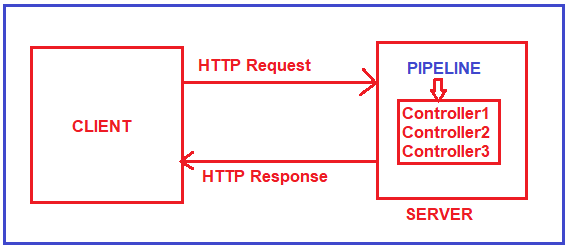
In this article, I am going to discuss **Middleware in ASP.NET Core Web API Application** with Examples. Please read our previous article where we discussed [**how to convert a console application to the ASP.NET Core Web API application**](https://dotnettutorials.net/lesson/build-asp-net-core-web-api-project/). Middleware is one of the core functionalities of any ASP.NET Core Web Application. When we send a request from the client to the server, middleware comes into the picture. So, at the end of this article, you will understand what exactly middleware and how it is used to handle the incoming HTTP Request and Response.

**HTTP Request Pipeline:**

Before understanding ASP.NET Core Middleware components, let us first understand what is HTTP Request Pipeline and how does it work. Please have a look at the following image for a better understanding of the HTTP Request Pipeline. As you can see in the below image, on the left-hand side, we have the client i.e. a browser and on the right-hand side, we have the server where our ASP.NET Core Web API application is hosted. Further, the Web API Application has three controllers. So, when the client sends a request to the server, we generally believe that it is the controller action method that is going to serve the request and then we get the response.



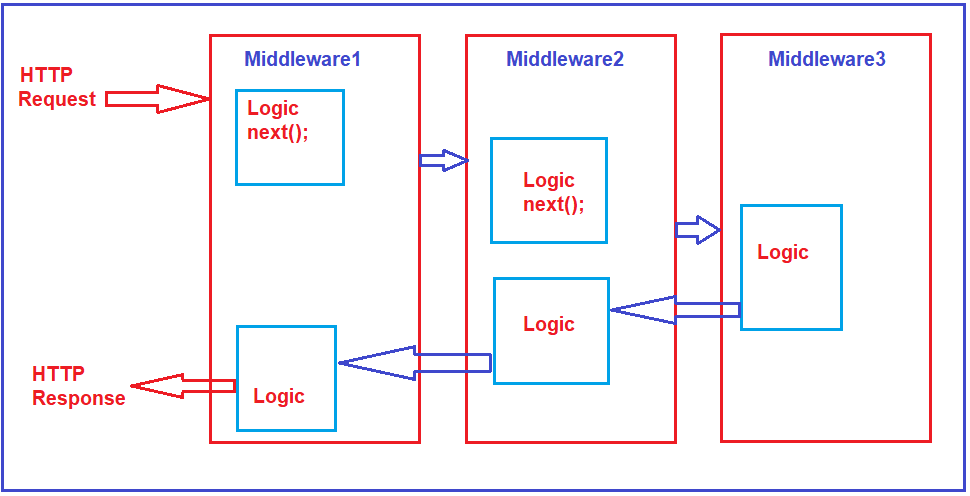
But in reality, before hitting the controller action method, the request has to pass through a pipeline. Once the pipeline is completed, then only it navigates the request to the corresponding controller action method as shown in the below image.



Let us understand what exactly a Pipeline is in detail.

**HTTP Request Pipeline in ASP.NET Core Web API Application:**

The Request Pipeline in ASP.NET Core Web API Application can have multiple middlewares as shown in the below image. If you are confusing what is Middleware, let us assume a middleware is a piece of code with some logic. Whenever a request comes from a client to the server, then the request comes to the first middleware which is registered in the request pipeline. In our case it is Middleware1. The code or logic which is there in Middleware1 will be executed and then if it will call the next method, then the request goes to the next middleware which is registered in the request processing pipeline i.e. Middlware2. The code or logic which is there in Middleware2 will be executed and if it calls the next method, then it navigates to the request to the next middleware i.e. Middleware3. Let us assume, in Middleware3, we don’t have the next method. So, the code or logic which is there in Middleware3 will be executed and as there is no next method call, so the request will come back to the previous middleware i.e. Middleware2. And if there is some code after the next method in Middleware2, then those codes will be executed and once the code executed the request again comes back to the previous middleware i.e. Middlwware1. Similarly, if there is some code after the next method in Middleware1, then those codes will be executed and once the code gets executed the final response sends to the client who initially made the request.



This is how the Request Pipeline works in ASP.NET Core Web API Application. These Middleware are nothing but a piece of code or we can say these are some functionalities that we want to insert in our ASP.NET Core Web API Application.

**Middleware in ASP.NET Core Web API:**

Middleware is a piece of code that is used in the HTTP Request Pipeline. An ASP.NET Core Web API Application can have n numbers of middleware. So, depending upon the requirement, we can configure n numbers of middleware in the application request processing pipeline.

The order of middleware matters a lot in the execution. That means in the order they are configured into the request processing pipeline; in the same order, they are going to be executed when a request comes. Each middleware in the ASP.NET Core Web API Application performs the following tasks.

1. Chooses whether to pass the HTTP Request to the next component in the pipeline. This can be achieved by calling the next() method within the middleware.
2. Can perform work before and after the next component in the pipeline.

ASP.NET Core provides some built-in middleware that is ready to be used, even if you want then you can also create your own custom middleware. The most important point that you need to keep in mind is, in ASP.NET Core a given Middleware component should only have a specific purpose i.e. single responsibility.

