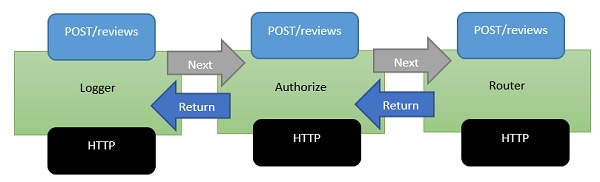
**ASP.NET Core – Middleware**

Middleware are software components that are assembled into an application pipeline to handle requests and responses.

Each component chooses whether to pass the request on to the next component in the pipeline, and can perform certain actions before and after the next component is invoked in the pipeline.

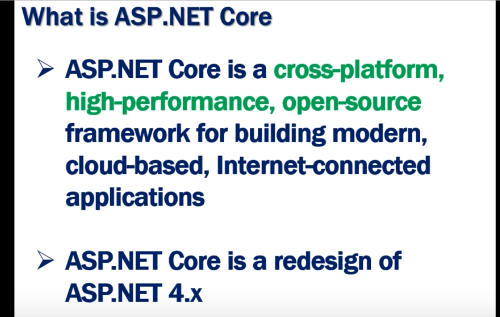
Let us now assume that we want to log information about every request into our application.

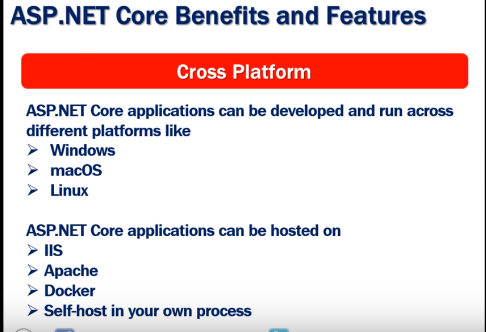
* In that case, the first piece of middleware that we might install into the application is a logging component.
* This logger can see everything about the incoming request, but chances are a logger is simply going to record some information and then pass along this request to the next piece of middleware.



* Middleware is a series of components present in this processing pipeline.
* The next piece of middleware that we've installed into the application is an authorizer.
* An authorizer might be looking for specific cookie or access tokens in the HTTP headers.
* If the authorizer finds a token, it allows the request to proceed. If not, perhaps the authorizer itself will respond to the request with an HTTP error code or redirect code to send the user to a login page.
* But, otherwise, the authorizer will pass the request to the next piece of middleware which is a router.
* A router looks at the URL and determines your next step of action.
* The router looks over the application for something to respond to and if the router doesn't find anything to respond to, the router itself might return a **404 Not Found error**.

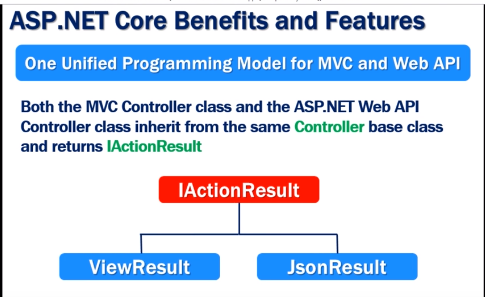
**What is ASP.Net Core**





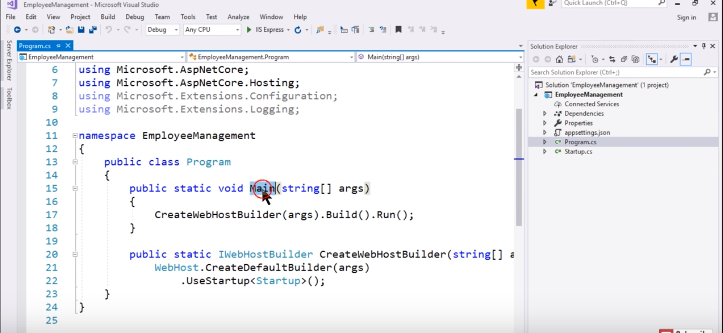
-In .Net framework application can be develop and run in Windows only.

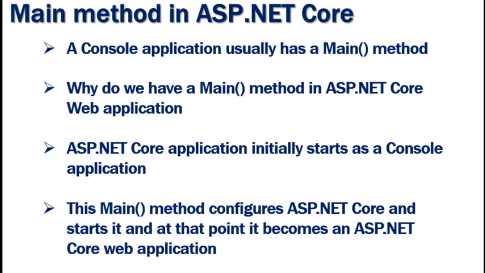
-In .Net framework we can host the application in IIS only.



.Net Core support built-in dependency injection.

**Main Method in Asp.net Core**

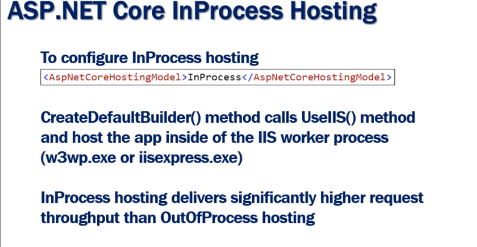




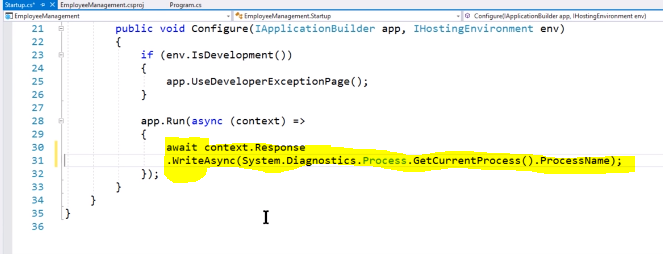
Main method called the CreateWebHostBuilder () which return the object that implement the IWebHostBuilder.The object return by the CreateDefaultBuilder () called the build() method which build the web host that host the asp.net core application and that web host call the Run() method and start the incoming requesting.





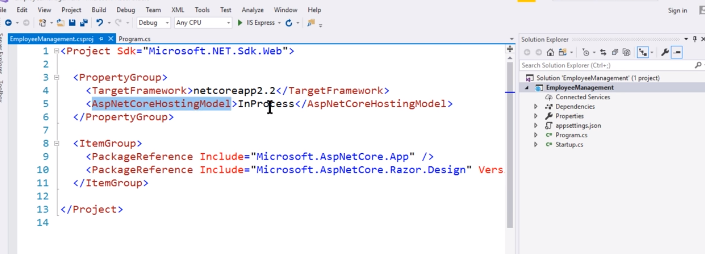


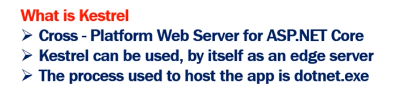


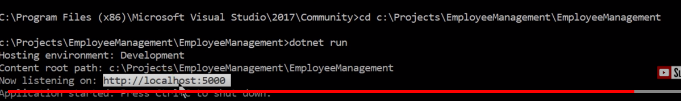


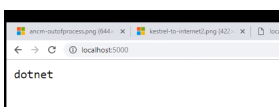
Code for check the server name like:-IISExpress,dotnet(Kestral) etc

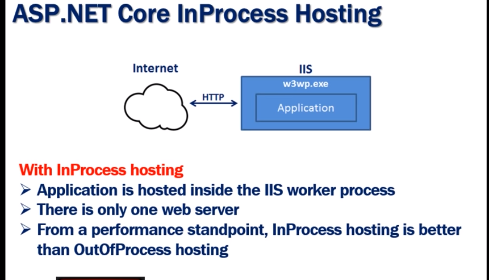
By default Asp.Net core application are hosted in InProcess (IIS Express).By default means when we create Asp.Net Core empty project below are the attribute are element are adding in project template.







