##### ****What is .NET Core?****

.NET Core is a new version of the .NET Framework, which is a free, open-source, general-purpose development platform maintained by Microsoft. It is a cross-platform framework that runs on Windows, Linux, and macOS. .NET Core framework can be used to build different types of applications such as console, desktop, web, mobile, cloud, IoT, machine learning, Microservices, games, etc…

.NET Core is written from scratch to make it a modular, lightweight, fast, and cross-platform framework. It includes the core features that are required to run a basic .NET Core app. Other features are provided as NuGet Packages, which you can add to your application as needed. In this way, the .NET Core application speed up the performance, reduce the memory footprint, and becomes easy to maintain.

##### ****Why .NET Core?****

There are some limitations to the .NET Framework. For example, it only runs on the Windows OS Platform. Also, you need to use different .NET APIs for different windows devices such as Windows Desktop, Windows Store, Windows Phone, and Web Applications. In addition to this, the .NET Framework is a machine-wide framework. Any changes made to it affect all applications taking a dependency on it.

##### ****.NET Core Characteristics:****

**Open-source Framework:**.NET Core is an open-source framework maintained by Microsoft and available on GitHub under MIT and Apache 2 Licenses. You can view, download, or contribute to the source code using the following GitHub repositories:

**.NET Core Runtime:** <https://github.com/dotnet/runtime>

**.NET Core SDK:** <https://github.com/dotnet/sdk>

**ASP.NET Core:** <https://github.com/dotnet/aspnetcore>

**Language Compiler Platform Roslyn:** <https://github.com/dotnet/roslyn>

**Cross-Platform:**.NET Core runs on Windows, Linux, and macOS operating systems. There is different runtime for each operating system that executes the code and generates the same output.

**Consistent across Architecture:**Execute the code with the same behavior in different instruction set architectures, including x64, x86, and ARM.

**Wide range of Applications:**Various types of applications can be developed and run on .NET Core platforms such as Console, Desktop, Web, Mobile, Cloud, IoT, ML, Microservices, Gaming, etc…

**Support Multiple Languages:**You can use C#, F#, and Visual Basic programming to develop .NET Core applications. You can use your favorite IDE, including Visual Studio 2017/2019, Visual Studio Code, Sublime Text, Vim, etc…

**Modular Architecture:**.NET Core supports a modular architecture approach using NuGet Packages. There are different NuGet Packages available for various features that can be added to the .NET Core project as needed. Even though the .NET Core library is provided as a NuGet Package. In this way, the .NET Core application speed up the performance, reduce the memory footprint, and becomes easy to maintain.

**CLI Tools:**.NET Core includes CLI tools (Command Line Interface) for development and continuous integration.

**Flexible Deployment:**.NET Core applications can be deploying user-wide or system-wide or with Docker Containers.

**Compatibility:**Compatible with .NET Framework and Mono APIs by using .NET Standard Specification.

**What type of application we can develop with .Net Core?**

1. Web: ASP.NET Core MVC, Web API, Razor Pages, and Microservices
2. Mobile
3. Console
4. Desktop Applications (Starting from 3.0)
5. IoT
6. ML
7. Gaming Applications
8. Cloud Applications

**Tools and Software Requires for the development of .NET Core Applications.**

**Machine:** (Windows, Mac, Linux)

**Editor:** Recommended Visual Studio, VS Code

**Dot Net Core SDK:** This is the software development KIT and this KIT is helpful for the development and running of the application in the system.

**How to prepare a development environment for building .NET Core/ASP.NET Core applications?**

The .NET Core can be installed in two ways:

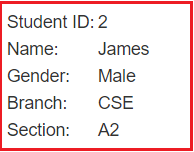
1. **By Installing Visual Studio 2017/2019**
2. **By Installing .NET Core SDK**

##### ****What is MVC?****

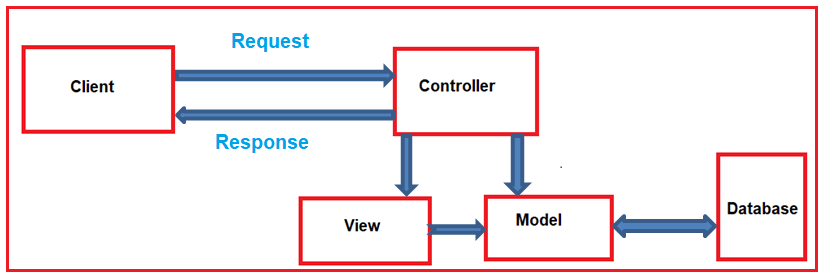
MVC stands for Model View and Controller. It is an architectural design pattern that means this design pattern is used at the architecture level of an application. So, the point that you need to remember is MVC is not a programming language, MVC is not a Framework, it is a design pattern. When we design an application, first we create the architecture of that application, and MVC plays an important role in the architecture of that particular application.