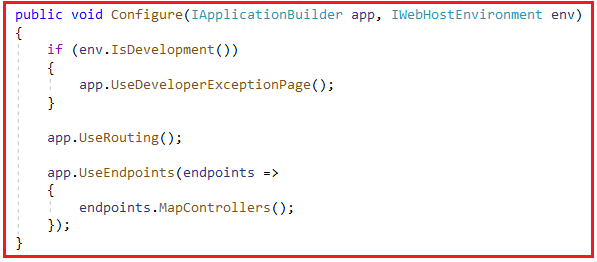
**Middleware Examples:**

1. **Routing**: If you want to implement Routing in your application, then you need to use Routing Middleware in the HTTP Request Processing pipeline.
2. **Authentication**: If you want to authenticate the user then you need to use Authentication Middleware.
3. **Authorize**: The Authorize Middleware is used to Authorize the users while accessing a specific resource.
4. **Log**: If you want to log request and response while processing, then you need Middleware.
5. **Exception Middleware:** You can also use Middleware to handle the exception globally.

**Note**: The Middleware in ASP.NET Core Web API Application is used to set up the HTTP Request processing pipeline. If you have prior experience of the previous .NET Framework then you may know, HTTP Handlers and HTTP Modules which are basically used to set up the request processing pipeline. It is this pipeline that will determine how the HTTP request and response are going to be processed.

**How to Configure Middleware Components in ASP.NET Core application?**

In the ASP.NET Core Web API application, the Middleware components are configured within the Configure method of the Startup class. The Startup class is the class that is going to run when the application starts. The following is the Configure method of the Startup class that we have created in our previous article. Even though if you created an ASP.NET Core Application with an Empty Project template, then also you will find the following code within the Configure method of the Startup class.



As you can see in the above image, within the Configure method we have configured three Middleware components to the HTTP Request Processing Pipeline. They are as follows.

1. **UseDeveloperExceptionPage() Middleware:**The UseDeveloperExceptionPage() middleware will come into picture only when the hosting environment is set to “development”. The UseDeveloperExceptionPage middleware is going to execute when there is an unhandled exception that occurred in the application and since it is in development mode, it is going to show you the detailed information of the exception.
2. **UseRouting() Middleware:**The UseRouting middleware is used to add Endpoint Routing Middleware to the request processing pipeline i.e. it will map the URL (or incoming HTTP Request) to a particular resource.
3. **UseEndpoints() Middleware:**In this middleware, the routing decisions are going to be taken using the Map extension method.

So, if you want to configure any middleware components in any type of ASP.NET Core applications, then you need to configure it within the Configure method by calling the **Use\*** methods on the IApplicationBuilderobject.

**Note:**The ASP.NET Core Middleware components are executed in the same order as they are added to the pipeline. So, it is our key responsibility to take care when adding middleware components.

**Run, Use, Next, and Map Methods in Middleware:**

In order to work with ASP.NET Core Middleware Components, we need to learn about few methods are as follows:

1. **Run() Method:** The [**Run() Extension Method**](https://dotnettutorials.net/lesson/run-method-in-asp-net-core/) is used to complete the Middleware Execution.
2. **Use() Method:** The Use() Extension Method is used to insert a new Middleware component to the Request Processing Pipeline.
3. **Next() Method:** The Next() Extension Method is used to call the next middleware component in the request processing pipeline.
4. **Map() Method:** The Map() Extension Method is used to map the Middleware to a specific URL.

**Run, Use, and Next Method in ASP.NET Core**

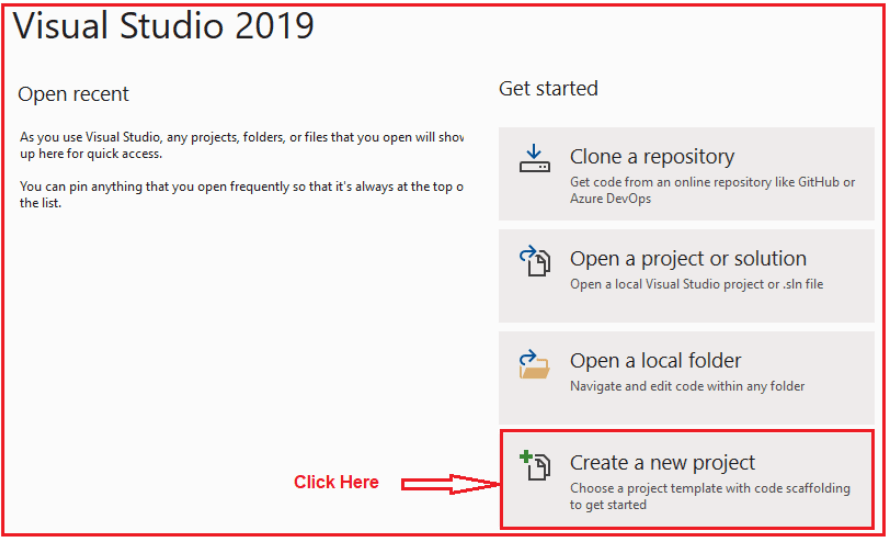
**Run, Use, and Next Extension Method in ASP.NET Core Application**

In this article, I am going to discuss the **Run, Use, and Next Extension Method in ASP.NET Core Web API** Application. Please read our previous article, where we discussed [**Middleware Components in ASP.NET Core Web API**](https://dotnettutorials.net/lesson/middleware-in-asp-net-core-web-api/)Application.

