**We will be discussing in detail**

* ASP.NET Core
* ASP.NET Core MVC
* ASP.NET Identity Core for security
* Entity Framework Core for data access

**What is ASP.NET Core and what are it's benefits**  
ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, Internet-connected applications.  
  
ASP.NET Core is a redesign of ASP.NET 4.x. For this reason it was initially, called ASP.NET 5, but later renamed to ASP.NET Core 1.0.  
  
ASP.NET Core has been redesigned from the ground up and offers the following benefits and features.  
  
**Cross Platform :**ASP.NET 4.x applications run only on windows platform, where as ASP.NET Core applications can be developed and run across different platforms like Windows, macOS, or Linux. ASP.NET 4.x applications can be hosted only on IIS, where as ASP.NET Core applications can be hosted on IIS, Apache, Docker, or even self-host in your own process. From a development standpoint, you can either use Visual Studio or Visual Studio Code for building .NET Core applications. You can also use third party editors like Sublime.  
  
**Unified Programming Model for MVC and Web API :** With ASP.NET core, we use the same unified programming model to create MVC style web applications and ASP.NET Web API's. In both the cases, the Controller that we create inherits from the same Controller base class and returns IActionResult. As the name implies IActionResult is an interface and it has got several implementations. ViewResult and JsonResult are just 2 examples of the built-in result types that implement IActionResult interface. So, in the case of a Web API, the controller returns a JsonResult and in the case of an MVC style web application it returns a ViewResult. If this does not make much sense at the moment, do not worry, it will be crystal clear as we progress through the course.

**Dependency Injection :**Out of the box, ASP.NET Core has built-in support for dependency injection. If you are new to this powerful concept, please do not worry, we will discuss it in detail as we progress through this couse.  
  
**Testability :** With built-in dependency injection and the unified programming model for creating Web Applications and Web API's, unit testing ASP.NET Core applications is easy.  
  
**Open-source and community-focused :**ASP.NET Core is fully open source and is being actively developed by the .NET team in collaboration with a vast community of open source developers. So, ASP.NET core is continually evolving as the vast community behind it is suggesting ways to improve it and help fix bugs and problems. This means we have a more secure and better quality software.  
  
**Modular HTTP Request Pipeline :** ASP.NET Core Provides Modularity with Middleware Components. In ASP.NET Core, we compose the request and response pipeline using the middleware components. It includes a rich set of built-in middleware components. We can also write our own custom middleware components. As we progress through the course we will be discussing, what middleware components are and using them to compose request and response pipeline.  
  
**Course prerequisites:**

* Basic HTML, CSS and C#. You do not have to be an expert by any means. All you need is the basic knowledge.
* Prior MVC knowledge is helpful but not required. Even if you do not have any experience with MVC, that's OK. We will be discussing all the required MVC basics in this course.

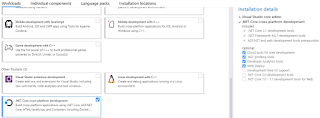
**There are 2 softwares that we need to install for .net core application development.**

* An Editor
* .NET Core SDK (Software Development Kit)

**Download and Install an editor for .NET Core Development**  
  
I have windows operating system, so, I will be using **Visual Studio**as the editor for .NET Core application development. You can use any editor of your choice. 

* Visual Studio
* Visual Studio Code
* Sublime
* Vim
* Atom
* Etc.

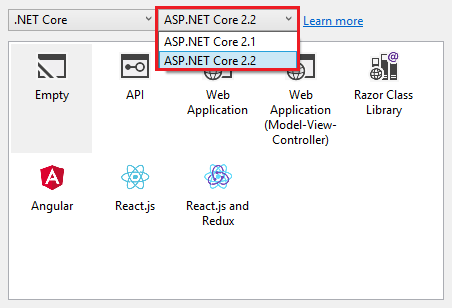
**Download and install Visual Studio**

* Visual Studio Community Edition is free
* As of this recording the latest version is Visual Studio 2017 and can be downloaded from <https://visualstudio.microsoft.com/>
* To develop .NET Core applications in Visual Studio 2017, please select .NET Core cross-platform development workload  
  [](https://4.bp.blogspot.com/-XzaLWj9hH28/XCiKMPpEe-I/AAAAAAAArkY/FrKGv27jMz8MBnSPiSPjYkc_h_e4FSL1ACLcBGAs/s1600/asp.net%2Bcore%2Bsoftware%2Brequirements.png)
* With this selection, .NET Core SDK 2.1 is installed
* You can verify this by creating a new .NET Core Application
* Notice, in the .NET Core Dropdownlist we only have ASP.NET Core 2.1
* As of this recording, the latest stable version of .NET Core is 2.2

**Download and install .NET Core SDK**

* Navigate to <https://dotnet.microsoft.com/download>
* Depending on the operating system you have, download and install the SDK.
* The SDK includes everything you need to build and run .NET core applications.
* The .NET Core Runtime is already included in the SDK. So, if you have installed the SDK, there is no need to install .NET Core Runtime
* The .NET Core Runtime includes just the resources required to run existing .NET Core applications.

After .NET Core SDK 2.2 is installed, create a new .NET Core Application. Notice, in the .NET Core Dropdownlist, we now have ASP.NET Core 2.2 in addition to ASP.NET Core 2.1.