##### ****How does MVC Design Pattern work in ASP.NET Core?****

Let us see an example to understand how the MVC pattern works in the ASP.NET Core MVC application. For example, we want to design an application, where we need to display the student details on a web page as shown below.



So, when we issue a request something like “**http://dotnettutorials.net/student/details/2**” from a web browser then the following things are happening in order to handle the request.



In the MVC design pattern, the controller component, who actually handles the incoming request. In order to handle the request, the controller components do several things are as follows. The controller component creates the model that is required by a view. The model is the component in the MVC design pattern which basically contains classes that are used to store the domain data or you can say business data.

In the MVC design pattern, the Model component also contains the required logic in order to retrieve the data from a database. Once the model created by the controller, then the controller selects a view to render the domain data or model data. While selecting a view, it is also the responsibility of the controller to pass the model data.

In the MVC design pattern, the only responsibility of view is to render the model data. So, in MVC, the view is the component whose responsibility is to generate the necessary HTML in order to render the model data. Once the HTML is generated by the view, then that HTML is then sent to the client over the network, who initially made the request.

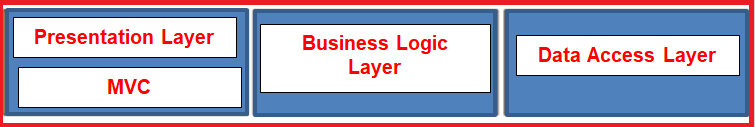
**Note:** In the MVC design pattern both the Controller and View depend on the Model. But the Model never depends on either view or controller. This is one of the main reasons for the separation of concerns. This separation of concerns allows us to build the model and test independently of the visual presentation.

**Where MVC is used in the real-time three-layer application?**

In general, a real-time application may consist of the following layers

1. **Presentation Layer:** This layer is responsible for interacting with the user.
2. **Business Layer:** This layer is responsible for implementing the core business logic of the application.
3. **Data Access Layer:** This layer is responsible for interacting with the database to perform the CRUD operations.

The MVC design pattern is basically used to implement the Presentation Layer of the application. Please have a look at the following diagram.



##### ****What is ASP.NET Core MVC?****

The ASP.NET Core MVC is a lightweight, open-source, highly testable presentation framework that is used for building web apps and Web APIs using the Model-View-Controller (MVC) design pattern. So, the point that you need to remember is, MVC is a design pattern and ASP.NET Core MVC is the framework that is based on MVC Design Pattern.

The ASP.NET Core MVC Framework provides us with a patterns-based way to develop dynamic websites and web apps with a clean separation of concerns. This ASP.NET Core MVC framework provides us the full control over the mark-up. It also supports for Test-Driven Development and also uses the latest web standards.

**What the ASP.NET Core doesn’t have?**

1. **The Global.asax file**
2. **Web.Config file**
3. **HTTP Handlers and HTTP Modules**
4. **ASP.NET Page Life-Cycle model**

A new ASP.NET Core Web Application in Visual Studio 2022 using .NET 6 with the following file and folder structure.



**What is Program Class?**

In ASP.NET Core, the Program class is the entry point for our ASP.NET Core Web Application. It contains the application startup code where we need to

1. Configure the Web Host, i.e., to host the ASP.NET Core Web Application.
2. Configure and register the services required by the application, such as MVC, Web API, Razor Pages, etc.
3. Register Middleware Components, i.e., configure the Application Request Processing Pipeline such as Authentication, Authorization, Routing, etc.
4. Start the Application so that it can listen to HTTP Requests.

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

var app = builder.Build();

app.UseRouting();

app.UseAuthentication();//Add method for cookie authentication

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();

**Note:** The earlier versions of ASP.NET Core created two files. One is Program.cs, and the other is Startup.cs. The Program.cs are responsible for configuring the host, and the startup class is responsible for configuring the Services and Middlewares. With .NET 6, both are merged into a Program.cs class file.