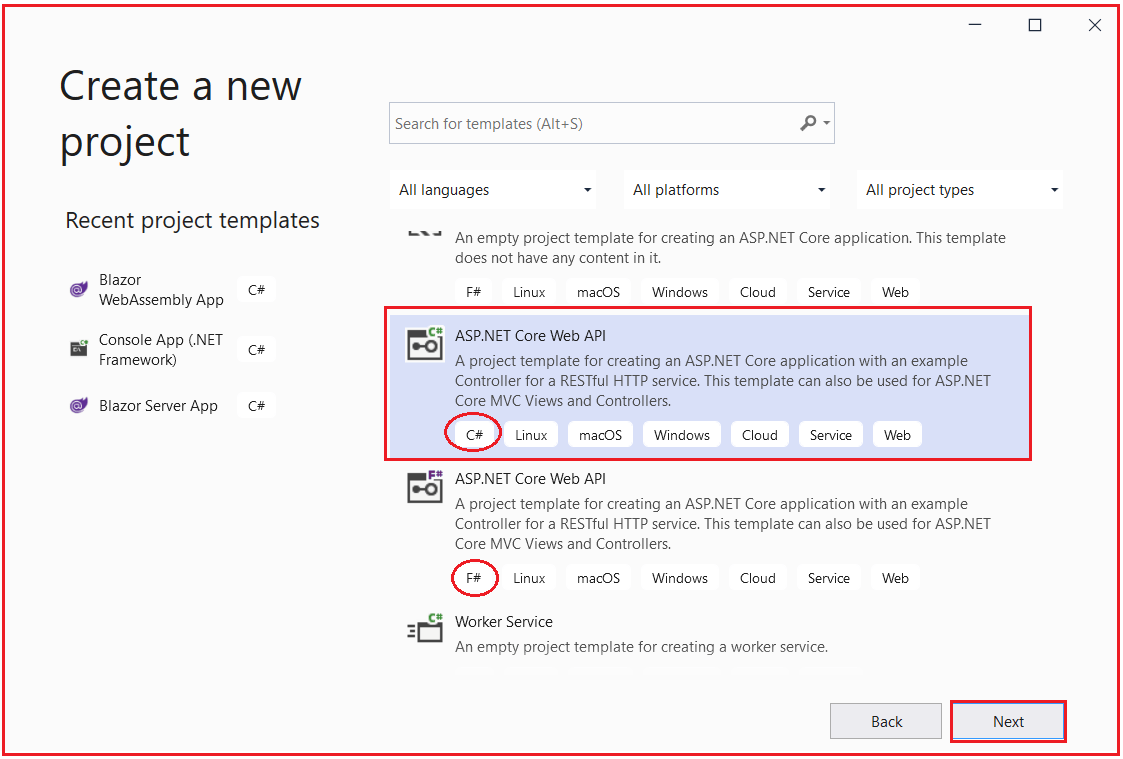
**Example to understand Run Method in ASP.NET Core:**

In order to understand the need and use of the Run, Use, and Next Extension method, please create a new empty ASP.NET Core Web Application by following the below steps.

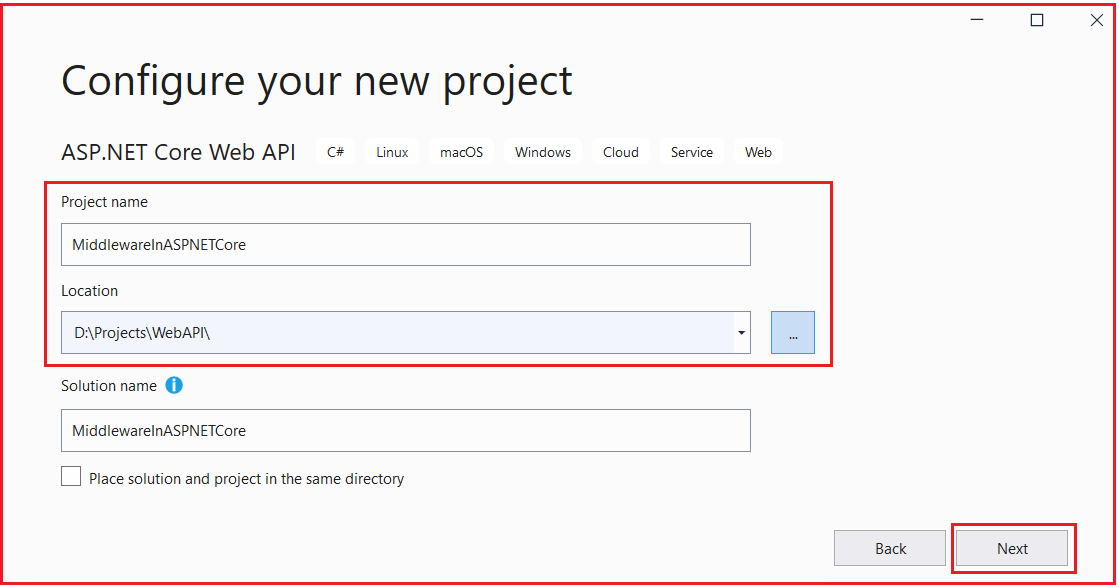
First, open Visual Studio 2019 and then click on the Create a new project option as shown in the below image.



