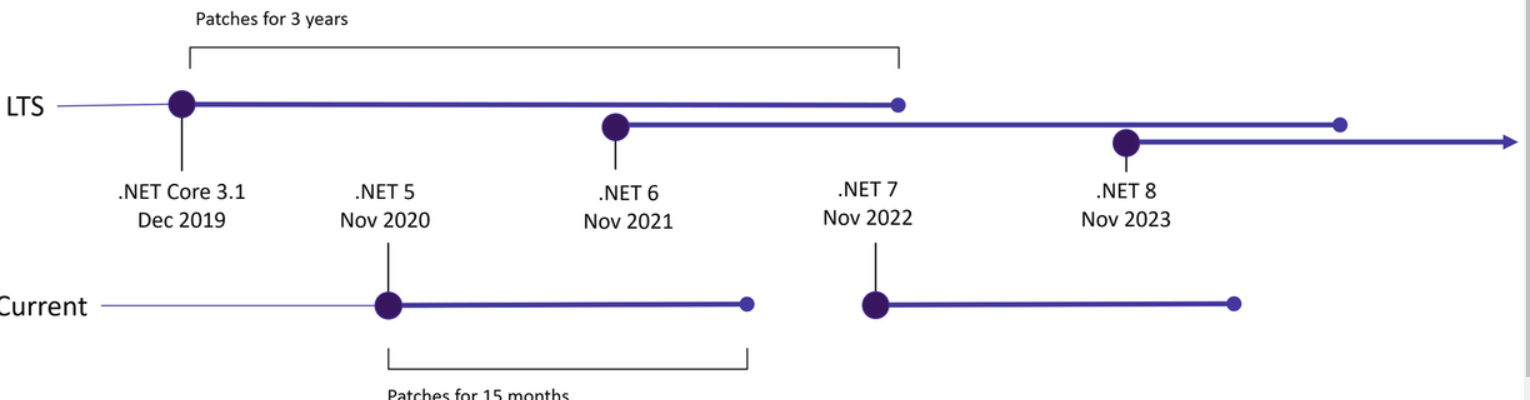
**dot.Net Core 6:**

* Early .Net was designed to work on many platforms. But the way .Net framework was built sharing code between platforms was not possible.

Ex: *A .Net core project can refer and use code from .Net framework project but vice versa is not possible*.

* .Net Standard is created as an interface to share code between platforms. .Net Core is an implementation of that .Net Standard interface. With this we can create .Net standard class libraries that can refer to .Net framework projects. Due to this .Net Core 3.1 was using .Net standard 2.1, .Net standard 1.6 is being used by .Net Core 1.0. Hence to avoid this confusion Microsoft came up with single version of .Net , which can be used on different platforms.



* All odd number releases like .Net 5, .Net 7 etc., will have non-LTS (Long-Term Support) and every even numbered releases like .Net 6, .Net 8 will have LTS.

.Net 6 unified all the .Net technologies under one umbrella.



**Razor Validations:**

asp-validation-summary 🡪 Displays list of validation errors when view is rendered, it displays this list from IEnumeration called ‘ValidationSummary’

**Commands**:

dotnet new 🡪 creates new projects , configuration file or solution file. Below is the table :

|  |  |
| --- | --- |
| Name | Description |
| *web* | Template used to set up minimum code for Aps.net Core development.  Example:  dotnet new web –output <projectname> --framework <frameworkversion> |
| mvc | Template used to create an mvc project  Dotnet new mvc --output myfirstproject --framework netcoreapp3.1 |
| webapp | Template used to create web project with razor pages |
| blazorserver | Template used to create web project to use blazor server. |
| angular | Template used to create client side features using Angular Javascript framework. |
| React | Template used to create client side features using React Javascript framework. |
| Reactredux | Template used to create client side features using React Javascript framework with redux library. |
| Globaljson | Template adds global.json file to a project specifying the .Net core version that will be used. |
| sln | This template creates a solution file, which is used to group multiple projects and is commonly used by Visual Studio. The solution file is populated with the **dotnet sln add**command |

dotnet new globaljson --sdk-version 3.1.101 --output MySolution/MyProject

dotnet new web --no-https --output MySolution/MyProject --framework netcoreapp3.1

dotnet new sln -o MySolution

dotnet sln MySolution add MySolution/MyProject

services.AddControllersWithViews();

endpoints.MapDefaultControllerRoute();