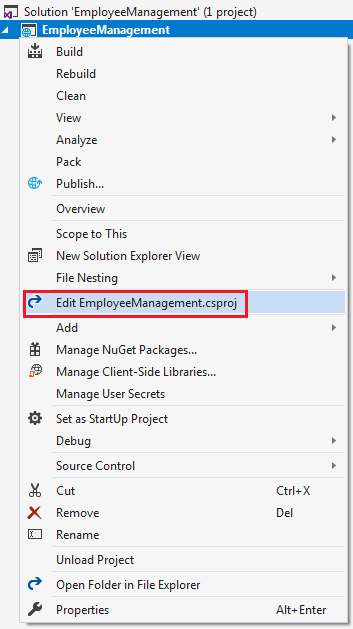


<https://csharp-video-tutorials.blogspot.com/2019/01/creating-aspnet-core-web-application.html>

**Explore and understand the asp.net core project file.**

We are using **C#** as the programming language, so the project file has **.csproj**extension. If you use **Visual Basic**as the programming language, then the project file extension is **.vbproj**. If you have worked with previous versions of ASP.NET, then this file might be very familiar to you, but the format and content that is included in this file has changed significantly in asp.net core.

One significant change is, **the project file does not contain any folder or file reference.** Let me explain what I mean. In previous versions of ASP.NET, when we add a file or folder to the project using solution explorer, **a reference to that file or folder is included in the project file**. In ASP.NET core, the project file does not contain any folder or file reference.  
  
**The File System determines what files and folders belong to the project.** All files and folders that are present in the project root folder are part of the project and will be displayed in the solution explorer. When you add a file or folder using the File Explorer, that file or folder is part of the project and will be immediately displayed in the solution explorer. Similarly when you delete a file or folder from the Project folder or any of it's sub folders, that deleted file or folder is no longer part of the project and that fact is immediately reflected in the Solution Explorer.  
  
Also **the way we work with this file has changed**. In previous versions of asp.net to be able to edit the project file we first have to unload the project, edit and save the project file and then reload the project. Where as in asp.net core, we can edit the project file without unloading the project.  
  
In the Solution Explorer, right click on the project Name and select **"Edit EmployeeManagement.csproj"**



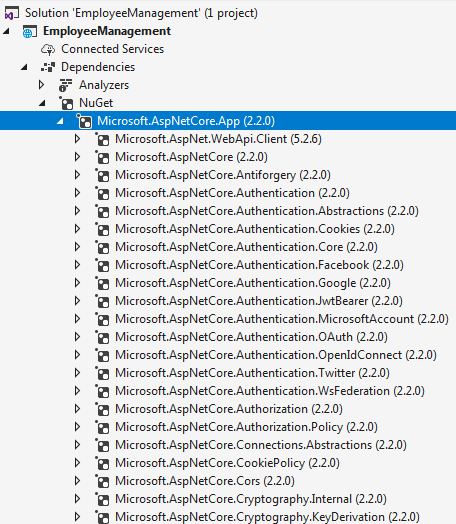
This opens the **.csproj**file in the editor.

**TargetFramework :**As the name implies this element is used to specify the target framework for your application i.e the set of APIs that you'd like to make available to the application. To specify a target framework we use something called **Target Framework Moniker (TFM)**. As you can see in the above example, our application targets one frameowrk **netcoreapp2.2**. netcoreapp2.2 is the Moniker for .NET Core 2.2. When we created this application, we selected .NET Core 2.2 as the target framework from the New Project Dialog dropdown list.  
  
**AspNetCoreHostingModel :**This element specifies how the asp.net core application should be hosted. Should it be hosted **InProcess**or **OutOfProcess**. The value of InProcess specifies that we want to use in-process hosting model i.e host our asp.net core app inside of the IIS worker process (**w3wp.exe**). The value of OutOfProcess specifies that we want to use  out-of-process hosting model i.e forward web requests to a back-end ASP.NET Core app running the **Kestrel**server. We will discuss InProcess and OutOfProcess hosting in detail in our upcoming videos.

**PackageReference :** As the name implies, this element is used to include a reference to all the NuGet packages that are installed for your application. At the moment in the project file we have the following 2 NuGet packages.

* Microsoft.AspNetCore.App
* Microsoft.AspNetCore.Razor.Design

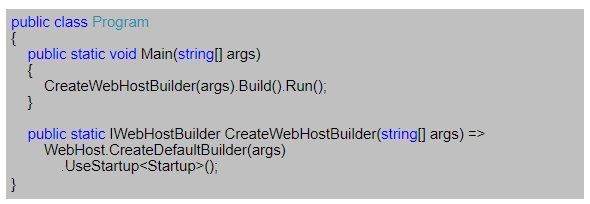
**Microsoft.AspNetCore.App :** This NuGet package is called **metapackage**. A metapackage has no content of its own but is a list of dependencies (other packages). You can find this metapackage, in the Solution Explorer, under **NuGet**which in turn is under **Dependencies**. When you expand the metapackage, you can find all the dependencies.



All the features of ASP.NET Core 2.1 and later and Entity Framework Core 2.1 and later are included in this **Microsoft.AspNetCore.App**package. The default project templates targeting ASP.NET Core 2.1 and later use this package.  
  
**Notice there is no version number on the metapackage.** When the version is not specified, an**implicit version**is specified by the SDK. The .NET Core team recommends relying on the implicit version specified by the SDK and not explicitly setting the version number on the package reference. Don't worry if this is not entirely clear at the moment. We will discuss metapackage and implicit version in detail in our upcoming videos.  
  
**Microsoft.AspNetCore.Razor.Design :**This package contains MSBuild support for Razor and is referenced by Microsoft.AspNetCore.App meta package.

**Main() method in an ASP.NET Core application**

In an ASP.NET Core project we have a file with name **Program.cs**. In this file we have a **public static void Main()** method



If you have any experience with previous versions of .NET, a console application has a **Main()** method and it is the entry point for that console application.  
  