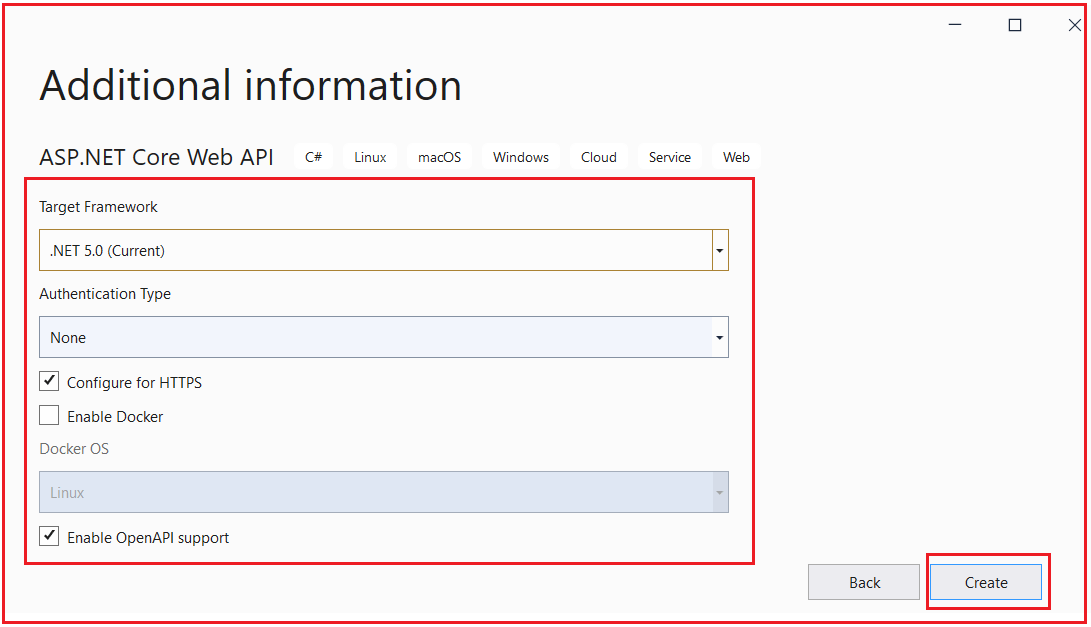
Once you click on the Create a new project option, it will open Create a new project window. Here, you can find two projects template for creating an **ASP.NET Core Web API**project. One is using C# language and the other one is using F# language. I am going to use C# as the programming language, so I select the project template which uses C# Language as shown in the below image.



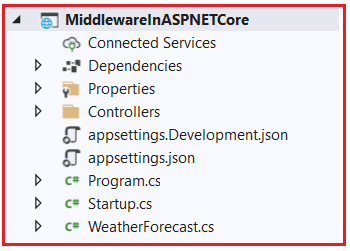
Once you click on the Next button, it will open configure your new project window. Here, you need to specify the Project name (MiddlewareInASPNETCore) and the location where you want to create the project. And then click on the **Next**button as shown in the below image.



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.



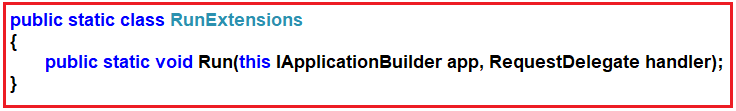
Once you click on the Create button, it will create the ASP.NET Core Web API project with the following file and folder structure.



**Run Method in ASP.NET Core**

The Run method in ASP.NET Core Application is used to complete the Middleware Execution. That means the Run extension method allows us to add the terminating middleware component. Terminating middleware means the middleware which will not call the next middleware components in the request processing pipeline.

The Run method is an extension method on the IApplicationBuilder interface and accepts a parameter of RequestDelegate delegate type which actually handles the request. Please have a look at the following image which shows the signature of the Run Extension method.

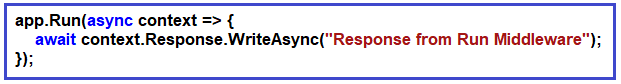


The Run Extension method is used for adding terminal middleware that means Adds a terminal middleware delegate to the application’s request pipeline. This method takes two Parameters:

1. **app**: The Microsoft.AspNetCore.Builder.IApplicationBuilder instance.
2. **handler**: A delegate that handles the request.

**Modifying the Configure Method of Startup class:**

Let us create a middleware using the Run method. As we already discussed if want to add middleware, then we need to add the same inside the Configure method of the Startup class. Let us add the following Middleware to the Request processing Pipeline.



As you can see in the above image, it is a simple middleware created using the Run method. The ASP.NET Core works on async mode, so we need to write the entire code in async in the Run method. Please include Microsoft.AspNetCore.Http namespace for async.

**Modify the Configure method of the Startup class as shown below.**

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

**{**

app.Run**(async** context =**>** **{**

**await** context.Response.WriteAsync**(**"Response from Run Middleware"**)**;

**})**;

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

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

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

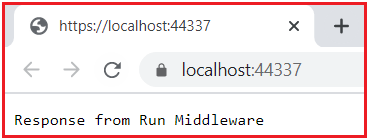
**{**

endpoints.MapControllers**()**;

**})**;

**}**

If you notice in the above code, the Run middleware component is configured as the first middleware component. At this moment if you run the application, then you will get the message in the browser which is coming from the Run method as shown in the below image.



As we already discussed the Run method is going to add terminating middleware i.e. it is not going to call the other middleware component configured in the request processing pipeline. To understand this concept, let us add another middleware to the request processing pipeline. So, please modify the Configure method of the Startup class as shown below.

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

**{**

app.Run**(async** context =**>** **{**

**await** context.Response.WriteAsync**(**"Response from First Run Middleware"**)**;

**})**;

app.Run**(async** context =**>** **{**

**await** context.Response.WriteAsync**(**"Response from Second Run Middleware"**)**;

**})**;

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

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

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

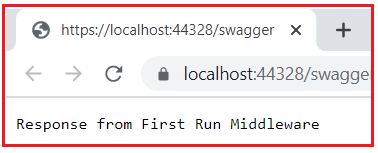
**{**

endpoints.MapControllers**()**;

**})**;

**}**

As you can see in the above code, the first two middleware components are configured using the Run method. When you run the application, it is the first middleware always going to serve the request and as it is configured using the Run method, so it is not going to call the next middleware component configured in the request processing pipeline. So, when you run the application, you will get the following response.