Above ones are used to configure or enable MVC routing

**Object Initializer**:

Normal use: Product objProduct = new Product();

objProduct.Name=’’;

objProduct.Price=’’;

With Object Initializer :

Product objProduct = new Product

{

Name=’’,

Price=’’

};

**Collection Initializer**  It can used to initialize the collections.

Normal use:

string[] strArr = new string[3];

strArrr[0]=”C#”;

strArr[1]=”Language”;

strArr[2]=”Features”;

With collection Initializer:

new string[] {“C#”,”Language”,”Features”}

**Index Initializer:**

Dictionary<string, Product> products = new Dictionary<string, Product>

{

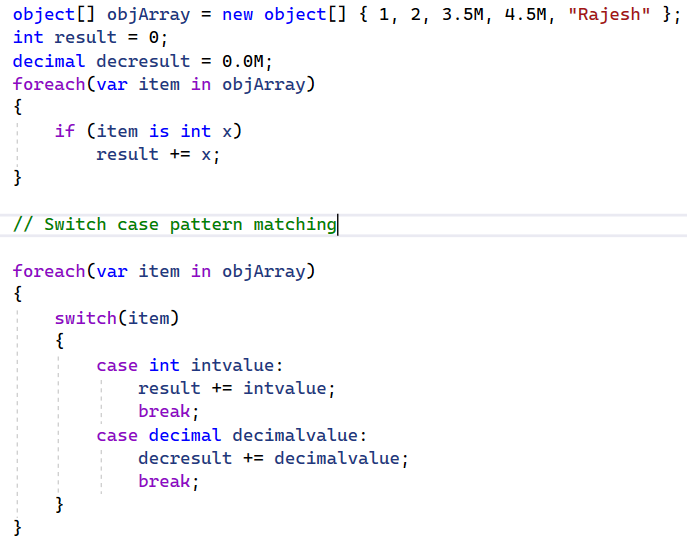
{ "Kayak", new Product { Name = "Kayak", Price = 275M } },

{ "Lifejacket", new Product{ Name = "Lifejacket", Price = 48.95M } }

};

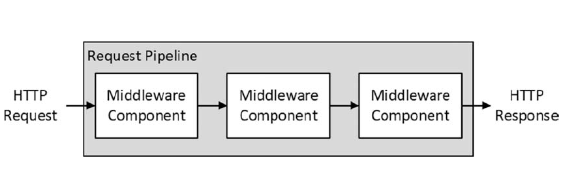
**Pattern Matching:**

To check if an object is of a specific time we can use the keyword “is”



**Asp.Net Core Platform**

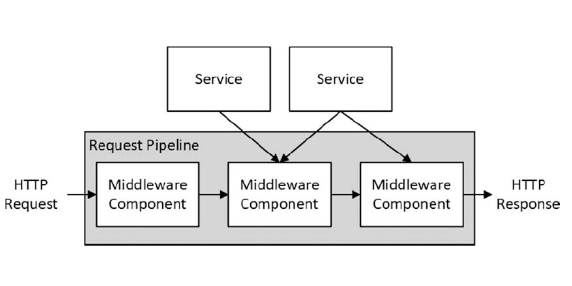
* Middleware components are arranged in a chain, known as the request pipeline.
* When a new HTTP request arrives, the ASP.NET Core platform creates an object that describes it and a corresponding object that describes the response that will be sent in return. These objects are passed to the first middleware component in the chain, which inspects the request and modifies the response. Finally Asp.Net Core returns the response.



* If no response is generated Asp.Net Core then 404 Not Found will be returned.