But here, we are creating an **asp.net core web application** and not a console application. So the obvious question that comes to our mind is why do we have a Main() method.  
  
Well, the important point to keep in mind is that, an **asp.net core application initially starts as a console application** and the **Main()** method in **Program.cs** file is the entry point.   
  
So, when the runtime executes our application **it looks for this Main() method**and this where the execution starts.  
  
This **Main() method configures asp.net core**and starts it and at that point it becomes an asp.net core web application.  
  
So, if you take a look at the **Main()** method, it calls CreateWebHostBuilder() method passing it the command line arguments.  
  
As you can see, CreateWebHostBuilder() method returns an object that implements IWebHostBuilder.  
  
On this object, **Build()** method is called which builds a web host that hosts our asp.net core web application.  
  
On the web host **Run()**method is called, which runs the web application and it begins listening for incoming HTTP requests.  
  
**CreateWebHostBuilder()** method calls CreateDefaultBuilder() static method of the WebHost class.  
  
CreateDefaultBuilder() method creates a web host with pre-configured defaults. CreateDefaultBuilder() method does several things to create a web host. We will discuss all that the CreateDefaultBuilder() method does in detail in our next video. For now, just understand that the CreateDefaultBuilder() method **sets up a web host with certain defaults**.  
  
As part of setting up a web host, Startup class is also configured using the UseStartup() extension method of IWebHostBuilder class. If you are new to the concept of extension methods, please check out the following video.

[**Extension Methods in C#**](https://www.youtube.com/watch?v=VkrKNXscoto)  
  
By convention, the startup class in ASP.NET Core is named Startup. This class has 2 methods.



* **In process hosting model in ASP.NET Core**
* **What is Kestrel server**

When an ASP.NET core application is executed, the .NET runtime looks for Main() method which is the entry point for the application. The Main() method then calls CreateDefaultBuilder() static method of the WebHost class.   
  
  
This CreateDefaultBuilder() method performs several tasks like 

* Setting up the web server
* Loading the host and application configuration from various configuration sources and
* Configuring logging

We will discuss the various configuration sources available in asp.net core, Loading the host and application configuration information and configuring logging in our upcoming videos.  
  
In this video, let's understand what the CreateDefaultBuilder() method does to configure and set up the web server. An ASP.NET core application can be hosted InProcess or OutOfProcess.  
  
In this video, we will discuss **InProcess hosting**and in our next video we will discuss **OutOfProcess hosting**.  
  
**InProcess hosting in ASP.NET Core**  
To configure InProcess hosting, add <AspNetCoreHostingModel> element to the app's project file with a value of InProcess  
**<AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>**

When we create a new ASP.NET Core project using one of the available project templates, the project defaults to the in-process hosting model for all IIS and IIS Express scenarios.  
  
In case of InProcess hosting, CreateDefaultBuilder() method calls UseIIS() method and host the app inside of the IIS worker process (w3wp.exe or iisexpress.exe). 

* From a **performance standpoint**, InProcess hosting delivers significantly higher request throughput than OutOfProcess hosting
* In the case of IIS, the process name that executes the app is **w3wp**and in the case of IIS Express it is **iisexpress**
* To get the process name executing the app, use System.Diagnostics.Process.GetCurrentProcess().ProcessName
* When we are run the project from Visual Studio it uses IISExpress by default.
* IIS Express is a lightweight, self-contained version of IIS, optimized for application development. We do not use it for production. In production we use IIS.
* We will discuss deploying ASP.NET Core applications on IIS in our upcoming videos.

**With out of process hosting**

* There are 2 web servers - An **internal web server** and an **external web server**. We will discuss out of process hosting in detail in our next video.
* The internal web server is **Kestrel**and the external web server can be IIS, Nginx or Apache.
* With **InProcess hosting**, there is only **one web server**i.e the IIS that hosts the asp.net core application.
* So, **we do not have the performance penalty**of proxying requests between internal and external web server.

**What is Kestrel**  
Kestrel is a cross-platform web server for ASP.NET Core. It is supported on all platforms and versions that .NET Core supports. It is included by default as internal server in ASP.NET Core. Kestrel can be used, by itself as an edge server i.e Internet-facing web server that can directly process the incoming HTTP requests from the client. In Kestrel, the process used to host the app is **dotnet.exe.**  
  
When we run a .NET Core application using the .NET Core CLI (Command-Line Interface), the application uses **Kestrel as the web server**.

**The .NET Core CLI is a cross-platform tool** for developing .NET core applications. Using the CLI we can

* Create a new project, configuration file, or solution based on the specified template
* Restore the dependencies and tools required for a .net core project
* Build a project and all of its dependencies
* Run a project etc...

There are a broad range of things that we can do with the .NET Core CLI. **To run our asp.net core application using the .NET Core CLI.**

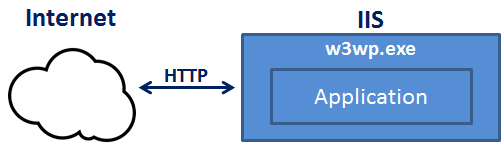
* Fire up Windows Command Prompt
* Change the directory to the folder that contains your asp.net core project and execute dotnet run command
* C:\Projects\EmployeeManagement\EmployeeManagement>**dotnet run**

After the .NET Core CLI builds and runs the project, it shows the URL using which we can access the application. In my case the application is available at http://localhost:5000  
  
In case of Kestrel, the process used to host and execute the app is **dotnet.exe**. So when we navigate to **http://localhost:5000**, we will see the process name **dotnet**displayed.

