1. .Net Core Fundamental

1.1 .Net Core Overview

- .NET Core is a free, open-source, and cross-platform framework for building modern applications. It's designed to be modular, lightweight, and highly performant.
- Key features :
    - **Cross-Platform** : .NET Core runs on Windows, macOS, and Linux, allowing developers to build applications that can run on any platform.
    - **High Performance** : It offers high-speed execution and efficient memory usage, making it suitable for high-performance scenarios.
    - **Modularity** : .NET Core is composed of smaller, reusable components, allowing developers to include only the necessary libraries in their applications, resulting in smaller application footprints.
    - **Command-Line Interface (CLI)** : It includes a CLI tool for creating, building, and managing .NET Core applications, simplifying the development process.
    - **Language Support** : .NET Core supports multiple programming languages, including C#, F#, and Visual Basic.NET, providing developers with flexibility.
    - **Support for Modern Workloads** : It supports modern workloads such as cloud-native applications, microservices, containers, and serverless computing.
    - **Integrated Development Environment (IDE) Support** : .NET Core is supported by popular IDEs like Visual Studio, Visual Studio Code, and JetBrains Rider, enhancing developer productivity.
    - **Community and Support** : .NET Core has a vibrant community and is backed by Microsoft, ensuring continuous support, updates, and a rich ecosystem of libraries and tools.

1.2 ASP.Net Core

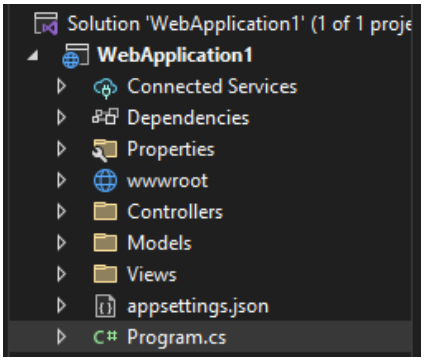
- ASP.NET Core is a high-performance, open-source framework for building modern web applications and services. Here's a brief overview:
    - **Cross-Platform** : ASP.NET Core runs on Windows, macOS, and Linux, enabling

developers to build and deploy applications on any platform.

- **Performance** : It offers high performance and scalability, with features like built-in support for asynchronous programming and efficient request handling.
- **Modular and Lightweight** : ASP.NET Core is modular, allowing developers to include only the necessary components in their applications, resulting in smaller application footprints and faster startup times.
- **Open Source** : ASP.NET Core is open source, with contributions from the community and continuous updates and improvements from Microsoft.
- **Cloud-Ready** : It is designed for cloud-native development, with built-in support for Docker containers, Kubernetes, and integration with cloud services like Azure.
- **Cross-Language Support** : ASP.NET Core supports multiple programming languages, including C#, F#, and Visual Basic.NET, providing developers with flexibility.
- **Integrated Development Environment (IDE) Support** : It is supported by popular IDEs like Visual Studio, Visual Studio Code, and JetBrains Rider, enhancing developer productivity.
- **Razor Pages and MVC** : ASP.NET Core supports both Razor Pages and Model-View-Controller (MVC) for building web applications, offering developers different options for structuring their projects.
- **Security** : ASP.NET Core includes built-in security features to help developers protect their applications from common vulnerabilities, such as cross-site scripting (XSS) and cross-site request forgery (CSRF).