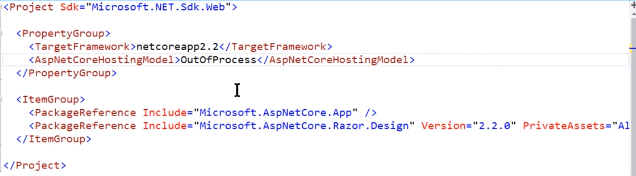




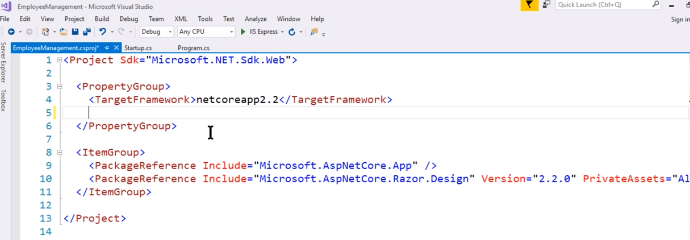
**Out of Process hosting:**

We have a two ways for out of hosting.

1-First way change the <AspNetCoreHostingModel> element to OutOfProcessing.



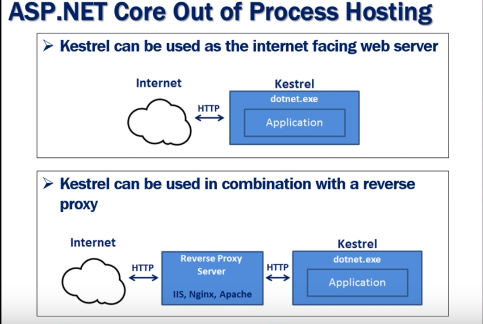
2-Remove the <AspNetCoreHostingModel> element. If we remove <AspNetCoreHostingModel> element then by default is OutOfProcess Hosting.



In Case of Out Of Process we have two web server internal and external.

Internal- Kestrel

External-IIS, Apache



Note-If we are running the Asp.net core application using the cli in that case it’s ignore the project file setting means it’s ignore the <AspNetCoreHostingModel> element and by default it’s set to out of process so it use the internal web server kestrel.

### Why Use a Reverse Proxy

A reverse proxy can be used in different contexts:

* Load balancing:

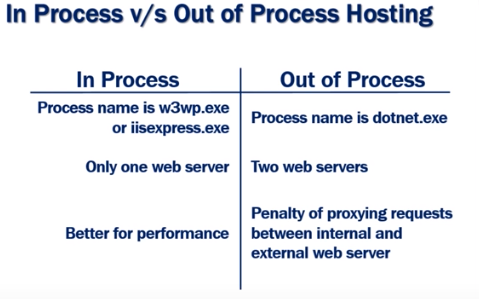
Maybe this is one of the most familiar uses of a reverse proxy. It can distribute the requests load among a set of identical servers accordingly to some specific algorithm providing support for scalability and availability of a system.

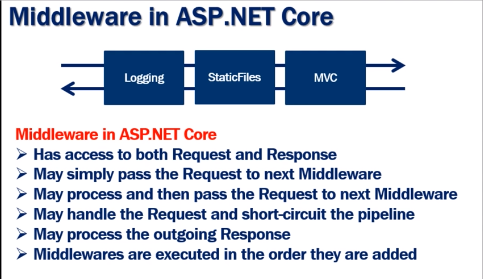
*API gateway*:

In a system with a microservice architecture, you have multiple servers offering different services through their APIs. You can use a reverse proxy to expose a single entry point for the combination of the server's APIs.

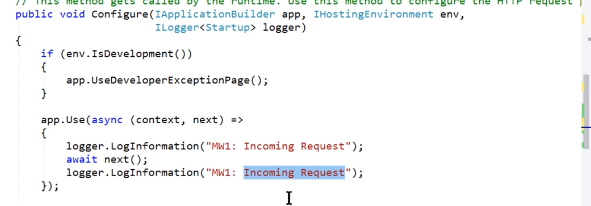
*Multiple website combining*:

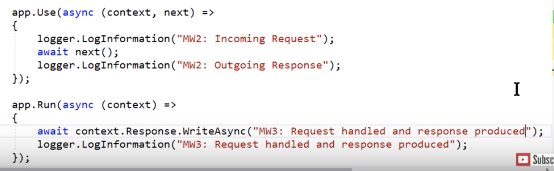
This is pretty similar to the API gateway context. In this case, you can have a single entry point for multiple websites, possibly with a centralized homepage.



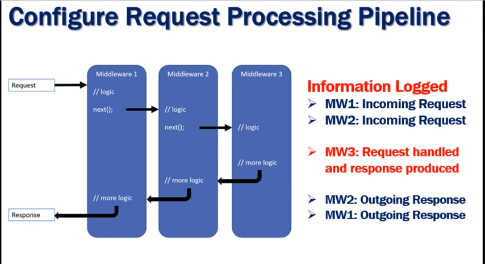


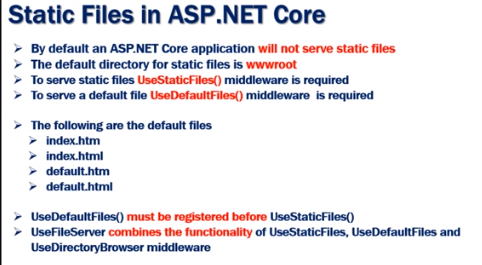




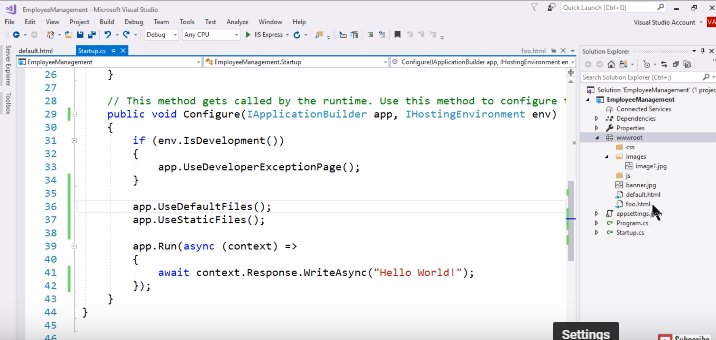


**Output:**



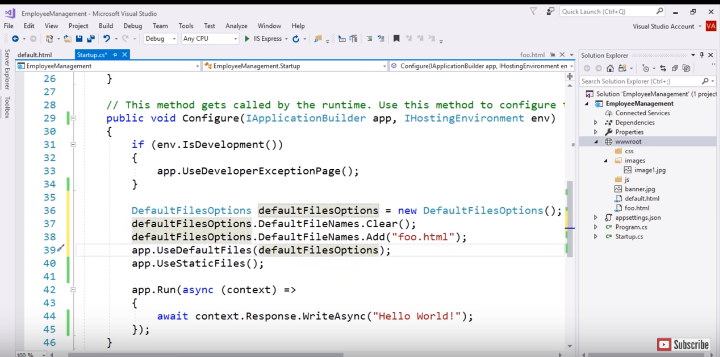


Ex:-For serve the default file then app.UseDefaultFile define before app.UseStaticFile.

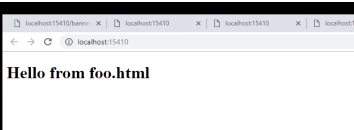


Output:

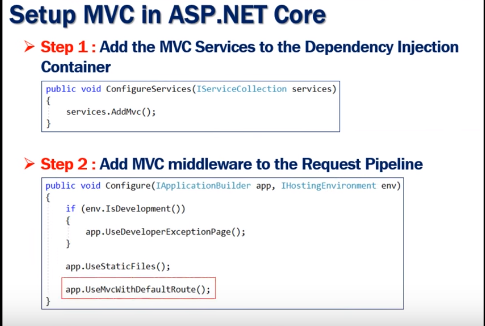


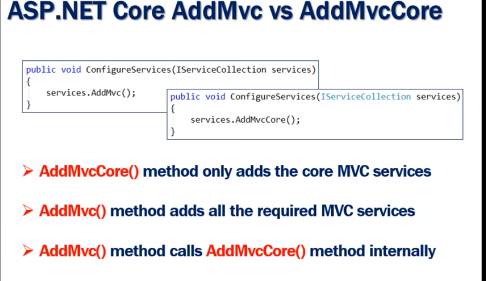


Output:-

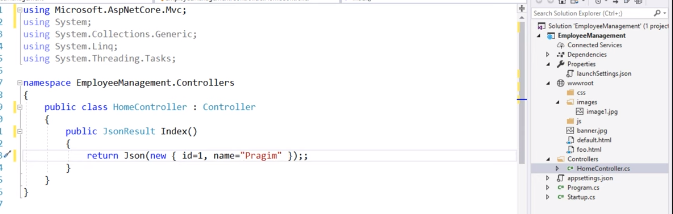








Ex: If we try to run the below code using the AddMVC() it will work fine and return the json result but is the same code we try to run using the AddMVCCore() it will show the exception. Because AddMVCCore() only add the Core MVc application but AddMVC() add all the required services.