**Out Of Process Hosting in ASP.NET Core**

**InProcess Hosting in ASP.NET Core**  
  
To configure InProcess hosting, add <AspNetCoreHostingModel> element to the app's project file with a value of InProcess

<AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>

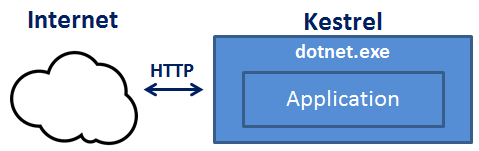
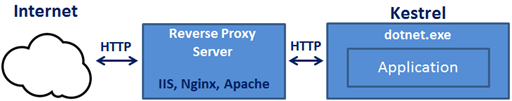
With **InProcess hosting**, the application is hosted in the IIS worker process (w3wp.exe or iisexpress.exe). With InProcess hosting, there is only one web server and that is the IIS server that hosts our application.  
  
  
  
We discussed InProcess hosting in detail in [Part 6](https://www.youtube.com/watch?v=ydR2jd3ZaEA) of [ASP.NET Core Tutorial](https://www.youtube.com/playlist?list=PL6n9fhu94yhVkdrusLaQsfERmL_Jh4XmU).  
  
**Out of Process Hosting in ASP.NET Core**  
  
There are 2 ways to configure **Out of Process hosting**  
  
**Option 1 :** Add <AspNetCoreHostingModel> element to the app's project file with a value of OutOfProcess

<AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>

**Option 2 :** The default is OutOfProcess hosting. So if we remove the <AspNetCoreHostingModel> element from the project file, OutOfProcess hosting will be used by default.  
  
With **out of process hosting**

* There are 2 web servers - An an internal web server and an external web server.
* The internal web server is Kestrel and the external web server can be IIS, Nginx or Apache. We discussed Kestrel in detail in Part 6 of ASP.NET Core Tutorial.

Depending on how you are running the asp.net core application, the external web server may or may not be used.   
  
**Kestrel is a cross-platform web server** that is embedded in your ASP.NET Core application. With Out of Process Hosting model, Kestrel can be used in one of the following 2 ways.

**Kestrel can be used as the internet facing web server** that process the incoming HTTP requests directly. In this model we are not using an external web server. Only Kestrel is used and it is this server that faces the internet, to directly handle the incoming HTTP requests. When we run the asp.net core application using the .NET core CLI, Kestrel is the only web server that is used to handle and process the incoming HTTP request.  
  
  
  
**Kestrel can also be used in combination with a reverse proxy server**, such as IIS, Nginx, or Apache.  
  
  
  
**If Kestrel can be used by itself as a web server, why do we need a reverse proxy server.**  
With **Out of Process Hosting**, using a reverse proxy server is a good choice as it provides an additional layer of configuration and security. It might integrate better with the existing infrastructure. It can also be used for load balancing.

So, with a reverse proxy server in place, it receives incoming HTTP requests from the network and forwards them to the Kestrel server for processing. Upon processing the request, the Kestrel server sends the response to the reverse proxy server which then ultimately sends the response to the requested client over the network.  
  
We will discuss **Deploying ASP.NET Core application to IIS**and using IIS as a reverse proxy server in our upcoming videos. When we run an asp.net core application directly from Visual Studio it uses by default IIS Express. Since we have configured our application to use Out of Process hosting, IIS Express in this case acts a reverse proxy server.  
  
**IIS Express** takes the incoming HTTP request and forwards it to Kestrel for processing. Kestrel process the request and sends the response to IIS Express. IIS Express, in turn sends that response to the browser.  
  
With **Out of Process Hosting**, whether you use a reverse proxy server or not, it is the Kestrel server that hosts the application and process the request. The reverse proxy server if used, takes the incoming HTTP request and forwards it to the Kestrel server. Similarly it takes the response from the Kestrel server and sends it to the client. So the name of the process that hosts the application is dotnet.exe.  
  
**Use the following code to get the process name**

System.Diagnostics.Process.GetCurrentProcess().ProcessName

When we run the asp.net core project using the .NET Core CLI, by default it ignores the hosting setting we specified in the csproj file. So the AspNetCoreHostingModel element value in the **csproj file**is ignored.   
  
Irrespective of the value you specified (InProcess or OutOfProcess), it always uses OutOfProcess hosting and Kestrel is the web server that hosts the application and handle the http requests.  
  
**One common question :**Can we run an asp.net core application without using the built in kestrel web server.  
The answer is YES. If we use the InProcess hosting model, the application is hosted inside of the IIS worker process (w3wp.exe or iisexpress.exe). Kestrel is not used with InProcess hosting model.

**launchsettings.json file in ASP.NET Core project**.

**launchsettings.json file**

* You will find this file in the **"Properties"** folder in the project root folder.
* The settings in this file are used when we run this ASP.NET core project either from Visual Studio or by using .NET Core CLI.
* This file is **only used on local development machine**. We do not need it for publishing our asp.net core application.
* If there are certain settings that you want your asp.net core application to use when you publish and deploy your app, store them in **appsettings.json file**. We usually store our application configuration settings in this file.
* We can also have **environment specific appsettings.json files**. For example, appsettings.Staging.json for the staging environment. In ASP.NET Core, in addition to appsettings.json file, we also have other configuration sources like Environment variables, User Secrets, Command Line Arguments and even our own custom configuration source.
* More on these different configuration sources and appsettings.json file in our next video.