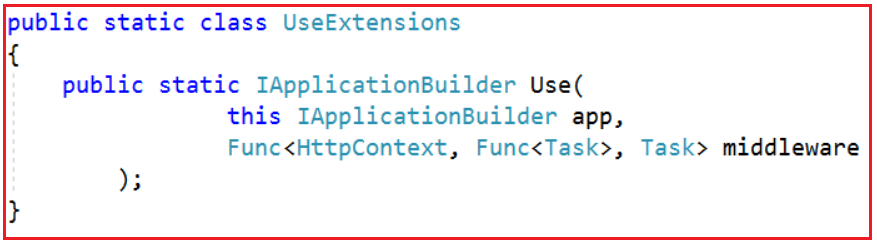


**Use() and Next() Extension Methods In ASP.NET Core**

The Use Extension Method in ASP.NET Core is used to add a new Middleware component to the Request Processing Pipeline whereas the Next Extension Method in ASP.NET Core is used to call the next middleware component configured in the request processing pipeline.

**Use Method:**

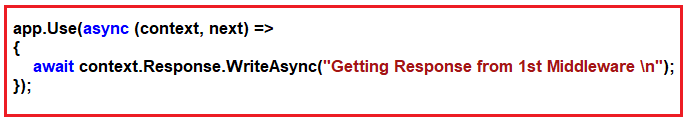
The Use Extension Method in ASP.NET Core Application allows us to add a new middleware component which may call the next middleware component in the request processing pipeline. The Use extension method adds a middleware delegate defined in-line to the application’s request pipeline. The Use method is also implemented as an extension method on the IApplicationBuilder interface. Following is the signature of the Use extension method:



As you can see in the above image, the Use Extension method takes two input parameters. The first parameter is the HttpContext context object using which it accesses both the HTTP request and response. The second parameter is the Func type i.e. it is a generic delegate that can handle the request or call the next middleware component in the request pipeline.

**Example to Understand Use Extension Method in ASP.NET Core:**

Let us add the following middleware component to the request processing pipeline.



With the above Middleware component in place, now our Configure method looks like below.

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

**{**

app.Use**(async** **(**context, next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Getting Response from 1st Middleware \n"**)**;

**})**;

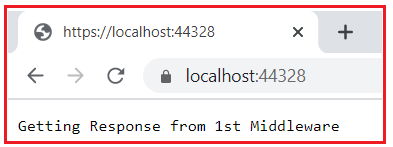
app.Run**(async** context =**>** **{**

**await** context.Response.WriteAsync**(**"Response Response from second Middleware \n"**)**;

**})**;

**}**

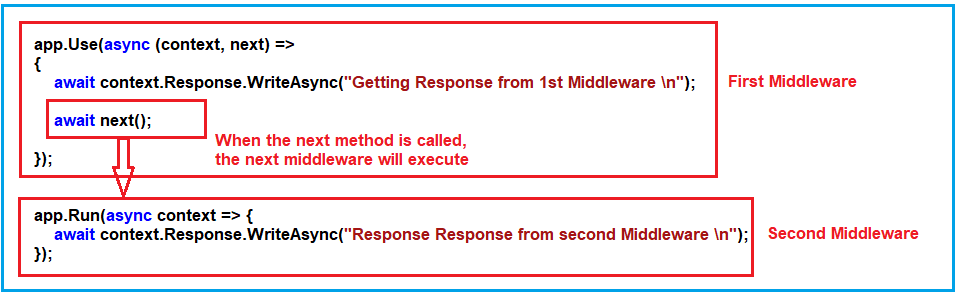
As you can see in the above code, the first middleware component is registered using the Use method while the second middleware component is registered using the Run method. With the above changes in place, now run the application and you should get the following output.



As you can see in the above image, the response is coming from the first middleware component i.e. the Middleware registered using the Use extension method. The next Middleware component which is registered using the Run method is not invoked.

**Example to understand Next Extension Method in ASP.NET Core**

Now the question that should come to your mind is, how to invoke the second middleware component? The answer is by calling the next method from the first middleware component as shown in the below image.



So, modify the Configure method of the Startup class as shown below.

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

**{**

app.Use**(async** **(**context, next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Getting Response from 1st Middleware \n"**)**;

**await** next**()**;

**})**;

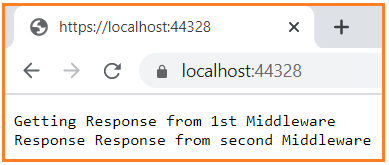
app.Run**(async** context =**>** **{**

**await** context.Response.WriteAsync**(**"Response Response from second Middleware \n"**)**;

**})**;

**}**

With the above changes in place, now run the application and you should get the response from both the middleware as shown in the below image.



If you write some code after the next method, then those code is going to be executed at the returning time. For better understanding, please have a look at the following image.

**Understanding Execution Order of Middleware in ASP.NET Core**

Let us understand the Request Processing Pipeline in detail with an example. First, modify the Configure method of the Startup class as shown below. As you can see in the below code, here, we are registering three middleware components. The first two middleware components are registered using the **Use()** Extension method and they are calling the next middleware component using the next method. The last Middleware component is registered using the **Run()** Extension method which is nothing but the terminating middleware component.

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

**{**

app.Use**(async** **(**context, next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Use Middleware1 Incoming Request \n"**)**;

**await** next**()**;

**await** context.Response.WriteAsync**(**"Use Middleware1 Outgoing Response \n"**)**;

**})**;

app.Use**(async** **(**context, next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Use Middleware2 Incoming Request \n"**)**;