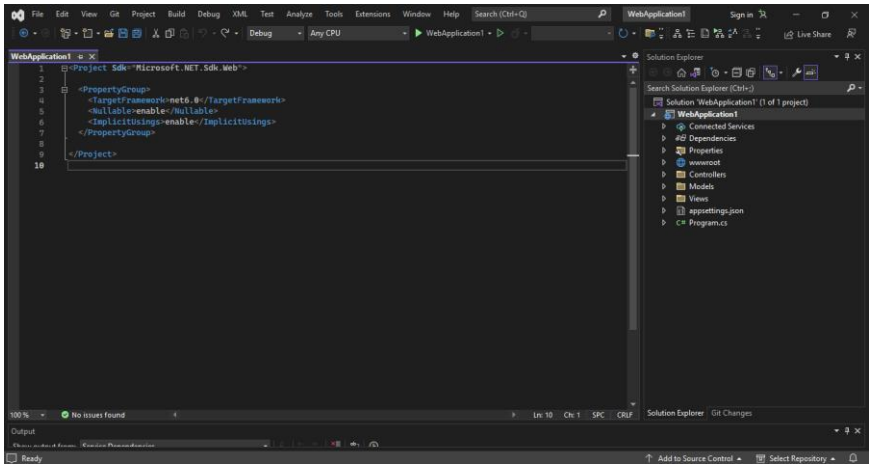
1.3 Project Structure

- The details of the project structure can be seen in the solution explorer as follow :



- **.csproj File :**

  Edit Project File used in order to edit the .csproj file. As shown in the following image.



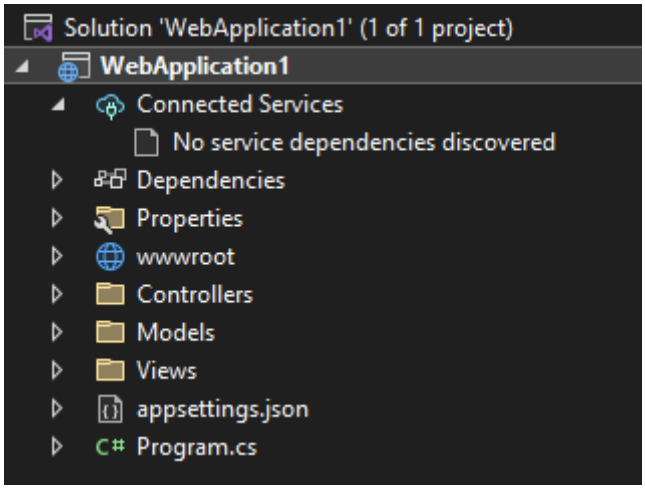
  Project's SDK is Microsoft.NET.Sdk.Web. The target framework is net6.0 indicating that we are using .NET 6.

  The <Nullable> elements decide the project wide behaviour of Nullable of Nullable reference types. The value of enable indicates that the Nullable reference types are enabled for the project.

  The <ImplicitUsings> element can be used to enable or disable. When it is set to enable, certain namespaces are implicitly imported for you.

- **Connected Services :**

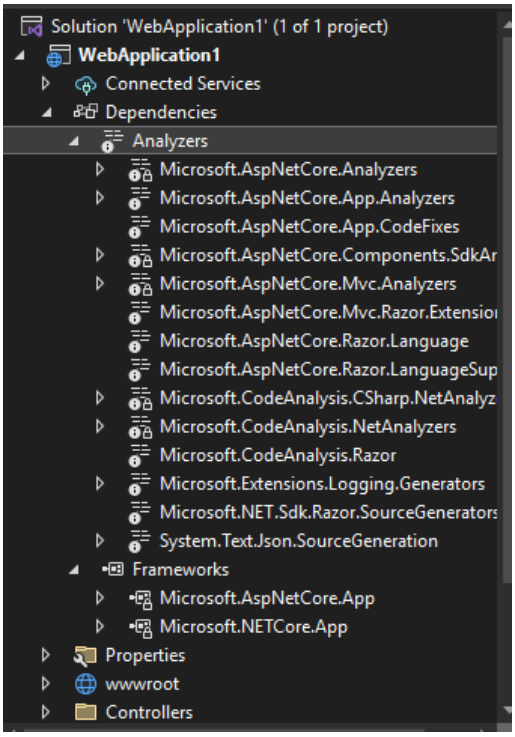
  It contains the details about all the service references added to the project. A new service can be added here, for example, if you want to add access to Cloud Storage of Azure Storage you can add the service here.



- **Dependencies :**

  The Dependencies node contains all the references of the NuGet packages used in the project. Here the Frameworks node contains reference two most important

dotnet core runtime and asp.net core runtime libraries. Project contains all the installed server-side NuGet packages, as shown below.



- **Properties :**

  Properties folder contains a launchSettings.json file, which containing all the information required to lunch the application. Configuration details about what action to perform when the application is executed and contains details like IIS settings, application URLs, authentication, SSL port details, etc.