**Lanuch Profiles in ASP.NET Core**  
  
At the moment, the following are the settings in **launchSettings.json file**

{

  "iisSettings": {

    "windowsAuthentication": false,

    "anonymousAuthentication": true,

    "iisExpress": {

      "applicationUrl": "http://localhost:48118",

      "sslPort": 0

    }

  },

  "profiles": {

    "IIS Express": {

      "commandName": "IISExpress",

      "launchBrowser": true,

      "environmentVariables": {

        "ASPNETCORE\_ENVIRONMENT": "Development"

      }

    },

    "EmployeeManagement": {

      "commandName": "Project",

      "launchBrowser": true,

      "environmentVariables": {

        "ASPNETCORE\_ENVIRONMENT": "Development"

      },

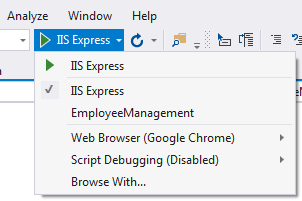
      "applicationUrl": "http://localhost:5000"

    }

  }

}

Notice, we have 2 profiles - **IIS Express**and **EmployeeManagement**  
  
When we run the project from Visual Studio by pressing **CTRL + F5** or just **F5**, by default, the profile with "commandName": "IISExpress" is used. On the other hand, if we run the project using .NET Core CLI (dotnet run), the profile with the  "commandName": "Project" is used.

However, we can change which profile to use by clicking on the dropdownlist in Visual Studio  
  
  
  
The value of the **commandName**property can be any one of the following. 

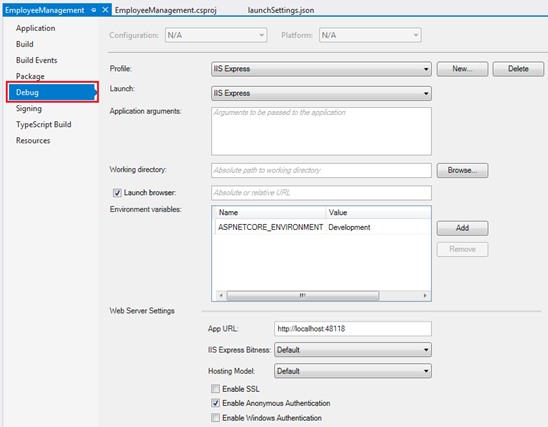
* Project
* IISExpress
* IIS

This value along with the value of **AspNetCoreHostingModel**element in the project file, specifies the internal and external web server (reverse proxy server) to launch.

|  |  |  |  |
| --- | --- | --- | --- |
| **commandName** | **AspNetCoreHostingModel** | **Internal Web Server** | **External Web Server** |
| Project | Hosting Setting Ignored | Only one web server is used - Kestrel | |
| IISExpress | InProcess | Only one web server is used - IIS Express | |
| IISExpress | OutOfProcess | Kestrel | IIS Express |
| IIS | InProcess | Only one web server is used - IIS | |
| IIS | OutOfProcess | Kestrel | IIS |

You can change the settings in **launchSettings.json**file by directly editing the file or we can also change the settings using the Graphical User Interface (GUI) provided by Visual Studio.   
  
**To access the GUI**

* Right click on the project name in Solution Explorer in Visual Studio and select **"Properties"** from the context menu.
* Click on the **"Debug"**tab on the project **"Properties"** window

  
  
Using the GUI we can change the settings in **launchSettings.json file**. Notice, the Environment Variable **"ASPNETCORE\_ENVIRONMENT"** is set to **"Development"**. We can change this value to **Staging**or **Production**depending on whether we are running this project in Staging or Production environment.   
  
We can also add new environment Variables. These environment variables are available throughout our asp.net core application and we can include code that conditionally executes depending on the value of these environment variables.

**For example**, consider the following code in the **Configure()** method in **Startup.cs** file

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

    if (env.IsDevelopment())

    {

        app.UseDeveloperExceptionPage();

    }

    // Rest of the code...

}

**Developer Exception Page** is only displayed if the environment is **Development**. We will discuss environment variables and developer exception page in detail in our upcoming videos.

**appsettings.json file in ASP.NET Core project**.

**Configuration Sources in ASP.NET Core**  
  
In previous versions of ASP.NET, we store application configuration settings, like database connection strings for example, in **web.config**file. In ASP.NET Core application configuration settings can come from the following **different configurations sources.**

* Files (appsettings.json, appsettings.{Environment}.json, where {Environment} is the app's current hosting environment)
* User secrets
* Environment variables
* Command-line arguments

**appsettings.json file :**In the asp.net core project that is generated by the **"Empty"** project template we already have a file with name **appsettings.json**. I have modified this file to include a new setting with the key - **MyKey**.

{

  "Logging": {

    "LogLevel": {

      "Default": "Warning"

    }

  },

  "AllowedHosts": "\*",

  "MyKey": "Value of MyKey from appsettings.json"

}

**Accessing configuration information**  
  
To access configuration information in the **Startup**class, inject the IConfiguration service provided by the Framework. Startup class is in Startup.cs file.

public class Startup

{

    private IConfiguration \_configuration;

    // Notice we are using Dependency Injection here

    public Startup(IConfiguration configuration)

    {

        \_configuration = configuration;

    }

    public void ConfigureServices(IServiceCollection services)   {    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)

    {

        if (env.IsDevelopment())

        {

            app.UseDeveloperExceptionPage();

        }

        app.Run(async (context) =>

        {

            await context.Response.WriteAsync(\_configuration["MyKey"]);

        });

    }

}

**Dependency Injection**  
  
In previous versions of ASP.NET Dependency Injection was optional and to configure it we have to use frameworks like Ninject, StructureMap etc.**In ASP.NET Core Dependency Injection is an integral part.** Dependency Injection allow us to create systems that are loosely coupled, extensible and easily testable. We will discuss Dependency Injection in detail in our upcoming videos.  
  