**await** next**()**;

**await** context.Response.WriteAsync**(**"Use Middleware2 Outgoing Response \n"**)**;

**})**;

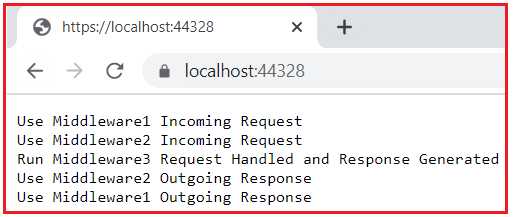
app.Run**(async** context =**>** **{**

**await** context.Response.WriteAsync**(**"Run Middleware3 Request Handled and Response Generated\n"**)**;

**})**;

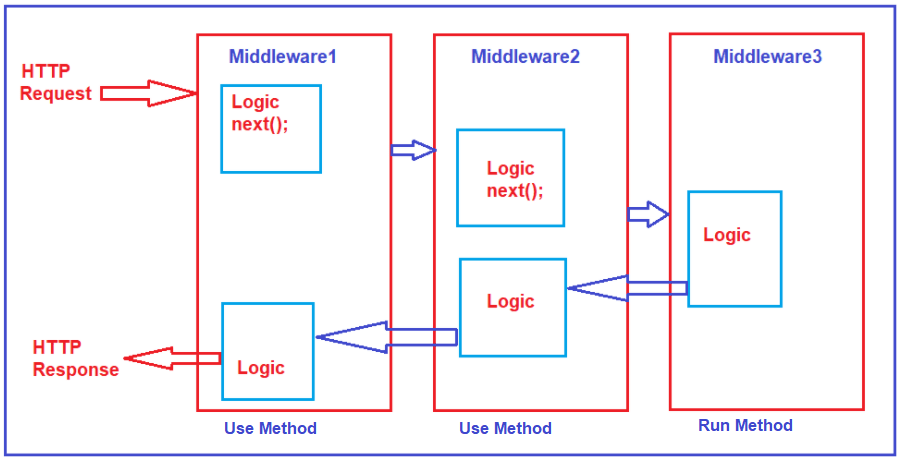
**}**

Further, if you notice in the above code, we have written some code before and after the next method call in the first two middleware components. Now save the change and run the application and you should get the following response in the browser.



**Understanding ASP.NET Core Request Processing Pipeline Execution Order:**

In order to understand the request processing pipeline execution order in ASP.NET Core Application, let us compare the above output with the following diagram which we have created in our Middleware in ASP.NET Core article.



When the incoming HTTP request comes, it is the first middleware that is registered in configure method that will receive the request i.e. Middleware1 which logs the message “**Use Middleware1 Incoming Request**” in the response stream. So as a result, we see this message first on the browser. Once the first middleware component logs the message to the response stream, then it calls the next() extension method which will invoke the next middleware component configured in the request processing pipeline i.e. Middleware 2 in our example.

The second middleware then logs the message “**Use Middleware2 Incoming Request**” into the output stream and that what you can see as the second message in the browser. Once it logs the message, then again it calls the next method, which will call the next middleware component registered in the request processing pipeline, and in our example, it is the third middleware component that is registered using the Run extension method.

The third middleware handles the request and then produces the response. So, the third information logs the third message in the browser, and that you can see “**Run Middleware3 Request Handled and Response Generated**”. As this middleware component is registered using the Run() extension method, so it is going to be a terminal middleware. That is it is not going to call the next middleware component.

So, from this point, the request pipeline starts reversing. That means from Middleware3, the execution control is given back to the previous middleware i.e. Middleware2, and the second middleware will check is there any code after the next method, if yes, then those code gets executed. In our example, it logs the message as “**Use Middleware2 Outgoing Response**” to the response stream and that you can see as the fourth message in the browser.

Again, once the second middleware complete its execution, the control back to the previous middleware i.e. Middleware1 and the Middleware1 will check is there any code after the next method call and if yes, then those code gets executed. In our example, it logs the message “**Use Middleware1 Outgoing Response”**to the response stream, and that what you can see as the last message in the browser.

This is how the Use, Next, and Run Method works in ASP.NET Core Application. In the next article, I am going to discuss the [**Map Extension Method in ASP.NET Core**](https://dotnettutorials.net/lesson/map-method-in-asp-net-core/) Web API Application. Here, in this article, I try to explain the need and use of the **Run, Use, and Next Extension Method in ASP.NET Core** Application and I hope you enjoy this Run, Use, and Next Extension Method in the ASP.NET Core Web API Application article.

# Map Method in ASP.NET Core

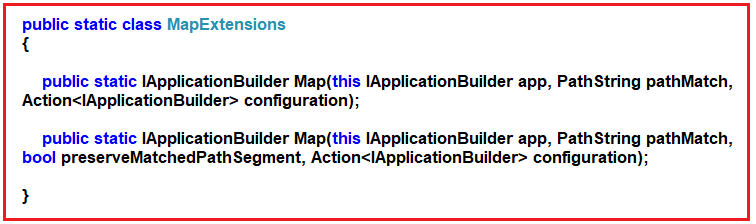
Back to: [ASP.NET Core Web API Tutorials](https://dotnettutorials.net/course/asp-net-core-web-api-tutorials/)

## ****Map Extension Method in ASP.NET Core Application****

In this article, I am going to discuss how to work with the **Map Extension Method in ASP.NET Core** HTTP Request Processing Pipeline. Please read our previous article where we discussed [**how to work with the Run, Next, and Use Extension methods in ASP.NET Core**](https://dotnettutorials.net/lesson/run-next-use-methods-in-asp-net-core/) Application. We are also going to work with the same application that we created in our previous article.

##### ****Map Extension Method in ASP.NET Core****

If you want to insert some specific middleware logic for some specific URL, then you can do the same using the Map Extension Method in any type of ASP.NET Core Application. Before using the Map Method, let us first have a look at the definition of this method which is shown in the below image. There are two overloaded versions available for this method in ASP.NET Core.



The Map method Branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed else the Middleware simply ignored. The Map Method takes the following Parameters:

1. **app**: The Microsoft.AspNetCore.Builder.IApplicationBuilder instance.
2. **pathMatch**: The request path to match.
3. **configuration**: The branch to take for positive path matches.
4. **preserveMatchedPathSegment**: if false, matched path would be removed from Request.Path and added to Request.PathBase.