**Startup.cs file**

In startup.cs file two method there-

1-ConfigureServices ()-> This method is used to define all the service which we want to use in our application and this method called in the runtime.

2-Configure()-> This method is used to configure the http request pipe line this method called in the riuntime.

## app.Use vs app.Run in ASP.NET Core middleware

Middleware are executed in the same order in which they are added. The difference is, middleware defined using **app.Use may call next middleware component in the pipeline. On the other hand, middlware defined using app.Run will never call subsequent middleware.** Let’s see via code.

public void Configure(Iappl2icationBuilder app, IWebHostEnvironment env)

{

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

{

await context.Response.WriteAsync("<html><body>");

await context.Response.WriteAsync("<div>Inside middleware defined using app.Use</div>");

await next();

await context.Response.WriteAsync("</body></html>");

});

app.Run(async context => {

await context.Response.WriteAsync("<div>Inside middleware defined using app.Run</div>");

});

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

{

await context.Response.WriteAsync("<html><body>");

await context.Response.WriteAsync("<div>Another Middleware defined using app.Use</div>");

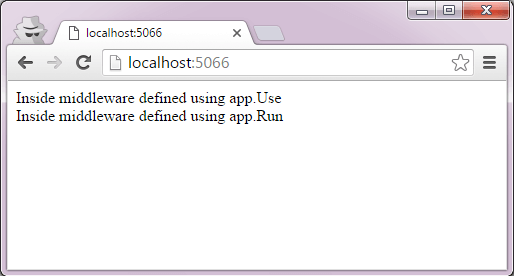
await next();

await context.Response.WriteAsync("</body></html>");

});

}

And as expected following is the output when you run the application.

[](https://www.talkingdotnet.com/wp-content/uploads/2016/04/app.Use-vs-app.Run-in-ASP.NET-Core-middleware.png)

As I told you earlier, middleware defined after app.Run will not be executed. And that’s why you don’t get the response of second app.Use middleware.

**Use of app.Map in ASP.NET Core middleware**

Map simply accepts a path and a function that configures a separate middleware pipeline. In this example, we will hook one middleware/delegate to HTTP pipeline using Map extension.

private static void MyDelegate(IApplicationBuilder app)

{

app.Run(async context =>

{

await context.Response.WriteAsync("Returning from Map");

});

}

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

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

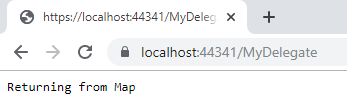
{

app.Map("/MyDelegate", MyDelegate);

}

In Map extension, we have hooked MyDelegate delegate with “/MyDelegate” path. So, when user will perform any HTTP call to the same path, the delegate function will get triggered.

**Output:**



**MapWhen() in .Net core**

MapWhen method supports predicate-based middleware branching, allowing separate pipeline to be constructed in a flexible way. In this example, we have mapped middleware execution with the presence of query string “q” in URL.

private static void HandleQuery(IApplicationBuilder app)

{

app.Run(async context =>

{

await context.Response.WriteAsync("Returning from Map");

});

}

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

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

//Execute when "q" is there in query string

app.MapWhen(context => {

return context.Request.Query.ContainsKey("q");

// return context.Request.Path.StartsWithSegments("/Info");

}, HandleQuery);

//Return response for other request

app.Run(async context =>

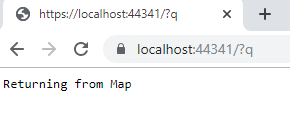
{

await context.Response.WriteAsync("From Run extension");

});

}

**Output:**



## Change Startup.cs class name

You can easily change the startup class name. Open the Startup.cs file and change the startup class name from Startup to “AppStart” (or anything of your choice). And also change the name of the constructor.

Now you need to tell ASP.NET Core about new Startup class name, otherwise application will not start. So open Program.cs file and change the UseStartup() call as follows:

public class Program

{

public static void Main(string[] args)

{

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

}

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

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

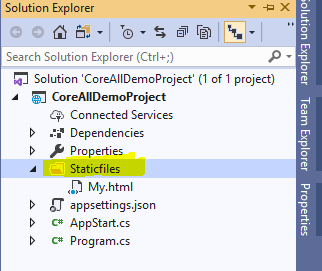
webBuilder.UseStartup<AppStart>();

});

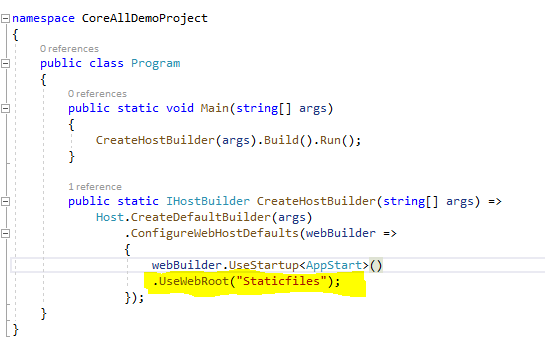
}

## Change wwwroot folder name

To change the name, right on wwwroot folder and rename it to “Staticfiles” (or anything of your choice.

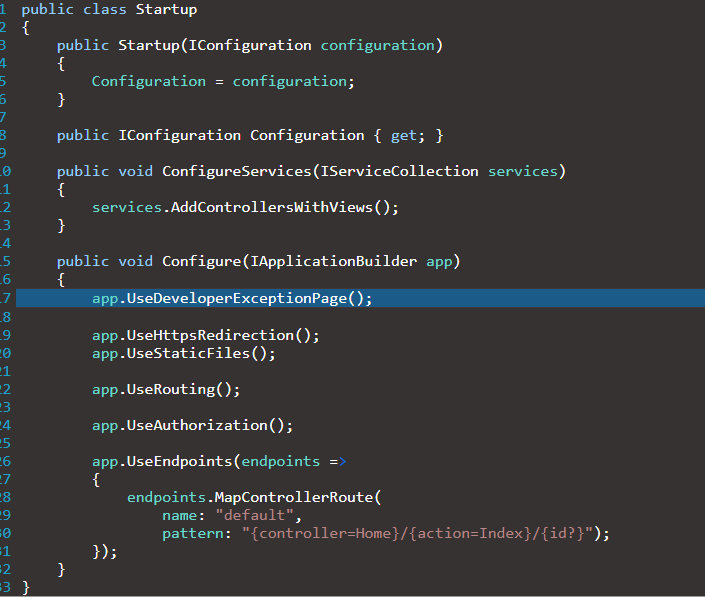


Now, open Program.cs file and add highlighted line of code to Main().