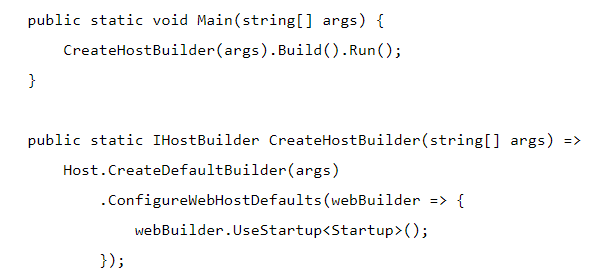
**Services:**

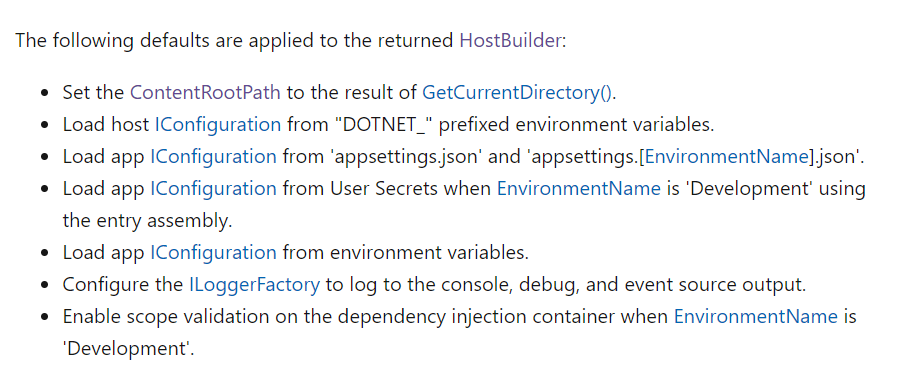
* Services are objects that provide features in a web application. Any class can be used as a service, and there are no restrictions on the features that services provide.
* Services are managed by ASP.NET Core, and a feature called dependency injection makes it possible to easily access services anywhere in the application, including middleware components.