**Returns**: The Microsoft.AspNetCore.Builder.IApplicationBuilder instance.

##### ****Example to Understand the Map Extension Method in ASP.NET Core:****

Let us understand the Map Extension Method with Examples. First, add the following method in the Startup class. This is the custom logic that we want to execute for a specific URL.

**private** **void** MapCustomMiddleware**(**IApplicationBuilder app**)**

**{**

app.Use**(async** **(**context, next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Specific URL Logic Middleware \n"**)**;

**})**;

**}**

Now, we need to register the Middleware component using the Map Extension method. As you can see here, we have provided the path as “/testmap” and also provided the second parameter as **MapCustomMiddleware**method that we just created.

**app.Map(“/testmap”, MapCustomMiddleware);**

With this when the request comes, if it contains **/testmap**as part of the URL, then only this Middleware component going to be executed else simply is ignored.

###### **Complete code of Startup class:**

The following is the complete code of the Startup class.

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

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

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

**using** *Microsoft.Extensions.Configuration;*

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

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

**namespace** *MiddlewareInASPNETCore*

**{**

**public** **class** Startup

**{**

**public** Startup**(**IConfiguration configuration**)**

**{**

Configuration = configuration;

**}**

**public** IConfiguration Configuration **{** **get**; **}**

// This method gets called by the runtime. Use this method to add services to the container.

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

**{**

services.AddControllers**()**;

**}**

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

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

**{**

app.Use**(async** **(**context, next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Use Middleware Component \n"**)**;

**await** next**()**;

**})**;

app.Map**(**"/testmap", MapCustomMiddleware**)**;

app.Run**(async** context =**>** **{**

**await** context.Response.WriteAsync**(**"Run Middleware Component\n"**)**;

**})**;

**}**

**private** **void** MapCustomMiddleware**(**IApplicationBuilder app**)**

**{**

app.Use**(async** **(**context, next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Specific URL Logic Middleware using Map Method \n"**)**;

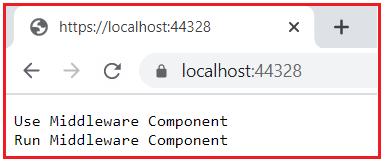
**})**;

**}**

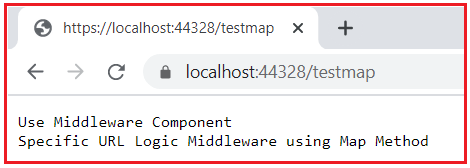
**}**

**}**

Now, save the changes and run the application. First, check without the custom path as shown in the below image.



As you can see in the above image, the Middleware which is configured using the Map Extension method will not be executed. This is because the incoming request URL does not include the path **/testmap**as part of the URL. Now, modify the URL to include **/testmap** as part of the URL and see the response as shown in the below image.



Now, you can see in the above, first, the first middleware executed and then the second middleware executed which is specifically designed for this request.

