# ASP.NET CORE

## WHAT IS ASP.NET CORE?

* 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

## ASP.NET CORE BENEFITS AND FEATURES



### CROSS PLATFORM:

1. ASP.NET Core applications can be developed and run across different platforms like Windows, MacOS, Linux
2. ASP.NET Core applications can be hosted on IIS, Apache, Docker, Self-host in your own process

### One Unified Programming Model for MVC and Web API:

Both the MVC Controller class and the ASP.NET Web API Controller class inherit from the same 'Controller' base class and returns 'IActionResult'



### Modular:

* ASP.NET Core Provides Modularity with Middleware Components
* Both the request and response pipelines are composed using the middleware components
* Rich set of built-in middleware components are provided out of the box
* Custom Middleware Components can also be created

## ASP.NET Core Project File

### TargetFramework

* Specifies the target framework for the application
* To specify a target framework we use Target Framework Moniker (TFM)



In our case the TFM is **netcoreapp3.1**

### AspNetCoreHostingModel

* Specifies how the application should be hosted
* InProcess or OutOfProcess
* InProcess hosts the app inside of the IIS worker process(w3wp.exe)
* OutOfProcess hosting model forward web requests to a backend ASP.NET Core app running the Kestrel Server

# Main Method in ASP.NET Core

* A Console application usually has a **Main()** method
* Why do we have a **Main()** method in ASP.NET Core web application
* ASP.NET Core application initially starts as a Console application and the **Main()** method of Program.cs class is the entry point



* This **Main()** method configures ASP.NET Core and starts it and at that point it becomes an ASP.NET Core web application

# ASP.NET Core InProcess Hosting

## Some of the Tasks that CreateDefaultBuilder() performs

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

## An ASP.NET core application can be hosted

* InProcess or
* OutOfProcess

### To configure InProcess hosting



#### With InProcess

* Application is hosted inside the IIS worker process
* There is only one web server
* From a performance standpoint, InProcess hosting is better than OutOfProcess hosting



CreateDefaultBuilder() method calls UseIIS() method and host the app inside of the IIS worker process (w3wp.exe or iisexpress.exe)



ASP.NET application when run from CLI and **InProcess** hosting is configured than kestrel is used as the web server as shown below:





InProcess hosting delivers significantly higher request throughput than OutOfProcess hosting

### With OutOfProcess hosting

* 2 Web Servers – Internal and External Web Server
* The internal web server is Kestral
* The external web server can be IIS, Nginx, Apache

## What is Kestral

* Cross-Platform Web Server for ASP.NET Core
* Kestral can be used, by itself as an edge server
* The process used to host the app is dotnet.exe

# ASP.NET Core OutOfProcess Hosting

## To configure OutOfProcess hosting

We can either set AspNetCoreHostingModel to **OutOfProcess** or remove the AspNetCoreHostingModel, application uses OutOfProcess hosting by default



When application is run from CLI and OutOfProcess hosting is configured the Kestral server is used as edge server as shown below



Another way Kestral can be used in combination with a reverse proxy server, where Kestral is not facing the internet, it’s the Reverse Proxy server that takes the incoming http request and forwards it to the Kestral server that is hosting and running our ASP.NET Core application



A **Reverse Proxy Server**  such as IIS, Nginx etc. provides an additional layer of configuration and security, it might integrate better with our existing infrastructure and can also be used for load balancing.



# ASP.NET Core launchsettings.json File



# ASP.NET Core appsettings.json File

## Configurations Sources in ASP.Net Core

* Files(appsettings.json, appsettings.{Environment}.json)
* User secrets
* Environment variables
* Command-line arguments

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

# Middleware 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

## Middleware in ASP.NET Core

* Has access to both Request and Response
* May simply pass the Request to Middleware
* May process and then pass the Request to next Middleware
* May handle the Request and short-circuit the pipeline
* May process the outgoing Response
* Middlewares are executed in the order they are added

**Consider the following code in the Configure() method.**



**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.
* 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

# Configure Request Processing Pipeline





## So here are the 3 very important points to keep in mind regarding the request processing pipeline (see above image for reference)

* 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**.