- **wwwroot :**

  All the static files required by the project are stored and served from here. The webroot folder contains a sub-folder to categorize the static file types, like all the Cascading Stylesheet files, are stored in the CSS folder, all the javascript files are stored in the js folder and the external libraries like bootstrap, jquery are kept in the library folder.

- **Controllers :**

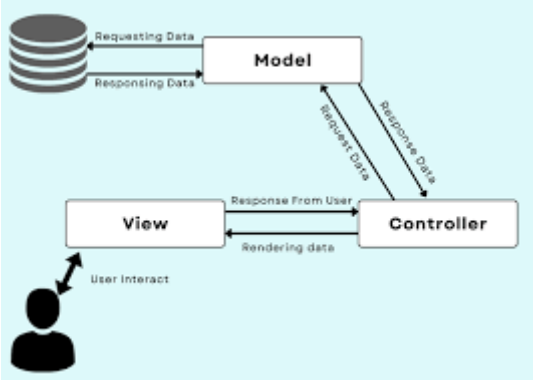
  Controller handles all the incoming requests. All the controllers needed for the project are stored here. Controllers are responsible for handling end user interaction, manipulating the model and choose a view to display the UI. Each controller class inherits a Controller class or ControllerBase class. Each controller class has “Controller” as a suffix on the class name, for example, the default “HomeController.cs” file can be found here. As shown below.

- **Models :**

  A Model represents the data of the application and a ViewModel represents data that will be displayed in the UI. The models folder contains all the domain or entity classes. Please note user can add folders of his choice to create logical grouping in the project.

- **Views :**

  A view represents the user interface that displays ViewModel or Model data and can provide an option to let the user modify them. Mostly a folder in name of the controller is created and all the views related to it are stored in it. Here HomeController related view Index.cshtml and Privacy.cshtml is stored in Home folder and all the shared view across the application is kept in Shared folder. Under this shared folder there are _Layout.cshtml, _ValidationScriptsPartial.cshtml and Error.cshtml view files. There are two more view files, _ViewImports.cshtml and _ViewStart.cshtml under the view folder.



- **appsettings.json :**

  This file contains the application settings, for example, configuration details like logging details, database connection details.

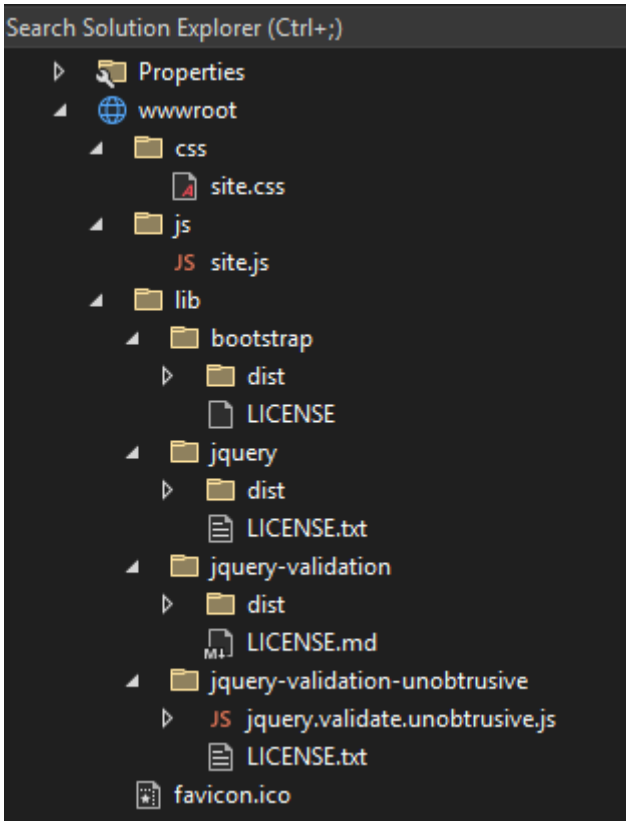
- **Program.CS :**

  This class is the entry point of the web application. It builts the host and executes the run method.