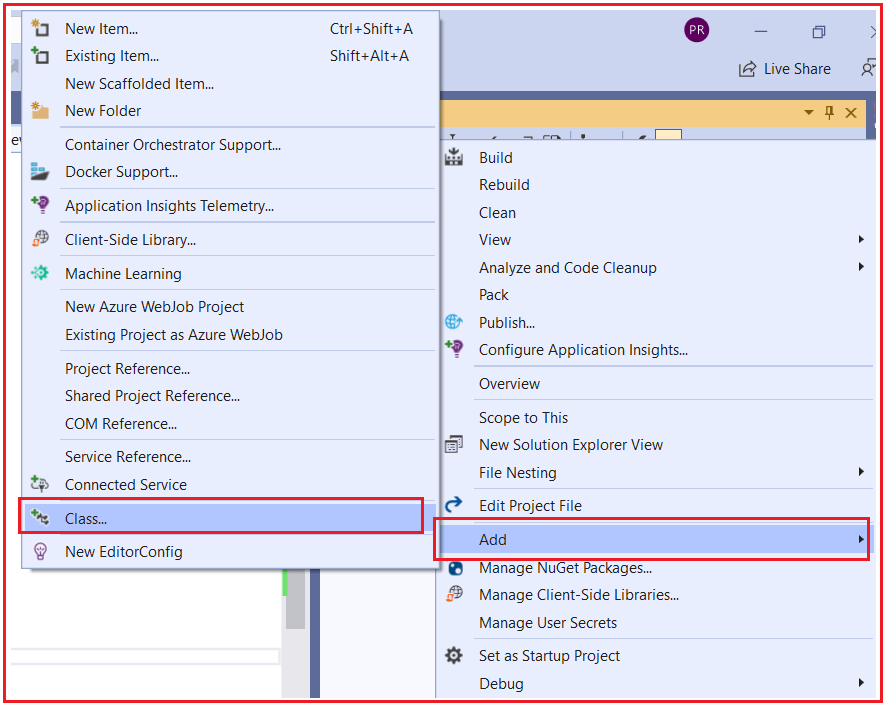
**Custom Middleware in ASP.NET Core**

**Custom Middleware in ASP.NET Core**

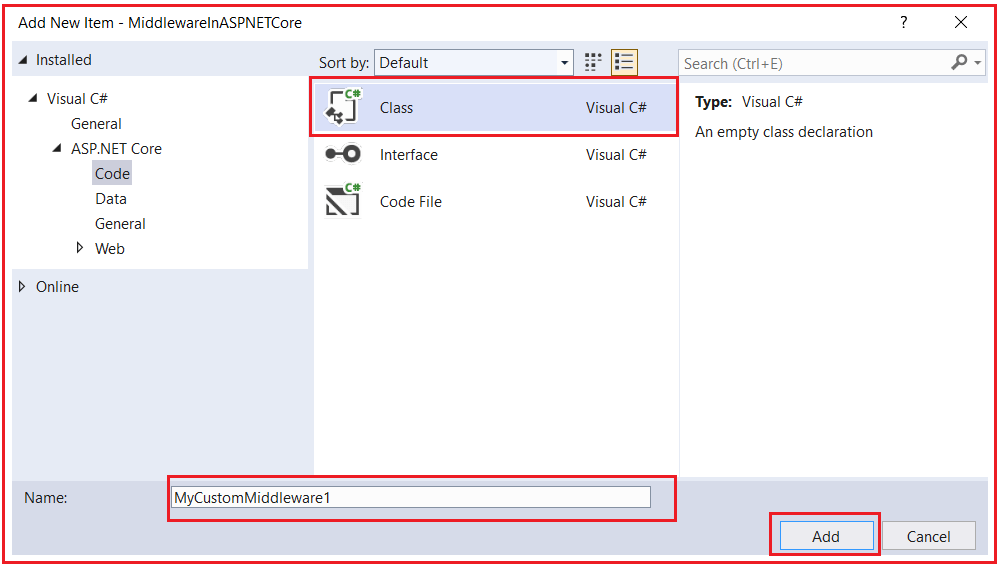
In this article, I am going to discuss **How to Create, Register, and use Custom Middleware Components in ASP.NET Core** Application. Please read our previous article where we discussed [**Run. Next, Map, and Use Extension methods in ASP.NET Core**](https://dotnettutorials.net/lesson/run-next-use-methods-in-asp-net-core/) with examples. In fact, we are also going to work with the same example that we created in our Run, Next, and Use Extension Methods article.

**Creating Custom Middleware in ASP.NET Core**

While working with the real-time applications in ASP.NET Core Web API, it is a common requirement to create Custom Middleware Components. So, let us add a new class file to our project. It is this class file that is going to contain the logic. So, right-click on the project name and then select add => class as shown in the below image.



From the next add new class screen, select the class template and provide the class name as MyCustomMiddleware1 and then click on the Add button as shown in the below image.



In order to make a class a Middleware component, the class needs to be inherited from the IMiddleware interface. Further IMiddleware interface belongs to Microsoft.AspNetCore.Http namespace. And we need to implement the InvokeAsync method. And you need to write your logic within the InvokeAsync method. So, modify the MyCustomMiddleware1 class as shown below.

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

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

**namespace** *MiddlewareInASPNETCore*

**{**

**public** **class** MyCustomMiddleware1 : IMiddleware

**{**

**public** **async** Task InvokeAsync**(**HttpContext context, RequestDelegate next**)**

**{**

**await** context.Response.WriteAsync**(**"Custom Middleware Incoming Request \n"**)**;

**await** next**(**context**)**;

**await** context.Response.WriteAsync**(**"Custom Middleware Outgoing Response \n"**)**;

**}**

**}**

**}**

**Note:**While calling the next method from any custom middleware components, we need to pass the context object and that you can see in the above code.

Our Custom Middleware component is ready. Now we need to use it in our HTTP Request Processing pipeline. Now it is a two-step process to use this custom middleware component.

**Step1: Inject the service to the built-in dependency injection container**

Remember if you want to use any custom service, before using it, you must inject the service into the built-in IoC Container. You can inject the service using the ConfigureService method of the Startup class as **services.AddTransient<MyCustomMiddleware1>();**. We will discuss AddTransient and its working in detail while we discuss dependency injection. For now, just we this method to configure the custom services. So, modify the ConfigureService method of the Startup class as shown below.

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

**{**

services.AddControllers**()**;

services.AddTransient**<**MyCustomMiddleware1**>()**;

**}**

**Step2: Registering the Custom Middleware in the HTTP Request Processing Pipeline**

Once you configure the service to the built-in IoC Container, then you can use the Middleware. As we already discussed we can configure the Middleware to the Request Processing Pipeline using the Configure method of the Startup class. To configure the custom Middleware we need to use the UseMiddleware Extension method as **app.UseMiddleware<MyCustomMiddleware1>()**; So, modify the Configure method of the Startup class as shown below.

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

**{**

app.Use**(async** **(**context, next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Use Middleware Incoming Request \n"**)**;

**await** next**()**;

**await** context.Response.WriteAsync**(**"Use Middleware Outgoing Response \n"**)**;

**})**;

app.UseMiddleware**<**MyCustomMiddleware1**>()**;

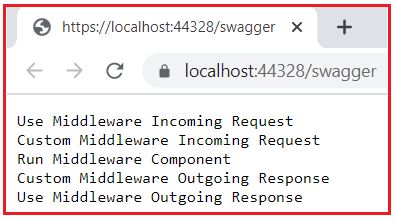
app.Run**(async** context =**>** **{**

**await** context.Response.WriteAsync**(**"Run Middleware Component\n"**)**;

**})**;

**}**

Now save the changes and run the application and you should get the following response in the browser as expected.



**Points to Remember while working with ASP.NET Core Middleware:**

1. The ASP.NET Core Request Processing Pipeline consists of a sequence of middleware components (custom plus built-in) that are going to be called one after the other. If we want to call the next middleware components then we need to use the next method.
2. Each middleware component in ASP.NET Core Application can perform some operations before and after calling the next component using the next method. A middleware component in ASP.NET Core Application can also decide not to call the next middleware component which is called short-circuiting the request pipeline.
3. The ASP.NET Core middleware component can access both the incoming request and the outgoing response.
4. The Run method in ASP.NET Core is the terminating Middleware Component which means it is not possible to call the next middleware component.
5. The order in which the middleware components are registered in the Configure method defines the order in which these middleware components are going to be invoked on requests and the reverse order for the response. So, the order is critical for defining the security, performance, and functionality of the application.