What is .NET Core?

.NET Core is a new version of the .NET Framework, which is a free, open-source, 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.

NET Core Composition: The .NET Core Framework is composed of the following parts:

CLI Tools: A **set of tooling for development and deployment**.

Roslyn: .NET Compiler Platform for C# and Visual Basic.NET languages from Microsoft.

CoreFx: A Set of framework libraries.

CoreCLR: A JIT-based CLR (Common Language Runtime). CoreCLR is the .NET execution engine in .NET Core, performing functions such as garbage collection and compilation to machine code.

NET Core Runtime: The .NET Core Runtime **is required only to run .NET Core applications**. The .NET Core Runtime just contains the resources or libraries which are required to run existing .NET Core applications.

.NET Core SDK: If you want to develop and run the .NET Core application, then you need to download the .NET Core SDK. The .NET Core SDK contains the .NET Core Runtime. So, if you installed the .NET Core SDK, then there is no need to install .NET Core Runtime separately.

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="12.0.3" />
  </ItemGroup>

</Project>
```

Main Method

```
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<Startup>();
            });
}
```

ASP.NET Core Web Application initially starts as a Console Application and the Main() method is the entry point to the application. So, when we execute the ASP.NET Core Web application, first it looks for the Main() method and this is the method from where the execution starts. The Main() method then configures ASP.NET Core and starts it. At this point, the application becomes an ASP.NET Core web application.

Main() method, makes a call to the CreateHostBuilder() method by passing the command line arguments args as an argument.

Main() method of the Program class **calls the static CreateHostBuilder() method**. Then the **CreateHostBuilder() method calls the static CreateDefaultBuilder()method on the Host class**. The **CreateDefaultBuilder() method sets the web host which will host our application** with default preconfigured configurations. **CreateDefaultBuilder()- configures Kestrel (Internal Web Server for ASP.NET Core), IISIntegration, and other configurations.**

CreateHostBuilder() method returns an object that implements the IHostBuilder interface. The Host is a static class that can be used for creating an instance of IHostBuilder with pre-configured defaults.

CreateDefaultBuilder() method creates a new instance of the HostBuilder with pre-configured defaults. Internally, it **configures Kestrel (Internal Web Server for ASP.NET Core), IISIntegration, and other configurations. CreateDefaultBuilder() method sets up a web host with default configurations.**

CreateDefaultBuilder() method does several things. Some of them are as follows

1. **Setting up the webserver**
2. **Loading the host and application configuration from various configuration sources**
3. **Configuring logging**

Within the Main() method, on this IHostBuilder object, the **Build() method** is called which **actually builds a web host**. Then it **hosts our asp.net core web application within that Web Host.** Finally, on the **web host, it called the Run()method, which will actually run the web application and it starts listening to the incoming HTTP requests**.

StartupClass-

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add services to the container.
    // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                await context.Response.WriteAsync("Hello World!");
            });
        });
    }
}
```

While setting up the host, the Startup class is also configured using the UseStartup() extension method of the IHostBuilderclass. The Startup class has two methods
ConfigureServices() method of the Startup class configures the services which are required by the application.

Configure() method of the Startup class sets up the application request pipeline using IApplicationBuilder instance that is provided by builtin IOC conatiner .

Hosting- hosting service is a service that runs servers connected to the Internet, allowing organizations and individuals to serve content or host services connected to the Internet
ASP.NET core Web application can be hosted in two ways i.e.
1- InProcess hosting (default In Process hosting model).
2- OutOfProcess hosting.

Right-click on your **project** , select the **properties,** select the **Debug** and have a look at the value of the **Hosting Model drop-down** listThe drop-down list contains three values i.e. **Default (InProcess), InProcess, and Out Of Process**