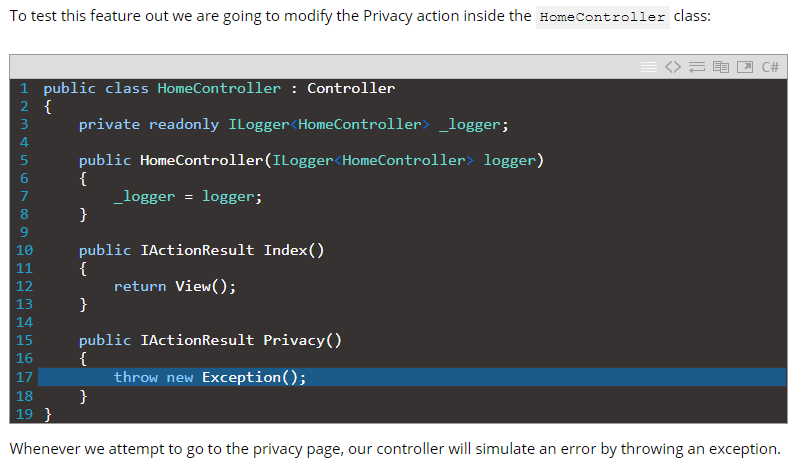


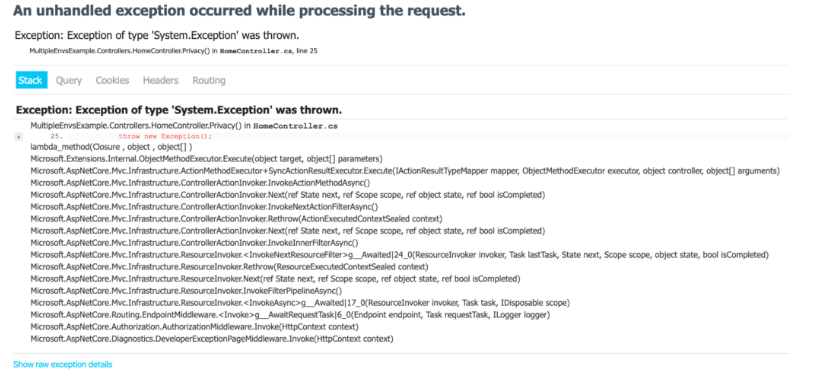
**Multiple Environments in ASP.NET Core**

Below code are available in Startup.cs file.



The UseDeveloperExceptionPage() method configures the application to call the debugging page whenever an error occurs.





Of course, this is a great resource when we need to know what went wrong. However, we don’t want our users to see this page.

So, how do we deal with this problem?

We are going to solve this problem by using multiple environments.

## Accessing the Environment in the Startup Class

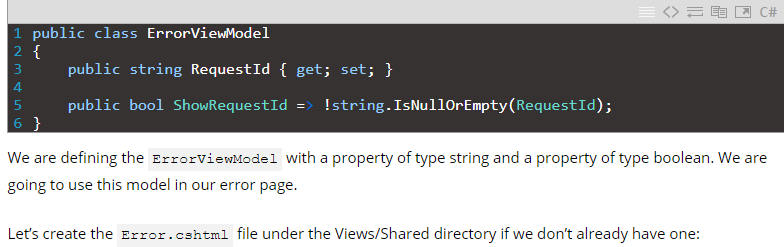
Now, we are going to see how to use our environments to manipulate our application. Currently, our application displays a debugging page whenever there is an error. This is great while we’re working in the development environment.

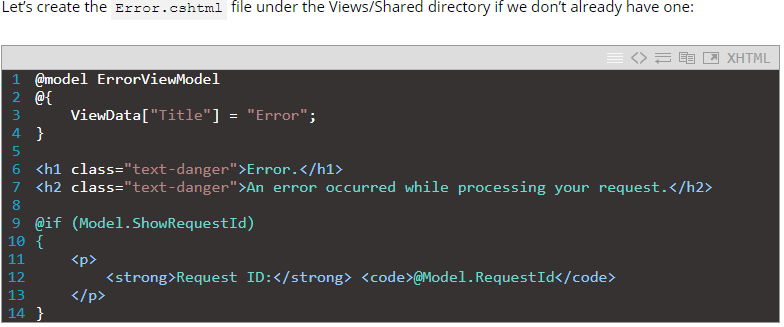
But, what about the production environment?

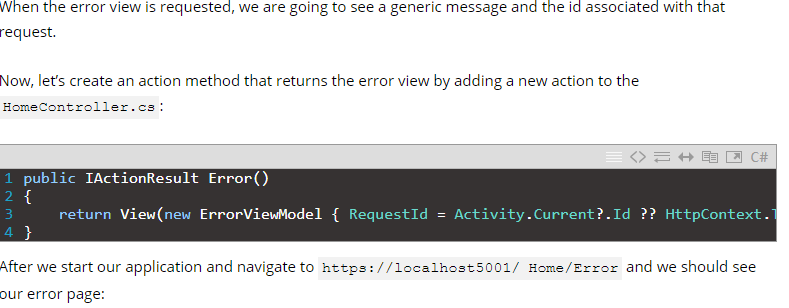
For the production environment, we want to display a custom error page. To do this, we are going to explore two methods: dependency injection and method conventions in the Startup class.

But first, we are going to create our custom error page.

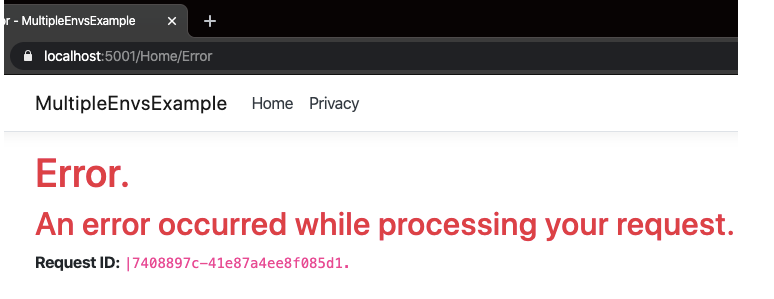
Let’s create the ErrorViewModel class under the Models directory if we don’t already have one:







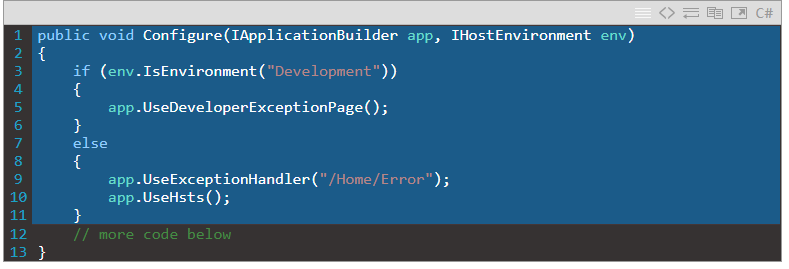
|  |  |
| --- | --- |
| 1  2  3  4 | public IActionResult Error()  {      return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });  } |



### By Dependency Injection in the Startup Class

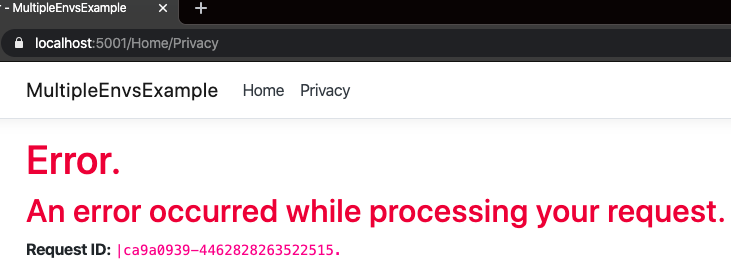
However, we want our error page to display when an error occurs in the production environment. The simplest way to do this by taking advantage of dependency injection in the Startup class.

Let’s go ahead and modify Startup.cs:



Using the DI we supply an argument of type IHostEnvironment into the Configure method. Then we use a simple if-else structure to determine which page is displayed by using the IsEnvironment method. This method checks which environment we are using. If it is set to Development then we use the developer exception page. For any other environment, we use an exception handler pointing to the custom error page.

After starting the application in the production environment and clicking on Privacy, an error is simulated and we see our custom error page:

**[](https://code-maze.com/wp-content/uploads/2020/04/error-page-2-e1586112605653.png)**

**Purpose Of UseDeveloperExceptionPage() In .Net Core.**

The major purpose of  UseDeveloperExceptionPage() is to help the user to inspect exception details during the development phase.