**What is Kestrel Web Server?**

As we already discussed, ASP.NET Core is a Cross-Platform framework. It supports developing and running applications on operating systems such as Windows, Linux, or MacOS.

The Kestrel is the Cross-Platform Web Server for the ASP.NET Core Web Application. This Server supports all the platforms that the ASP.NET Core Supports. By default, it is included as the Internal Web Server in the ASP.NET Core application.

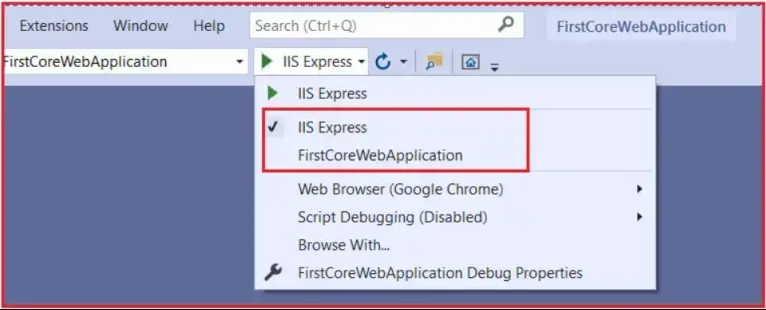
Kestrel is a lightweight, cross-platform web server built specifically for ASP.NET Core applications. It’s designed to be a fast, scalable, and efficient web server that can handle incoming HTTP requests and serve content to clients. Kestrel is the default web server that comes with ASP.NET Core, and it can be used standalone or in combination with other web servers like Apache, IIS, or Nginx.

##### ****How do you run applications using Kestrel Web Server in ASP.NET Core?****

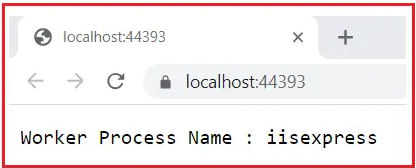
Before using the Kestrel Web Server to run our application, open the **[launchSettings.json](https://dotnettutorials.net/lesson/asp-net-core-launchsettings-json-file/)** file from your application’s Properties folder. Once you open the launchSettings.json file, you will find the following code by default.



**Note:** In our example, for IIS Express, the port number is 9623 for HTTP and 44393 for HTTPs, and the worker process is iisexpress while for the Kestrel Server, the port number is 7061 for HTTPs and 5125 for HTTPS, and the worker process name is FirstCoreWebApplication (It is nothing but your application name).



If you select IIS Express, it will use the IIS Server; if you select FirstCoreWebApplication, it will use Kestrel Server. To display the process name in the browser, you need to use **System.Diagnostics.Process.GetCurrentProcess().ProcessName** within the Main method of the Program class as shown below.



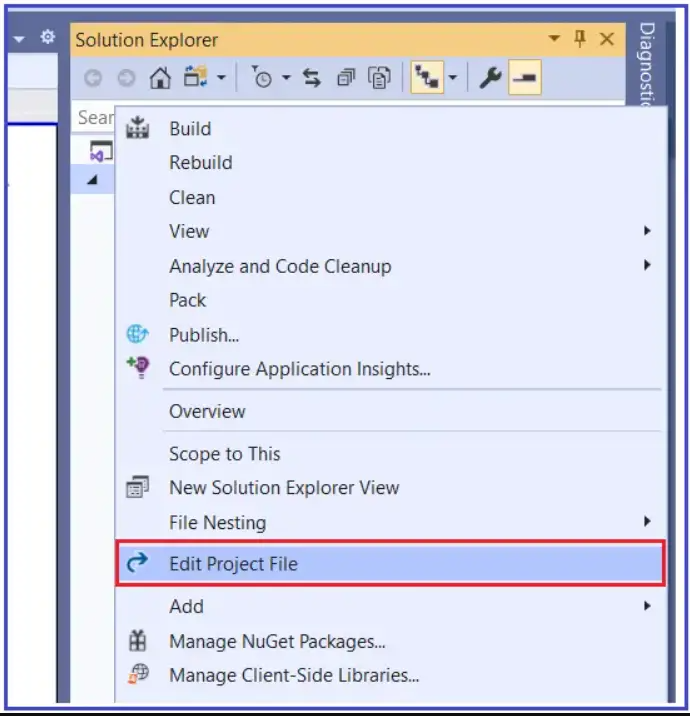
**What is the Hosting Model in ASP.NET Core?**

In ASP.NET Core, the “**Hosting Model**” refers to how the application is hosted and executed. The hosting model defines how the application starts, how the requests are processed, and how the responses are returned to clients. ASP.NET Core has two types of Hosting Models: InProcess and OutOfProcess.