InProcess Hosting Model -

add <AspNetCoreHostingModel> element and set its value to InProcess. The order possible value for this element is OutOfProcess.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
 <PropertyGroup>
       <TargetFramework>netcoreapp3.1</TargetFramework>
      <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
 </PropertyGroup>
</Project>
```

In case of InProcess hosting (i.e. when the **CreateDefaultBuilder() sees the value as InProcess** for the AspNetCoreHostingModel element in the project file), behind the scene the **CreateDefaultBuilder() method internally calls the UseIIS() method**. Then host the application inside the IIS worker process (i.e. w3wp.exe for IIS and iisexpress.exe for IISExpress).

In ASP.NET Core, with InProcess Hosting Model our application is going to be hosted in the IIS worker process

From the performance point of view, InProcess hosting model, we do not have the performance penalty for navigating the requests between the internal and external web servers , the **InProcess hosting model delivers significantly higher request** throughput than the OutOfProcess hosting model.

Display process name- in the browser you need to use the System.Diagnostics.Process.GetCurrentProcess().ProcessName within the Startup

What is IIS Express? The IIS Express is a **lightweight, self-contained version of IIS.** It is **optimized for web application development.** The most important point that you need to remember is we use ***IIS Express only in development,*** not on production. In production we generally use IIS.

Out of Process Hosting - there are two web servers.

1. internal webserver which is the Kestrel web Server
2. external web server which can be IIS, Apache, and Nginx.

The most important point that you need to keep in mind is depending on how you are running your application with the OutOfProcess hosting model, the external web server may or may not be used.

we run the application using the .NET core CLI then Kestrel is the only web server that is going to be used to handle and process the incoming HTTP request.

What happens when we run the .NET Core application using .NET Core CLI?

it ignores the hosting setting that specified in the application's project file i.e. csproj file. So, in that case, the value of the AspNetCoreHostingModel element is going to be ignored. The .NET Core CLI always uses OutOfProcess Hosting Model and Kestrel is the webserver that will host the ASP.NET Core application and also handles the HTTP requests.

Kestrel Web Server- Kestrel is the **cross-platform web server** for the ASP.NET Core application. this Server **supports all the platforms and versions** that the ASP.NET Core supports. By default, it is **included as the internal web server** in the .NET Core application.

The Kestrel Web Server is generally used as an edge server i.e. the internet-facing web server which directly processes the incoming HTTP request from the client. **In the case of the Kestrel web server, the process name that is used to host and run the ASP.NET Core application is the project name.** When we run an ASP.NET Core application using the .NET Core CLI, then the .NET Core runtime uses Kestrel as the webserver.

Run the application using Kestrel Web Server-

By default, the visual studio uses IIS Express to host and run the ASP.NET Core application. So, the process name is IIS Express.
open the launchSettings.json file which is present inside the Properties folder

Changing the Port Number: If you want then you can also change the Port number for Kestrel Server. To do so open the launchSettings.json file and give any available Port number

LaunchSettings.json file- is only used within the local development machine. That means this file is not required when we publishing our asp.net core application to the production server.

If you have certain settings and you want your application to use such settings when you publish and deploy your asp.net core application to the production server, then you need to store such settings in the appsettings.json file. Generally, in the ASP.NET Core application, the configuration settings are going to be stored in the appsettings.json file.
the value of the commandName property of the launchSettings.json file can be any one of the following.

1- IISExpress 2- IIS 3- Project

The CommandName property value of the launchSettings.json file along with the AspNetCoreHostingModel element value from the application's project file will determine the internal and external web server (reverse proxy server) that are going to use and handle the incoming HTTP Requests.

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

Startup class
it is executed first when the application starts. The startup class can be configured using UseStartup<T>() extension method at the time of configuring the host in the Main() method of Program class.

ConfigureServices() method-configures the services which are required by the application.

ConfigureServices method includes the IServiceCollection parameter to register services to the IoC container.
ConfigureServices method is a place where you can register your dependent classes with the built-in IoC container. After registering the dependent class, it can be used anywhere in the application. You just

need to include it in the parameter of the constructor of a class where you want to use it. The IoC container will inject it automatically.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    services.AddMvc();
}
```

Configure() method - Configure method is a place where we can configure the application request pipeline for our asp.net core application using the IApplicationBuilder instance that is provided by the built-in IoC container.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

appsettings.json file *is an application configuration file used to store configuration settings such as database connections strings, any application scope global variable*

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

What are the Default Orders of reading the configuration sources?

The default orders in which the various configuration sources are read for the same key are as follows

1. appsettings.json,
2. appsettings.{Environment}.json here we use appsettings.development.json

3. User secrets 4- Environment variables 5-Command-line arguments

Middleware Components- Middleware Components are the software components (technically components are nothing but the C# Classes) that are assembled into the application pipeline to handle the HTTP Requests and Responses.we need to configure the Middleware components within the Configure() method. Each middleware component in ASP.NET Core Application performs the following tasks.

1. Chooses whether to pass the HTTP Request to the next component in the pipeline. This can be achieved by calling the next() method within the middleware.
2. Can perform work before and after the next component in the pipeline

Use* methods on the IApplicationBuilder object

Where we use Middleware Components in the ASP.NET Core application?