**Namespace for the method** - Microsoft.AspNetCore.Builder

**Purpose of function**

* To capture Synchronous and Asynchronous SystemException instance from the pipeline & generate HTML error response. It returns a reference to the application after the operation is completed.
* We use the UseDeveloperException() extension method to render the exception during the development mode
* This method adds middleware into the request pipeline which displays developer-friendly exception detail page. This helps developers in tracing errors that occur during the development phase

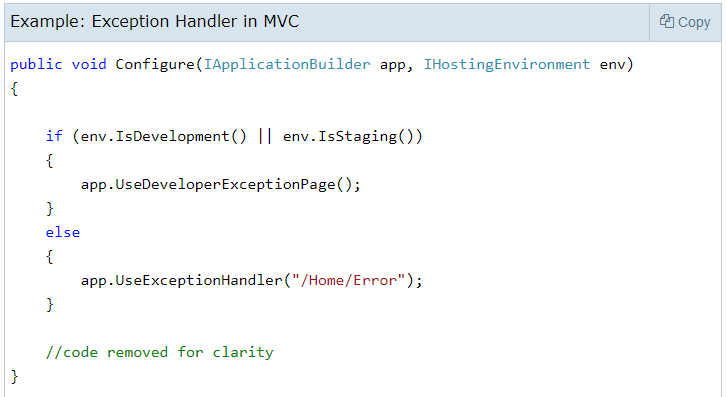
To use this method open the Startup.cs file, locate the Configure() method and make the below changes to it:

1. **public** **void** Configure(IApplicationBuilder app, IWebHostEnvironment env)
2. {
3. **if** (env.IsDevelopment())
4. {
5. app.UseDeveloperExceptionPage();
6. }
7. **else**
8. {
9. app.UseExceptionHandler("/Error");
10. }
11. ....
12. ....
13. }

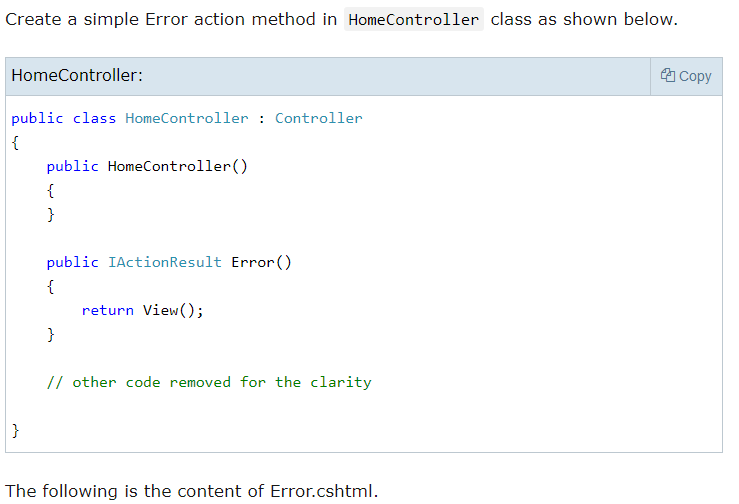
This is the way developers can add exception-related functionality to the code and can inspect their exceptions during the development phase.

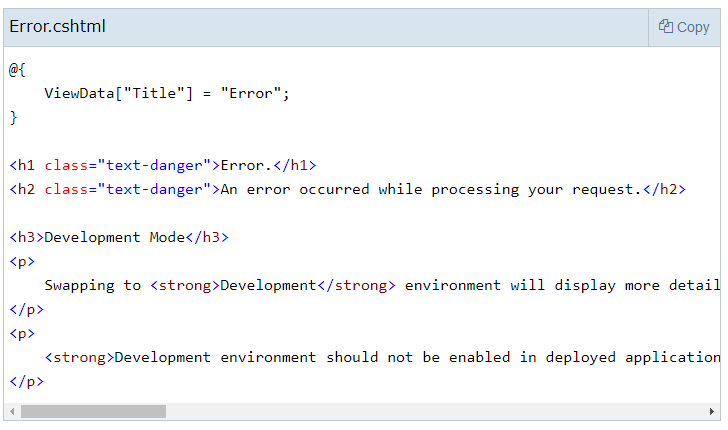
### UseExceptionHandler

In MVC Core application, we might want some other controller to handle all exceptions and display custom user friendly error messages. The UseExceptionHandler extension method allows us to configure custom error handling route. This is useful when an application runs under production environment.

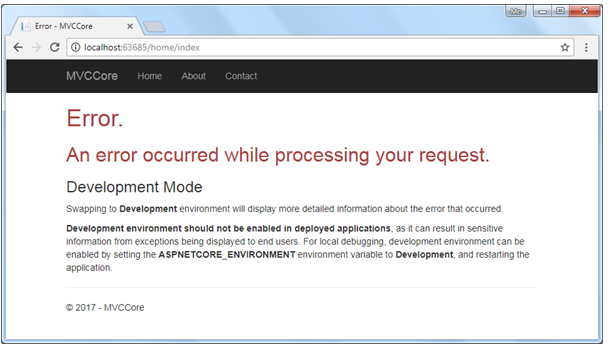


In the above example, the UseExceptionHandler("/Home/Error") sets the error handler path. If an error occurred in the MVC application then it will redirect the request to /home/error, which will execute the Error action method of HomeController.



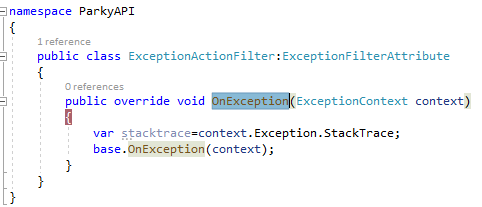


Now, when an error occurs, it displays the page shown below.

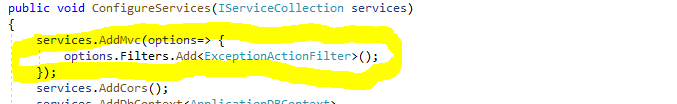


**Global Custom Exception in .Net Core**

-For crearting the custom exception first we need to create one class which is inherited from the ExceptionFilterAttribute class and overide the OnException() method like below. We can write the code here based on the requirement like we can log the error here also if require.



-Then we need to configure that custom class in the StartUp.cs file like below.



-So after that if any exception will occure in any class or controller it will call the our custom created class.

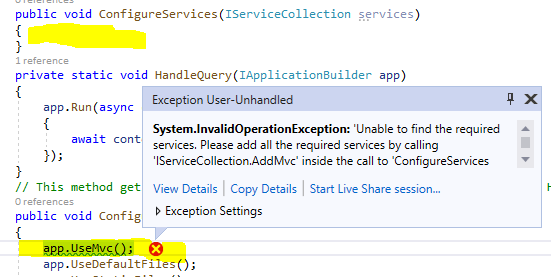
**Is ConfigureService() and Configure() are mandatory method**

-ConfigureService() are optional method where Configure() method is mandatory.

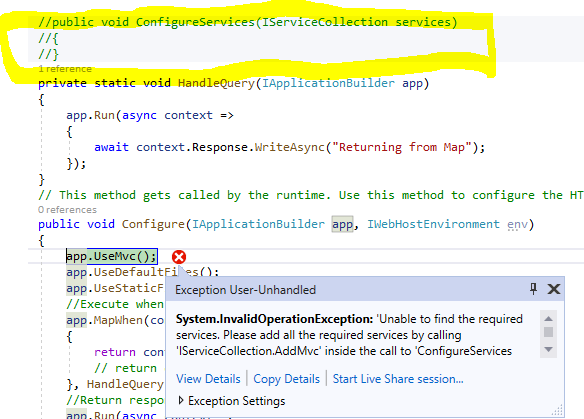
-ConfigureService() method called before the Configure() method.