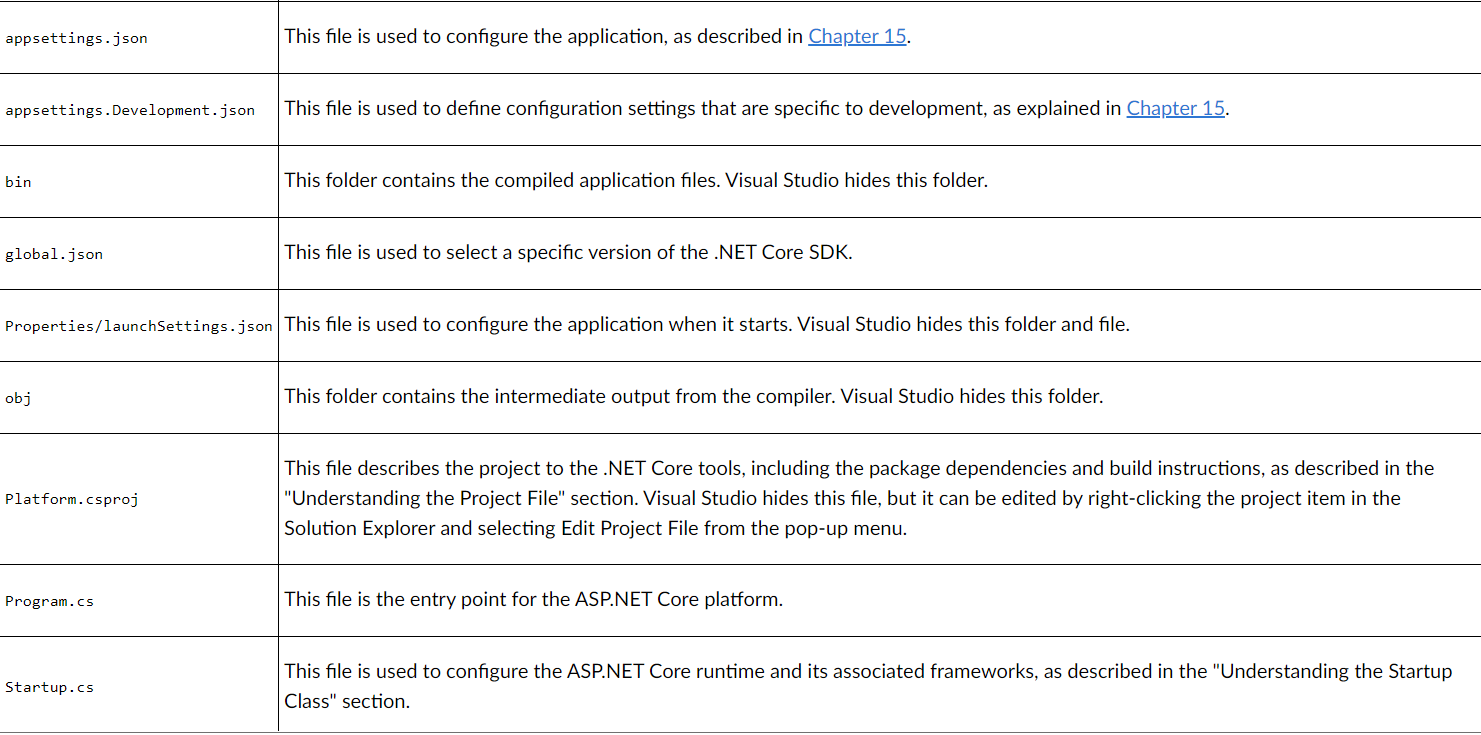


**Program.cs (Entry point for .Net Core application):**

* CreateDefaultBuilder(args) 🡪 Responsible for setting up basic features of Asp.Net platform including creating up services for loading configuration for data and logging. It is also responsible for creating a HTTP server *Kestrel* for taking HTTP requests and adds support for working with IIS.
* The result from CreateDefaultBuilder is passed to ConfigureWebHostDefaults method which selects ‘StartUp’ class as the next step in start up process.

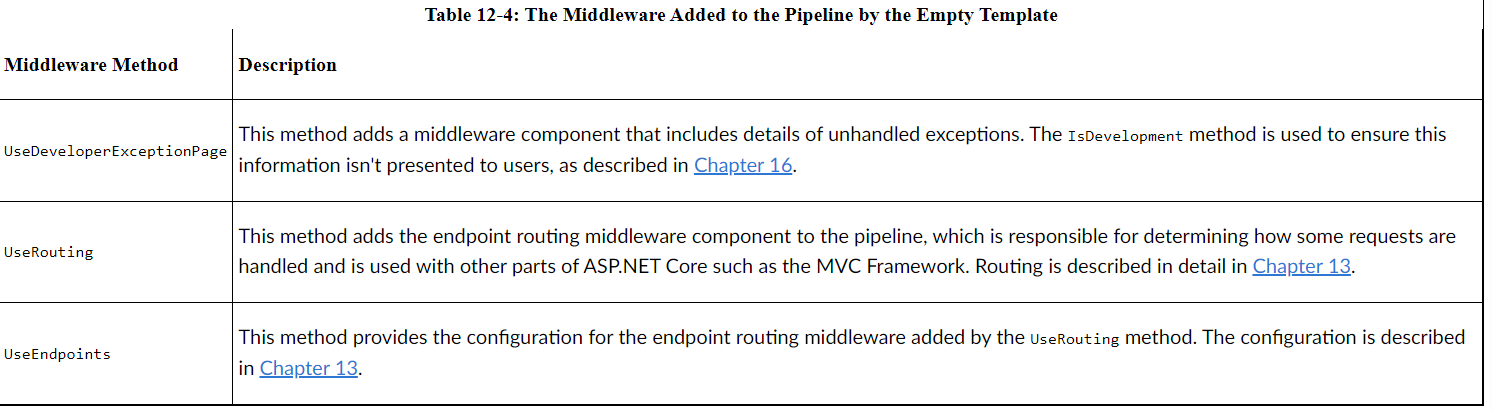


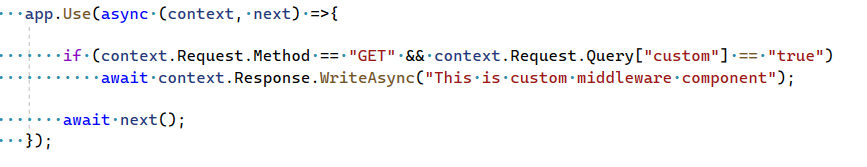




**Startup.cs**

* Most of the setup needed for the application is performed here with two methods:
  + ConfigureServices 🡪 Used to define services needed for application
  + Configure() 🡪 Used to define middleware components for the request pipeline.