**ASP.NET Core IConfiguration service**

* IConfiguration service is setup to read configuration information from all the various configuration sources in asp.net core
* If you have a configuration setting with the**same key in multiple configuration sources**, the later configuration sources override the earlier configuration sources
* CreateDefaultBuilder() method of the WebHost class which is automatically invoked when the application starts, reads the configuration sources in a specific order.
* To see the order in which the configuration sources are read, please check out ConfigureAppConfiguration() method on the following link  
  <https://github.com/aspnet/MetaPackages/blob/release/2.2/src/Microsoft.AspNetCore/WebHost.cs>

Upon inspecting the file, you will see, the following is the default order in which the various configuration sources are read

1. appsettings.json,
2. appsettings.{Environment}.json
3. User secrets
4. Environment variables
5. Command-line arguments

You can change this order if you want to or even add your own custom configuration sources in addition to all the existing configuration sources. We will discuss setting up custom configuration source in our upcoming videos.

**What is Middleware in ASP.NET Core**

In ASP.NET Core, Middleware is a piece of software that can handle an HTTP request or response. A given middleware component in ASP.NET Core has a very specific purpose. For example we may have a middleware component that authenticates a user, another piece of middleware to handle errors, yet another middleware to serve static files such as JavaScript files, CSS files, Images etc.

It is these middleware components that we use to setup a request processing pipeline in ASP.NET Core. It is this pipeline that determines how a request is processed. The request pipeline is configured as part of the application startup by the Configure() method in Startup.cs file.  The following is the code in Configure() method.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

    if (env.IsDevelopment())

    {

        app.UseDeveloperExceptionPage();

    }

    app.Run(async (context) =>

    {

        await context.Response.WriteAsync("Hello World!");

    });

}

As you can see, the code in the Configure() method that is generated by the empty project template sets up a **very simple request processing pipeline**with just two pieces of middleware.  
  
UseDeveloperExceptionPage is one middleware and the second middleware is setup using the Run() method. As it stands right now with this very simple request processing pipeline, all our application can do is write a message to the response object that will be displayed by the browser. We will come back and understand the code in this method in detail in our next video. For now let’s understand what is middleware and how it works in asp.net core.  
  
The following diagram helps us understand middleware components and how they fit in a request processing pipeline



In ASP.NET Core, **a Middleware component has access to both - the incoming request and the outgoing response.** So a Middleware component may process an incoming request and pass that request to the next piece of middleware in the pipeline for further processing. For example, if you have a **logging middleware**, it might simply log the time the request is made and pass the request to the next piece of middleware for further processing.  
  
**A middleware component may handle the request and decide not to call the next middleware in the pipeline.** This is called short-circuiting the request pipeline. Short-circuiting is often desirable because it avoids unnecessary work. For example, if the request is for a static file like an image or css file, the StaticFiles middleware can handle and serve that request and short-circuit the rest of the pipeline. This means in our case, the StaticFiles middleware will not call the MVC middleware if the request is for a static file.  
  
A middleware component may handle an incoming HTTP request by generating an HTTP response. For example, **mvcmiddleware**in the pipeline handles a request to the URL **/employees**and returns a list of employees. As we progress through this course, in our upcoming videos we will be including the **mvcmiddleware**in the request processing pipeline of our application.  
  
**A middleware component may also process the outgoing response**. For example, the logging middleware component may log the time the response is sent. In addition it may also calculate the over all time taken to process the request by computing the difference between request received and response sent times.  
  
**Middleware components are executed in the order they are added to the pipeline**. Care should be taken to add the middleware in the right order, otherwise the application may not function as expected. In our upcoming videos, we will discuss with an example, what happens if the middleware components are not added to the processing pipeline in the correct order.  
  
**The middleware components are available as NuGet packages**. This means updates are now handled by NuGet, providing the ability to update each middleware separately.  
  
Depending on your application requirements you may add as many or as few middleware components to the request processing pipeline. For example, if you are developing simple web application with a few static HTML pages and images, then your request processing pipeline may contain just **"StaticFiles" middleware**.  
  
On the other hand, if you are developing a secure data driven web application then you may need several middleware components like StaticFiles middleware, Authentication middleware, Authorization middleware, MVC middleware etc.   
  
The point that I am trying to make is, you have complete control over configuring the request processing pipeline. This also means from a memory and performance standpoint you only pay for the middleware components you have in your request processing pipeline.  
  
Now that we have a basic understanding of what middleware components are and how they fit in a request processing pipeline, in our next video, we will understand, how to configure a request processing pipeline for our ASP.NET Core application using middleware components.

**configuring a request processing pipleline for asp.net core application** using the middleware components

As part of the application startup, **Configure()** method sets up the request processing pipeline.

public class Startup

{

    public void ConfigureServices(IServiceCollection services)

    {

    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)

    {

        if (env.IsDevelopment())

        {

            app.UseDeveloperExceptionPage();

        }

        app.Run(async (context) =>

        {

            await context.Response.WriteAsync("Hello World!");

        });

    }

}

**At the moment we have 2 middlewares in the pipeline**

* UseDeveloperExceptionPage() and
* another middleware that is registered using the Run() method.