-If we define some service in configure() method at that time we need to define the configreservice() method Otherwise we will get the exception. For example if we use the app.UseMVC() service in configure() method but did not use the define that service and configureservice() or not using configureservice() method then we will get the below exception.

-If we did not define the service and configureservice().



-same exception if we commented and remove the configureservice method()

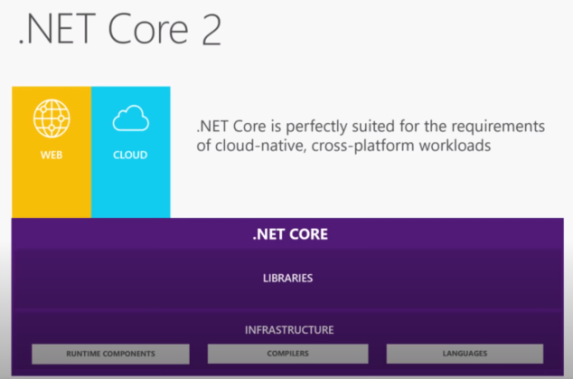


IApplicationBuilder is an interface that contains properties and methods related to current environment. It is used to get the environment variables in application.

IHostingEnvironment is an interface that contains information related to the web hosting environment on which application is running. Using this interface method, we can change behavior of application.

**What is New in .net Core 3.0 and later version**

-In .Net core 2 we can develop only the cloud based web application only.

****

-In .Net Core 3 apart from the cloud based web application we can develop desktop, IOT and AI.

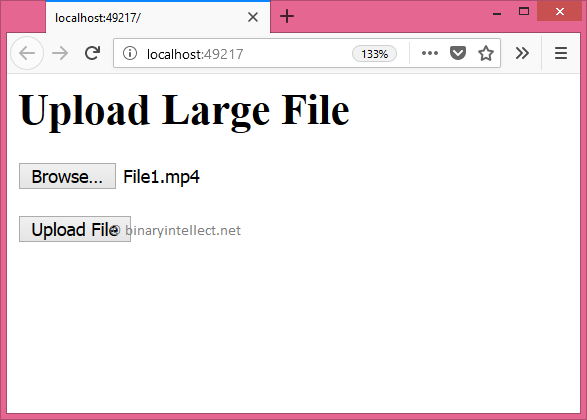


**Upload Large Files In ASP.NET Core**

By default, ASP.NET Core allows you to upload files up to 28 MB (approximately) in size. To raise this limit you need to make a couple of additions to your code. And there are a few variations of how that can be done.

## Understanding the problem

Consider a simple file upload page as shown below:



The page allows you to pick a file to be uploaded using a file input field. Clicking on the Upload File button attempts to upload the file on the server. The razor markup that renders this page is shown below:

<h1>Upload Large File</h1>

<h2>@ViewData["message"]</h2>

<form asp-controller="Home"

asp-action="Upload"

method="post"

**enctype="multipart/form-data"**>

<input type="file" name="file" />

<br /><br />

<button type="submit">Upload File</button>

</form>

Notice that the enctype attribute of the <form> element is set to multipart/form-data since we want to upload a file. The form POSTs to the Upload() action of the HomeController.

The Upload() action is shown below:

public IActionResult Upload

(IFormFile file,[FromServices] IHostingEnvironment env)

{

string fileName = $"{env.WebRootPath}\\{file.FileName}";

using (FileStream fs = System.IO.File.Create(fileName))

{

file.CopyTo(fs);

fs.Flush();

}

ViewData["message"] =

$"{file.Length} bytes uploaded successfully!";

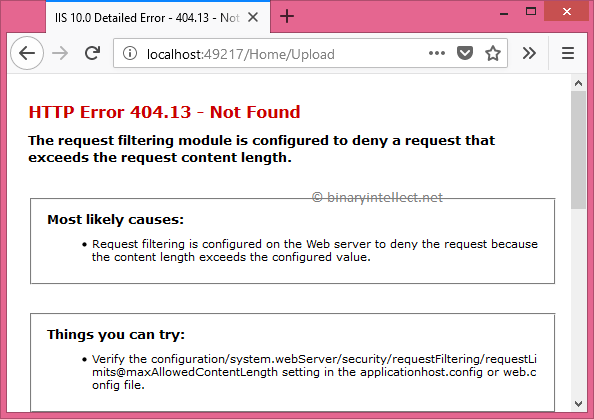
return View("Index");

}

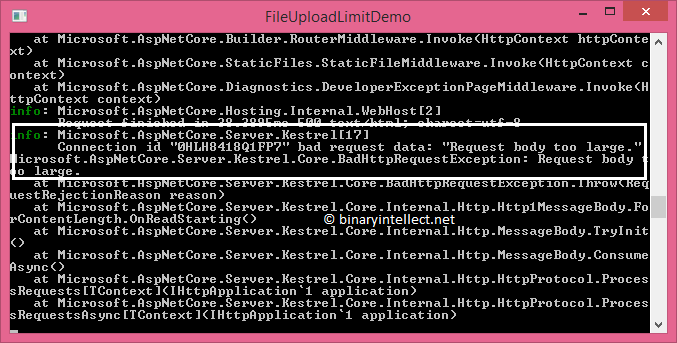
 It basically saves the uploaded file to the wwwroot folder. The uploaded file is received through IFormFile parameter and the IHostingEnvironment is injected in order to compute the physical path of the file.

To understand the problem, run the application and try to upload a file with larger size, say 100 MB.

If you are using **IIS Express** then you will get this error message:



And if you are using **Kestrel**then the error message should resemble this:



So, depending on whether you are hosting under IIS or Kestrel the solution is going to slightly differ. Let's see how.

## Solution for IIS Express

For IIS Express, you need to add web.config to your project and add the following markup into it:

<system.webServer>

<security>

<requestFiltering>

**<requestLimits maxAllowedContentLength="209715200" />**

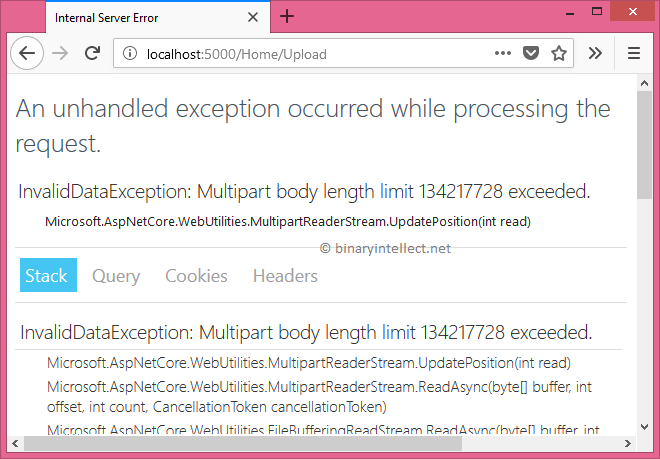
</requestFiltering>

</security>

</system.webServer>

The requestLimits element's maxAllowedContentLength is set to 200 MB. You can change this limit as per your requirement. This will change the default request length to 200 MB.

Save the project and run the application. You won't get the earlier error but now a new error crops up as shown below:



Now the error revels that there is some limit on the multipart form body. To overcome this limit add the [RequestFormLimits] attribute on top of the Upload() action as shown below:

[HttpPost]

**[RequestFormLimits(MultipartBodyLengthLimit = 209715200)]**

public IActionResult Upload(IFormFile file,

[FromServices] IHostingEnvironment env)

{

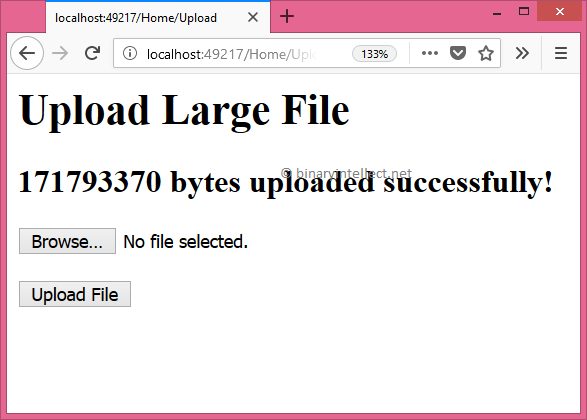
...