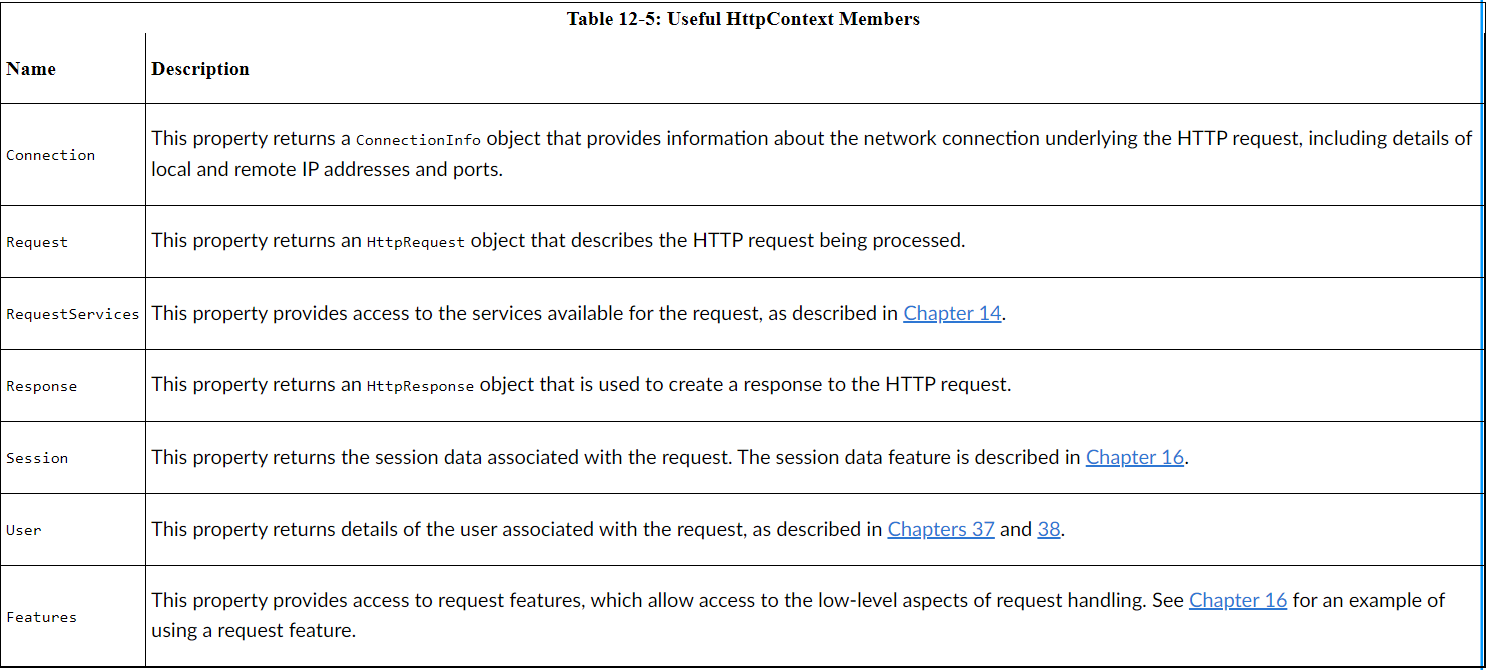




* + The *Use* method registers a middleware component that is typically expressed as a lambda function that receives each request as it passes through the pipeline (there is another method used for classes, as described in the next section).
  + The arguments to the lambda function are an HttpContext object and a function that is invoked to tell ASP.NET Core to pass the request to the next middleware component in the pipeline.
  + HttpContext object describes the HTTP request and HTTP response and provides additional context including details of user associated with the request.

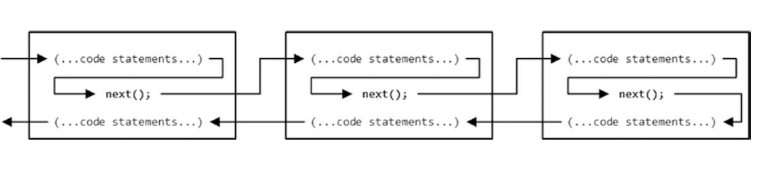
context specifies HttpContext here and next() specifies an async function that calls next middleware.