**UseDeveloperExceptionPage Middleware :** As the name implies, this middleware responds with the developer exception page, if there is an exception and if the environment is Development. We will discuss this DeveloperExceptionPage middleware and the  different environments in our upcoming videos.  
  
The second middleware that is registered using the Run() method, can only write a message to the Response object. At the moment, this is the middleware that responds to every request.   
Doesn’t matter what your request path is. All requests will be handled by this one piece of middleware and the response we get is the string message that the middleware is writing to the Response object. The response is plain text and not html. We can confirm this by inspecting the page source. Notice we do not have any html tags in the source. It’s just, plain text.  
  
Even if you have a file with name **foo.html**and if you include the path to that file in the request, our application will not be able to serve that static file. This is because, at the moment our request processing pipeline does not have the middleware that can serve static files like html files, images, css and JavaScript files. As we progress through this course we will add the required middleware to be able to serve static files.  
  
**Consider the following code in the Configure() method.**

app.Run(async (context) =>  
{  
    await context.Response.WriteAsync("Hello World!");  
});

**Code Explanation**

* We are using Run() method to add middleware to our application's request processing pipeline
* If you hover the mouse over the Run() method, from the intellisense you can see that this Run() method is implemented as an extension method of IApplicationBuilder interface. This is the reason we are able to invoke this Run() method on IApplicationBuilder object app.
* The parameter that we are passing to the Run() method is a RequestDelegate which we can see from the intellisense.
* RequestDelegate is a delegate that has HttpContext object as a parameter.
* It is through this HttpContext object, the middleware gains access to both the incoming http request and outgoing http response.
* At the moment, we are passing request delegate inline as an anonymous method using a lambda. If you are new to the concept of delegates and lambda, please check out the following 3 videos from our [C# tutorial](https://www.youtube.com/playlist?list=PLAC325451207E3105).
  + [C# Delegates](https://www.youtube.com/watch?v=D2h46fvQX04)
  + [C# Delegates Usage](https://www.youtube.com/watch?v=vBOzvNO8lvk)
  + [Lamda Expression in C#](https://www.youtube.com/watch?v=LDgQ-spnrYY)
* Instead of passing the request delegate inline as an anonymous method, we can define the request delegate in a separate reusable class.
* With this Run() extension method we can only add a terminal middleware to the request pipeline.
* A terminal middleware is a middleware that does not call the next middleware in the pipeline

**Consider the following code**

app.Run(async (context) =>  
{  
    await context.Response.WriteAsync("Hello from 1st Middleware");  
});  
  
app.Run(async (context) =>  
{  
    await context.Response.WriteAsync("Hello from 2nd Middleware");  
});

* We have 2 middlewares registered using the Run() method
* Upon running this project, we only see the response from the first middleware
* We do not see the response from the second middleware
* This is because, a middleware that is registered using the Run() method cannot call the next middleware in the pipeline
* So, the middleware that we register using Run() method is a terminal middleware

If you want your middleware to be able to call the next middleware in the pipeline, then register the middleware using **Use()** method as shown below.

app.Use(async (context, next) =>  
{  
    await context.Response.WriteAsync("Hello from 1st Middleware");  
    await next();  
});  
  
app.Run(async (context) =>  
{  
    await context.Response.WriteAsync("Hello from 2nd Middleware");  
});

Notice, **Use()** method has 2 parameters. The first parameter is the HttpContext context object and the second parameter is the Func type i.e it is a generic delegate that represents the next middleware in the pipeline.  
  
Now, consider the following code

public void Configure(IApplicationBuilder app, IHostingEnvironment env,  
                ILogger<Startup> logger)  
{  
    app.Use(async (context, next) =>  
    {  
        logger.LogInformation("MW1: Incoming Request");  
        await next();  
        logger.LogInformation("MW1: Outgoing Response");  
    });  
  
    app.Use(async (context, next) =>  
    {  
        logger.LogInformation("MW2: Incoming Request");  
        await next();  
        logger.LogInformation("MW2: Outgoing Response");  
    });  
  
    app.Run(async (context) =>  
    {  
        await context.Response.WriteAsync("MW3: Request handled and response produced");  
        logger.LogInformation("MW3: Request handled and response produced");  
    });

* ILogger<Startup> is injected into the Configure() method
* CreateDefaultBuilder() that is called by the Main() method configures logging
* You can verify this on their GitHub page using the link below  
  <https://github.com/aspnet/MetaPackages/blob/release/2.2/src/Microsoft.AspNetCore/WebHost.cs>
* Please check the method ConfigureLogging()
* You will find that, loggers for Console, Debug and EventSource are configured
* We are using the logger instance provided by the Dependency Injection system to log the information
* If you run the project using the .NET Core CLI, you can see the logged information on the Console window
* If you run the project directly from Visual Studio, you can see the logged information in the output window. Select ASP.NET Core Web Server from the dropdownlist in the output window.
* You will see that, the information is logged in the following order
  + **MW1: Incoming Request**
  + **MW2: Incoming Request**
  + **MW3: Request handled and response produced**
  + **MW2: Outgoing Response**
  + **MW1: Outgoing Response**

Now relate the above output, with the following diagram from MSDN to understand what's happening.



* Remember a middleware in asp.net core has access to both the incoming request and the outgoing response
* The request first arrives at Middleware1 which logs (MW1: Incoming Request) so we see this message first.
* Then Middleware1 calls next(). Calling next() invokes Middleware2 in the pipeline.
* Middleware2 logs (MW2: Incoming Request). So we see (MW2: Incoming Request) after (MW1: Incoming Request).
* Then Middleware2 calls next() which invokes Middleware3.
* Middleware3 handles the request and produces a response. Hence, the next message that we see is (MW3: Request handled and response produced)
* So, at this point the pipeline starts to reverse itself.
* The control is then given to Middleware2 and the response produced by Middleware3 is passed to it. Middleware2 then logs (MW2: Outgoing Response) which is what we see next.
* Finally Middleware2 gives control to Middleware1.
* Middleware1 logs (MW1: Outgoing Response) which is what we see finally.