1. We may have a Middleware component for **authenticating the user**
2. Another Middleware component may be used to **log the request and response**
3. Similarly, we may have a Middleware component that is used to **handle the errors**
4. We may have a Middleware component that is used to **handle the static files** such as images, Javascript or CSS files, etc.
5. Another Middleware component may be used to **Authorize the users** while accessing a specific resource

UseDeveloperExceptionPage() Middleware component

UseRouting() Middleware component

UseEndpoints() Middleware component

Middleware component in ASP.NET Core can

1. Handle the incoming HTTP request by generating an HTTP response.

2. Process the incoming HTTP request, modify it, and then pass it to the next middleware component
3. Process the outgoing HTTP response, modify it, and then pass it on to either the next middleware component or to the ASP.NET Core web server

Short-circuiting- A middleware component in ASP.NET Core may also handle the HTTP Request by generating an HTTP Response. The ASP.NET Core Middleware component may also decide not to call the next middleware component in the request pipeline. This concept is called short-circuiting the request pipeline

Request delegates *are used to build the request pipeline. Request delegates can be configured using the Run, Map, and Use extension methods.*

Use" and "Run" extension methods used to register the Inline Middleware component into the Request processing pipeline.
The **"Run"** *extension method allows us to add the terminating middleware (the middleware which will not call the next middleware components in the request processing pipeline). On the other hand, the* **"Use"** *extension method allows us to add the middleware components which may call the next middleware component in the request processing pipeline.*

DeveloperExceptionPage Middleware- This middleware component is going to execute when there is an unhandled exception occurred in the application and since it is in development mode. Developer Exception Page contains five tabs such as **Stack, Queue, Cookies, Headers, and Routing.**

1. Stack: The Stack tab gives the information of stack trace which **indicated where exactly the exception occurred, the file name, and the line number that causes the exception.**
2. Query: The Query tab **gives information about the query strings**.
3. Cookies: The Cookies tab **displays the information about the cookies set by the request.**
4. Header: The Header tab g**ives information about the headers which is sent by the client when makes the request.**
5. Route: The Route tab gives information about the **Route Pattern and Route HTTP Verb** type of the method, etc

UseRouting() Middleware: This middleware component is used to add Endpoint Routing Middleware to the request processing pipeline i.e. it will map the URL (or incoming HTTP Request) to a particular resource.

UseEndpoints() Middleware- In this middleware, the routing decisions are going to be taken using the Map extension method. Following is the default implementation of UseEndpoints middleware components. In the MapGet extension method, we have specified the URL pattern like "/". This means the domain name only. So, any request with only the domain name is going to be handle by this middleware. Instead of MapGet, you can also use the Map

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", async context =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
});
```

What is the difference between MapGet and Map method?

The MapGet method is going to handle the GET HTTP Requests whereas the Map method is going to handle all types of HTTP requests such as GET, POST, PUT, & DELETE, etc

wwwroot folder wwwroot folder is treated as the webroot folder and this folder or directory should be present in the root project folder. The Static files can be stored in any folder under the webroot folder and can be accessed with a relative path to that root. there should be separate folders for the different types of static files such as JavaScript, CSS, Images, Library scripts,

In order to serve the static files, you need to include the app.UseStaticFiles() middleware component in the 'Configure()' method of Startup.cs file

.NET Core CLI (Command Line Interface) is a new cross-platform tool that is used for creating, restoring packages, building, running, and publishing ASP.NET Core Applications. The .NET Core CLI command for any kind of web application uses Out of Process hosting i.e. it uses the Kestrel server to run the application.

command structure of .NET Core CLI Command:

dotnet <command> <argument> <option>

All the .NET Core CLI commands start with the driver named dotnet. The driver i.e. dotnet starts the execution of the specified command. After dotnet, we need to specify the command (also known as the verb) to perform a specific action. Each command can be followed by arguments and options.

dotnet help - to get all commands

1. *add: Add a package or reference to a .NET project.*
2. *build: Build a .NET project.*
3. *build-server: Interact with servers started by a build.*
4. *clean: Clean build outputs of a .NET project.*
5. *help: Show command-line help.*
6. *list: List project references for a .NET project.*
7. *msbuild: Run Microsoft Build Engine (MSBuild) commands.*
8. *new: Create a new .NET project or file.*
9. *nuget: Provides additional NuGet commands.*
10. *pack: Create a NuGet package.*
11. *publish: Publish a .NET project for deployment.*
12. *remove: Remove a package or reference from a .NET project.*
13. *restore: Restore dependencies specified in a .NET project.*
14. *run: Build and run a .NET project output.*
15. *sln: Modify Visual Studio solution files.*
16. *store: Store the specified assemblies in the runtime package store.*
17. *test: Run unit tests using the test runner specified in a .NET project.*
18. *tool: Install or manage tools that extend the .NET experience.*
19. *vstest: Run Microsoft Test Engine (VSTest) commands.*

Project Modification Commands:

1. *add package: Adds a package reference to a project.*
2. *add reference: Adds project-to-project (P2P) references.*
3. *remove package: Removes package reference from the project.*
4. *remove reference: Removes project reference.*
5. *list reference: Lists all project-to-project references.*

Advanced Command:

1. *nuget delete: Deletes or un-lists a package from the server.*
2. *nuget locals: Clear or lists NuGet resources.*
3. *nuget push: Pushes a package to the server and publishes it.*
4. *msbuild: Builds a project and all of its dependencies.*
5. *dotnet install script: Script used to install .NET Core CLI tools and the shared runtime.*