The above middleware component will first call the next middleware using ‘next()’ to pass the request in further pipeline and then a response object will be written.



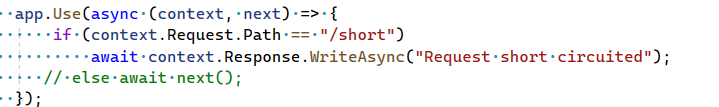
**Defining Middleware**

* Instead of adding middleware components which makes it hard in startup.cs we can create middleware classes.
  + app.UseMiddleWare<MiddleWareClass>();

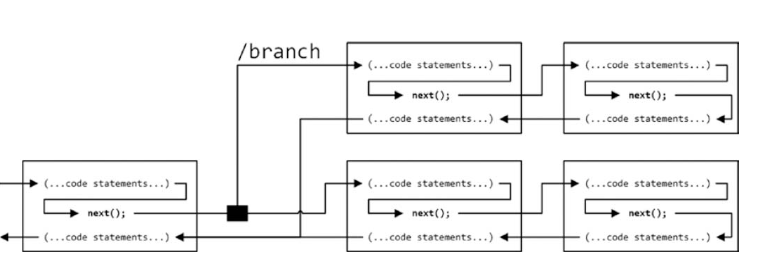
MiddleWareClass will be called and it’s ‘Invoke’ method is called to process requests When the ASP.NET Core is started, the <MiddleWareClass> class will be instantiated, and its **Invoke** method will be called to process requests as they are received.

Caution: A single middleware object is used to handle all requests, which means that the code in the Invoke method must be thread-safe.

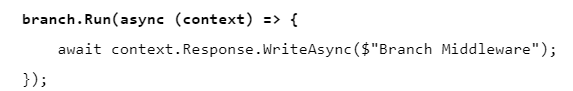
If any middleware component doesn’t call the ‘next’ function it means it is short circuited the further pipeline as httpcontext object can’t be passed further in the pipeline.

Example: 

**Pipeline Branches** (with MAP) method: ***Map*** method is used to create a section of pipeline that is used to process requests for specific URLs,

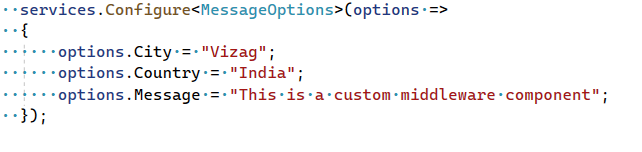


**Terminal Middleware**: These don’t have ‘next’ because they terminate the middleware components with ‘Run’ method which uses ‘Use’ behind the scenes.

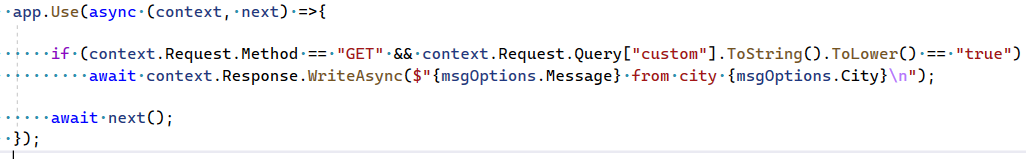


**Configuring Middleware with Options**:

* We can configure middleware with *options* pattern. We can configure them in ‘ConfigureServices’ method