**So here are the 3 very important points to keep in mind regarding the request processing pipeline**

* Everything that happens before the next() method is invoked in each of the middleware components, happen as the request travels from middleware to middleware through the pipeline and this is represented by the **incoming arrow**.
* When a middleware handles the request and produces response, the request processing **pipeline starts to reverse**.
* Everything that happens after the next() method is invoked in a middleware component, happens as the response travels from middleware to middleware through the pipeline and this is represented by the **outgoing arrow**.

**how to make ASP.NET Core application serve static files such as HTML, Images, CSS and JavaScript files**

**Static Files**

* By default, an asp.net core application will not serve static files
* The default directory for static files is wwwroot and this directory must be in the root project folder
* Copy and paste an image in **wwwroot**folder. let's assume the name of the file is banner.jpg. To be able to access this file from the browser we use  
    
  http://{{serverName}}/banner.jpg  
    
  In our case we are running on our local machine so the URL would be the following. The port number may be different on your machine.  
    
  http://localhost:49119/banner.jpg  
    
  On my machine when I navigate to the above URL, I still see the response produced by the middleware I have registered using the Run() method. I do not see the image banner.jpg.  
    
  This is because, at the moment our application request processing pipeline does not have the required middleware that can serve static files. The middleware that we need is UseStaticFiles() middleware.   
    
  Modify the code in Configure() method to add UseStaticFiles() middleware to our application's request processing pipeline as shown below.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)  
{  
    if (env.IsDevelopment())  
    {  
        app.UseDeveloperExceptionPage();  
    }  
  
    // Add Static Files Middleware  
    app.UseStaticFiles();  
  
    app.Run(async (context) =>  
    {  
        await context.Response.WriteAsync("Hello World!");  
    });  
}

Instead of having all files types like images, css and JavaScript files flat in the wwwroot folder, it is common to have separate folders for css, images and JavaScript under wwwroot as shown below. Consider the following folder hierarchy.



To be able to access **image1.jpg** from the browser we use

**Serving static files outside of wwwroot folder**  
  
By default, UseStaticFiles() middleware only serves the static files that are in wwwroot folder. We can also server static files outside of the wwwroot folder if you want to.  
  
**Serving a default document**  
  
Most web applications have a default document and it is the document that is displayed when a user visits the root URL of your application. For example, you have a file with name default.html and you want to serve it when the user visits the root url of your application i.e http://localhost:49119/  
  
At the moment, on my machine when I navigate to the root URL, I see the response produced by the middleware I have registered using the Run() method. I do not see the content of the default document default.html. To be able to serve default page we have to plug in the UseDefaultFiles() middleware in our application's request processing pipeline.  
  
// Add Default Files Middleware  
app.UseDefaultFiles();  
// Add Static Files Middleware  
app.UseStaticFiles();  
  
**Please Note :**UseDefaultFiles must be called before UseStaticFiles to serve the default file. UseDefaultFiles is a URL rewriter that doesn't actually serve the file. It simply rewrites the URL to the default document which will then be served by the Static Files Middleware. The URL displayed in the address bar still reflects the root URL and not the rewritten URL.  
  
The following are the default files which UseDefaultFiles middleware looks for  
index.htm  
index.html  
default.htm  
default.html  
  
If you want to use another document like **foo.html**for example as your default document, you can do so using the following code.  
  
// Specify foo.html as the default document  
DefaultFilesOptions defaultFilesOptions = new DefaultFilesOptions();  
defaultFilesOptions.DefaultFileNames.Clear();  
defaultFilesOptions.DefaultFileNames.Add("foo.html");  
// Add Default Files Middleware  
app.UseDefaultFiles(defaultFilesOptions);  
// Add Static Files Middleware  
app.UseStaticFiles();  
  
**UseFileServer Middleware**  
  
UseFileServer combines the functionality of UseStaticFiles, UseDefaultFiles and UseDirectoryBrowser middleware. DirectoryBrowser middleware, enables directory browsing and allows users to see files within a specified directory. We could replace UseStaticFiles and UseDefaultFiles middlewares with UseFileServer Middleware.  
  
// Use UseFileServer instead of UseDefaultFiles & UseStaticFiles

FileServerOptions fileServerOptions = new FileServerOptions();  
fileServerOptions.DefaultFilesOptions.DefaultFileNames.Clear();  
fileServerOptions.DefaultFilesOptions.DefaultFileNames.Add("foo.html");  
app.UseFileServer(fileServerOptions);  
  
The important point to note here is the pattern that we use to add middleware to our application's request processing pipeline. In most cases we add middleware using the extension methods that start with the word **USE**. For example,

* UseDeveloperExceptionPage()
* UseDefaultFiles()
* UseStaticFiles()
* UseFileServer()

If you want to customise these middleware components, we use the respective OPTIONS object. For example notice the respective **OPTIONS**objects we use.

|  |  |
| --- | --- |
| **Middleware** | **Options Object** |
| UseDeveloperExceptionPage | DeveloperExceptionPageOptions |
| UseDefaultFiles | DefaultFilesOptions |
| UseStaticFiles | StaticFileOptions |
| UseFileServer | FileServerOptions |