...

}

So, we set the MultipartBodyLengthLimitproperty of [RequestFormLimits] to 200 MB.

Now, run the application again. This time it will happily upload the file.



The maxAllowedContentLength attribute of <requestLimits> change the setting for the whole application. What if you don't want that much content length for one or more actions?

In that case you can use [RequestSizeLimit] attribute to override the content length limit. This will be clear in the next section because the solution for Kestrel uses it.

## Solution for Kestrel

If you are using Kestrel you have the option to change the request content length settings either at action level or at the application level.

To change the settings at the action level you need to use two attributes namely [RequestSizeLimit] and [RequestFormLimits]. The [RequestSizeLimit] attribute sets the maximum length of a request in bytes whereas [RequestFormLimits] sets the maximum length for multipart body length. The following code shows the Upload() action decorated with these attributes:

[HttpPost]

**[RequestFormLimits(MultipartBodyLengthLimit = 209715200)]**

[**RequestSizeLimit(209715200)]**

public IActionResult Upload(IFormFile file,

[FromServices] IHostingEnvironment env)

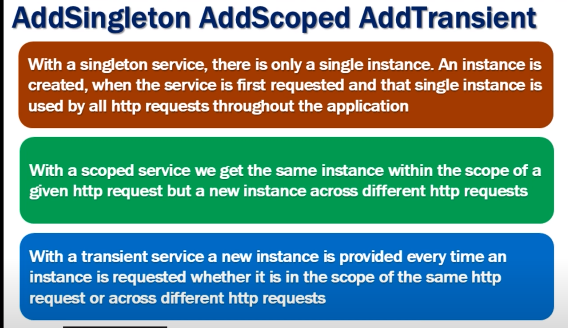
{

...

...

}

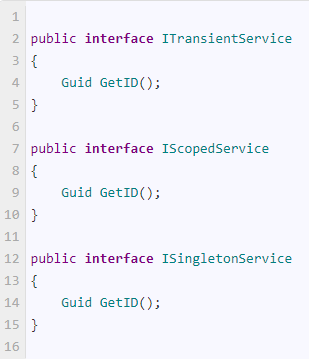
# Difference between AddTransient, AddSingleton & AddScoped



### Creating the Service Interface

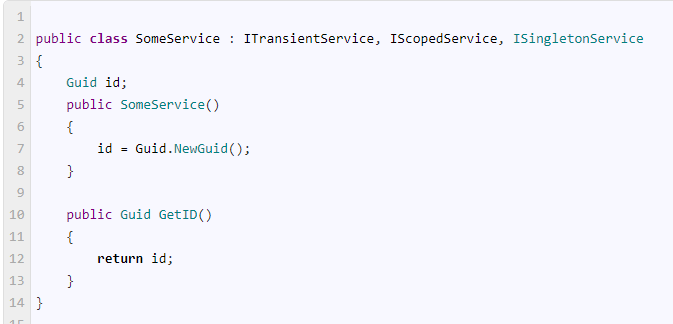
Create the service SomeService in the services folder.

Add the three interfaces. One for each lifetime provided by the ASP.NET Core. The interface is very simple. It contains single method GetID, which must return a unique ID.



### Creating the Service

Next, let us create a single service, which implements all the three Interfaces.



The service generates a unique id, whenever it is instantiated and returns that id in the GetID method.

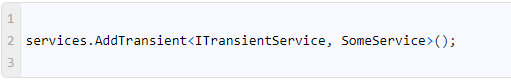
Now, let us look at each service lifetime in detail.

## Transient

The Transient services are always created new, each time the service is requested.

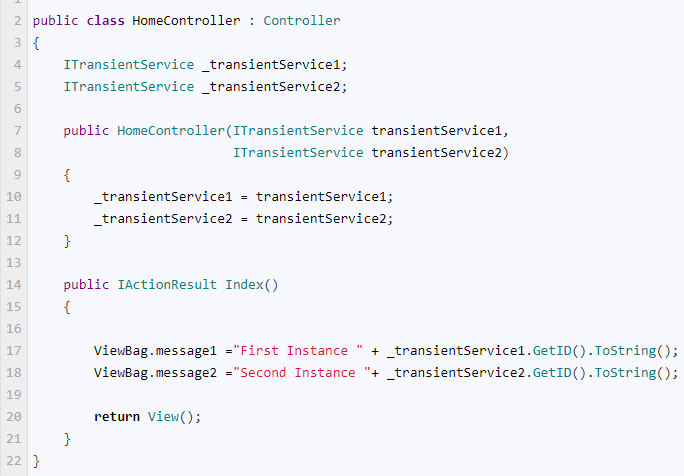
### Register the Transient Service

Now, under ConfigureServices method of the startup class register the SomeServive via ITransientService interface as shown below



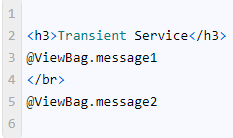
### Inject it into Controller

Open the HomeController and inject the two instance of the SomeService as shown below

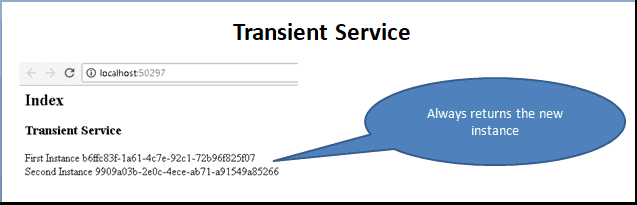


### View

Open the view and add the following code



Run the application and you should see two different Guid are displayed on the screen. It is evident that we have received two new instances of the Transient service.



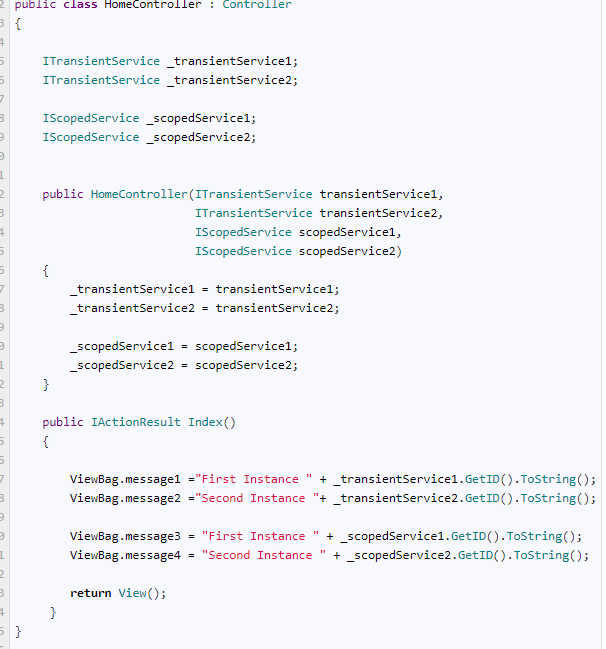
### Register the Scoped Service

Under the ConfigureServices method register the SomeService using the AddScoped method & using the IScopedService interface as shown below.

services.AddScoped<IScopedService, SomeService>();

### Inject scoped service into Controller

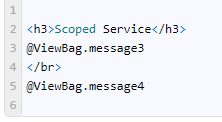
Next, inject the service into the controller. We already have transient service injected into the controller. Let us not change that. The HomeController is now as below



In the action method, message3 & message4 comes from the scoped service.