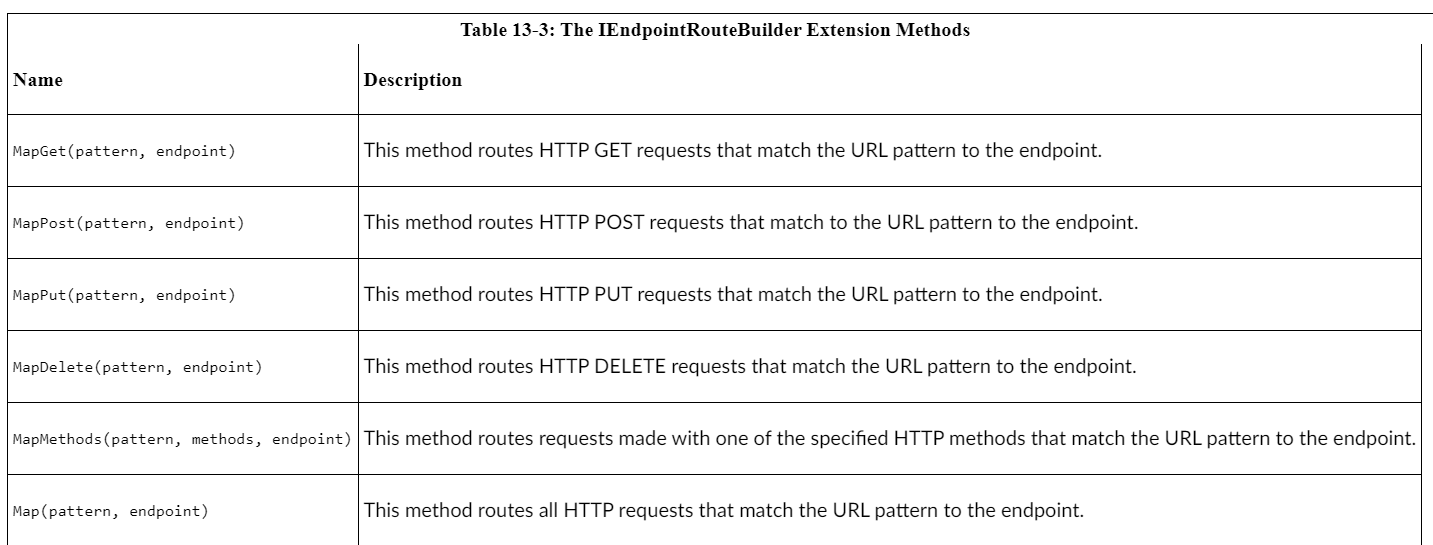


And use like below:

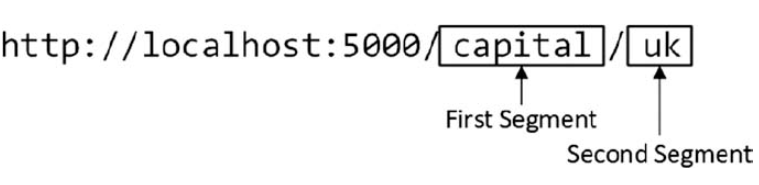


**Routing**

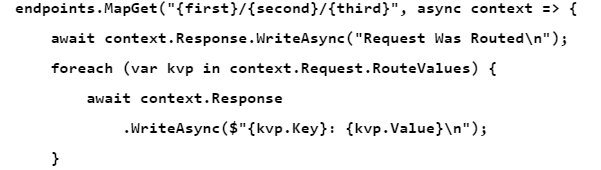
* Adding custom middleware components can be very difficult to maintain, difficult to enhance in future, difficult to test as well. Hence we can use ‘Routing’ feature.
* Routing adds a necessary middleware to process the requests for the pipeline.
  + UseRouting() : Adds ‘Microsoft.AspNetCore.Routing.EndpointRoutingMiddleware’ middleware to the app for processing requests to pipeline.
  + UseEndPoints(): Defines endpoints to match the URLs.

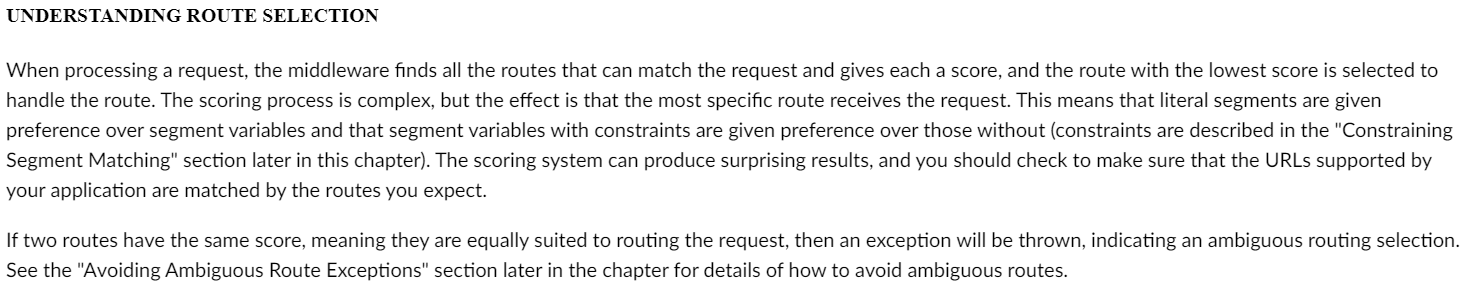
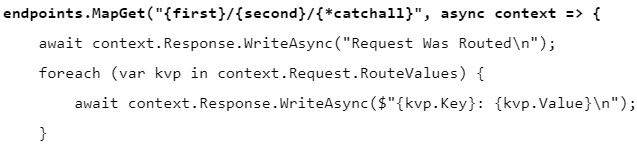