In InProcess hosting model, the ASP.NET Core application is hosted inside of the IIS Worker Process i.e. w3wp.exe. In OutOfProcess hosting model, Web Requests are forwarded to the ASP.NET Core app running on the Kestrel Server.

**How do you configure InProcess/OutOfProcess Hosting in ASP.NET Core 6?**

When we create a new ASP.NET Core Web Application by using any Project Template in .NET 6, by default, the project is created with **InProcess** Hosting, which is used for hosting the application in IIS or IIS Express because it will give you better performance than OutOfProcess hosting model.



Once you open the Application Project file, modify it as shown below. As you can see, we have added the **<AspNetCoreHostingModel>** element and set its value to InProcess. The other possible value for this element is OutOfProcess.

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>net6.0</TargetFramework>

<Nullable>enable</Nullable>

<ImplicitUsings>enable</ImplicitUsings>

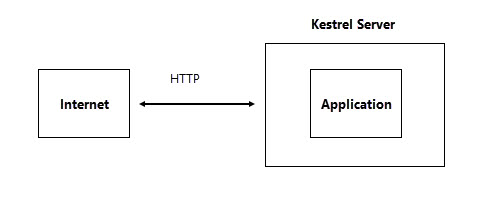
<AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>

</PropertyGroup>

< Project>

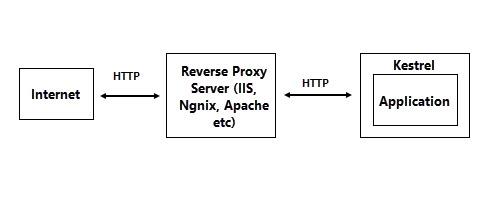
**Internal Web Server**

In an internal web server, the Kestrel web server is the internet-facing web server as all the HTTP requests are directly processed by it.



**External Web Server**

In External Web Server, a Reverse proxy server which can be either IIS, Ngnix, Apache, etc. is used along with Kestrel web Server. A reverse Proxy server provides additional security as well as configurations that are not available in Kestrel Server. It also provides load balancing functionality.



Some of the difference between the InProcess vs OutOfProcess hosting model is,

* In InProcess hosting model, request and response are served through w3wp.exe or IISExpress whereas in the OutOfProcess worker process involved is dotnet.exe
* In InProcess hosting model, a single web server is used whereas, In OutOfProcess hosting model, two web servers can be used.
* InProcess hosting model provides better performance as compared to OutOfProcess hosting model.

**Middleware Components in the ASP.NET Core Application?**

Some of the examples of using Middleware components in the ASP.NET Core application are as follows.

* We have a Middleware component (**UseAuthentication**) for authenticating the user.
* We have a middleware component (**UseHttpsRedirection**) for redirecting HTTP Requests to HTTPS.
* Another Middleware component (**UseHttpLogging**) is used to log the HTTP Requests and Responses.
* Another Middleware component (**UseDeveloperExceptionPage**) will execute when an unhandled exception occurs in the development environment.
* Similarly, we have a Middleware Component (**UseExceptionHandler**) that is used to catch exceptions, log them, and re-execute the request in an alternate pipeline.
* We have a Middleware component (**UseStaticFiles**) that handles static files such as Images, Javascript, CSS files, etc.
* We have a Middleware component (**UseAuthorization**) that is used to Authorize the users while accessing a specific resource.
* The **UseRouting** Middleware component adds the End Points Routing to the Request Processing Pipeline. If you want to define the Route using Pattern or Attribute Routing, then using the UseRouting Middleware Component is mandatory.

##### ****What is the ASP.NET Core Request Processing Pipeline?****

The ASP.NET Core Request Processing Pipeline, often called the “Middleware Pipeline,” is a sequence of middleware components that handle an incoming HTTP request and generate an appropriate HTTP response in an ASP.NET Core Web application. The Request Processing Pipeline plays a crucial role in processing requests and performing various tasks such as routing, authentication, authorization, caching, logging, and more. Each middleware component in the pipeline processes the request in a specific way and can modify the request or response as needed.