### View

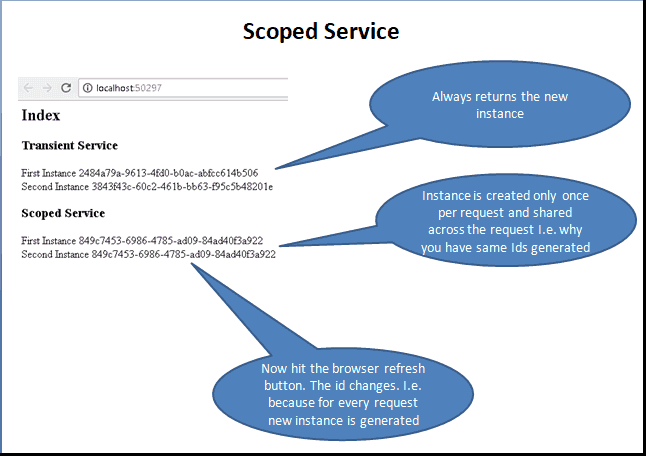
In the view add these lines



Run the application

The instance created only once per request, i.e. why you have same id’s generated.

Now hit the browser refresh button. The id changes i.e. because the new instance is created for every request.

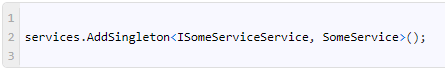


## Singleton

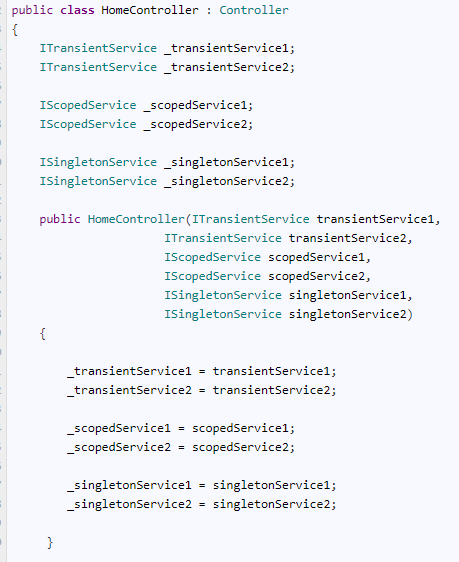
Single Instance of the service is created when it was requested for the first time. After that for every subsequent request, it will use the same instance. The new request does not create the new instance of the service but reuses the instance already created.

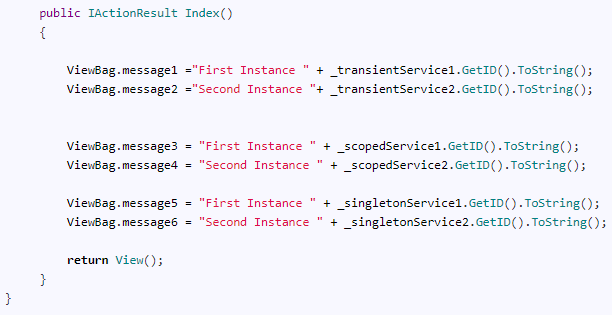
### Register the Singleton Service

Singleton services are registered using the AddSingleton method.

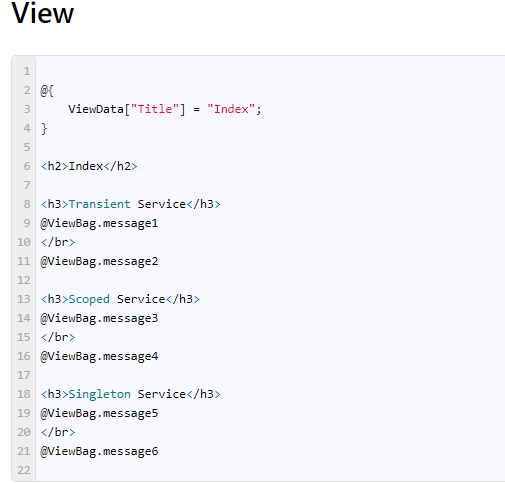


### Inject Singleton service into Controller

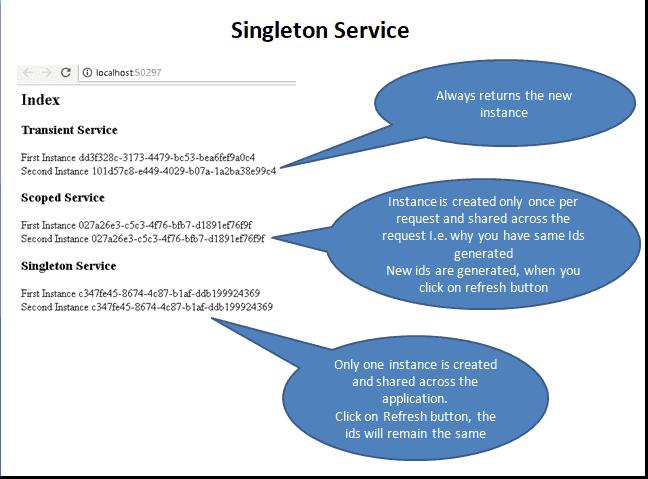




First, we are injecting 6 instances the SomeService. Two instances per each interface. This is done in the constructor of the Controller.



Run the application. The ids generated from Singleton services are same and will not change even if you refresh the application. You can see it from the image below



When to use which service

-Singleton will use if we need single instance which is share by all the user example error logs.

-Transient will use when we have no need to maintain any state in each request.

-Scope will use when we need to maintain some state for the same request.

**What is Kestrel?**

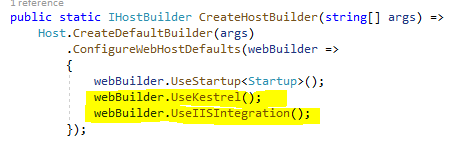
Kestrel is an open source, cross platform, light weight and a default webserver used for Asp.Net Core applications.Asp.Net Core applications run Kestrel webserver as in-process server to handle web request. Kestrel webserver is based on async I/O library called [libuv](https://github.com/libuv/libuv" \t "_blank) primarily developed for Node.js.

By default, all Asp.Net core project templates include Kestrel webserver when we create new Asp.Net Core Projects. As said before, Kestrel is a very light weight webserver that does not have every advanced features of webservers like IIS, Nginx, Apache, etc has. Due to its lightweight nature, Kestrel provides better request processing performance to Asp.Net Core applications.

features of Kestrel,

1. Kestrel does not support multiple applications sharing same port similar to IIS where applications are differentiated by host header value.
2. Kestrel is cross platform, runs in Windows, LINUX and Mac.
3. Kestrel webserver supports SSL.
4. **Why Kestrel Webserver when we have IIS/Nginx/Apache?**
5. The primary requirement of Asp.Net Core is to make Asp.Net Core applications to run across multiple platforms (or cross platform). Though, IIS is feature rich and highly used webserver it is a windows-only webserver. Without Kestrel, to run Asp.Net Core application on other cross platform webservers like Nginx, Apache, the Asp.Net Core application need to satisfy the Startup criteria of each of these webservers. In other words, each webserver has a different Startup configurations and this will make Asp.Net Core applications have different Startup mechanisms. This is why Asp.Net Core applications use Kestrel webserver as an in-process server where the application will have same and consistent Startup (Main() and Startup.ConfigireServices() & Startup.Configure()) process even when offering cross platform support.

**How to use Kestrel?**



In the above Main() method, calling UseKestrel() on WebHostBuilder object will configure the Kestrel webserver for the application.

**Note**– Every Asp.Net Core application requires a Host process to start Kestrel webserver and initialize application for request processing. Calling UseIISIntegration() integrates Kestrel with IIS webserver, let’s see this in detail in next section.

**Externally Hosted applications**

if your application is external facing public website then the application should be deployed behind a full-fledged webserver like IIS or Nginx or Apache for increased security and to get feature rich support. These webservers act as proxy (commonly called reverse proxy in this scenario) and forward the request to Kestrel for request processing. This is why the Program.Main() code calls UseIISIntegration() to integrate IIS as reverse proxy to forward request to Kestrel. This is the default mode the project template code uses.The request flow goes like below,

http://www.codedigest.com/PageFiles/5/Kestrel%20request%20flow.png