* The Routing middleware always ‘short circuits’ other routes when URL pattern is matched, in the sense that when a URL is matched then below routes or URLs will not be matched. Endpoints doesn’t have the ‘next()’ Func<Task> in it.
* Normal middlewares however pass onto next middlewares as next() function is usually called.
* The below are called *literal segments or static segments* as the first segment has to be *capital* and second segment has to be *uk*

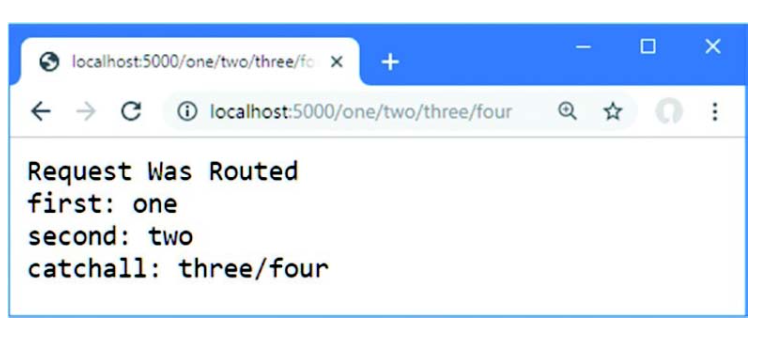


* *Segment Variables* : Also known as route parameters expand the range of path segments that a pattern segment will match, these are denoted by curly braces.



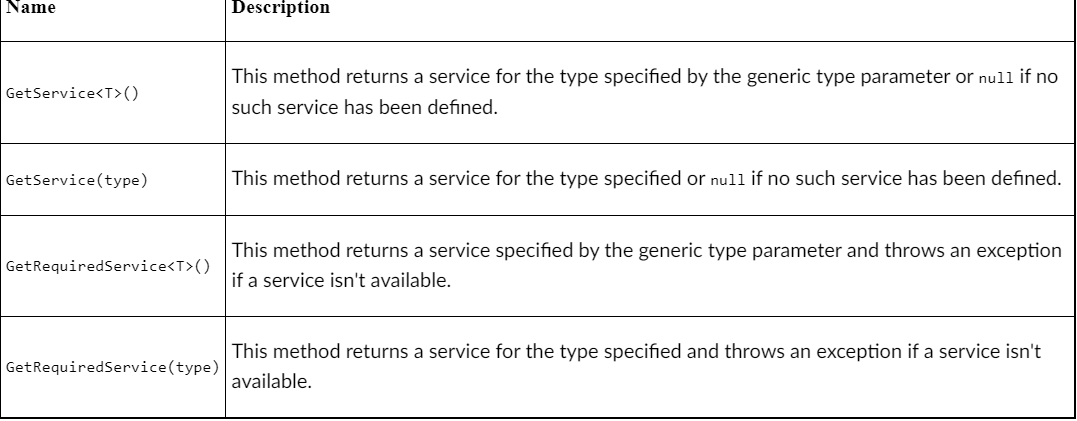
* 
* CatchAll: 

This catches all segments are {second}



**Dependency Injection**

* Microsoft.Extensions.DependencyInjection 🡪 HttpContext.RequestServices property returns IServiceProvider which provides access to all services configured in Start.ConfigureServices method()



* T endpointInstance = ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);