1.4 wwwroot Folder

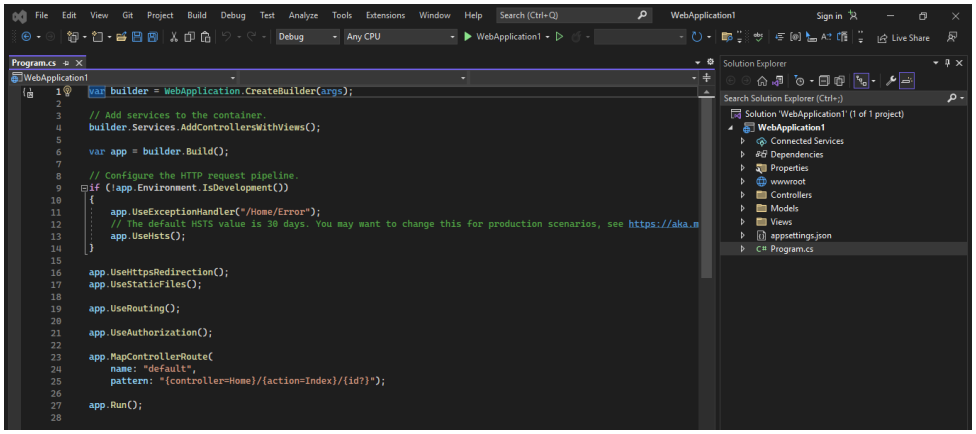
- This is the webroot folder and all the static files required by the project are stored and served from here. The webroot folder contains a sub-folder to categorize the static file types, like all the Cascading Stylesheet files, are stored in the CSS folder, all the javascript files are stored in the js folder and the external libraries like bootstrap, jquery are kept in the library folder.

- Generally, there should be separate folders for the different types of static files such as JavaScript, CSS, Images, library scripts, etc. in the wwwroot folder as shown below.



1.5 Program.cs

- Program.cs file in ASP.NET Core MVC application is an entry point of an application. It contains logic to start the server and listen for the requests and also configure the application.



```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.m
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

- The first line creates an object of WebApplicationBuilder with preconfigured defaults using the CreateBuilder() method.

```
var builder = WebApplication.CreateBuilder(args);
```

- The CreateBuilder() method setup the internal web server which is Kestrel. It also

specifies the content root and read application settings file appsettings.json.

- Using this builder object, you can configure various things for your web application, such as dependency injection, middleware, and hosting environment. We can pass additional configurations at runtime based on the runtime parameters.
- The builder object has the Services() method which can be used to add services to the dependency injection container.
- The AddControllersWithViews() is an extension method that register types needed for MVC application (model, view, controller) to the dependency injection. It includes all the necessary services and configurations for MVC So that our application can use MVC architecture.

```
builder.Services.AddControllersWithViews();
```

- The builder.Build() method returns the object of WebApplication using which you can configure the request pipeline using middleware and hosting environment that manages the execution of your web application.

```
var app = builder.Build();
```

- Now, using this WebApplication object app, we can configure an application based on the environment it runs on e.g. development, staging or production. The following adds the middleware that will catch the exceptions, logs them, and reset and execute the request path to '"/home/error" if the application runs on the development environment.

```
if (!app.Environment.IsDevelopment())
{
        app.UseExceptionHandler("/Home/Error");
}
```

- The method starts with "Use" word means it configures the middleware. The following configures the static files, routing, and authorization middleware respectively.

```
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();
```

- The UseStaticFiles() method configures the middleware that returns the static files from the wwwroot folder only.
- The MapControllerRoute() defines the default route pattern that specifies which controller, action, and optional route parameters should be used to handle incoming requests.

```
app.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}"
);
```

- Finally, app.run() method runs the application,start listening the incomming request. It turns a console application into an MVC application based on the provided configuration.

1.6 Startup.cs

- As the name suggests, it is executed first when the application starts.
- The startup class can be configured using UseStartup<T>() method at the time of configuring the host in the Main() method of Program class as shown below.

```
public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args)
    {
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
    }
}
```

- Startup class includes two public methods: ConfigureServices and Configure.

- The Startup class must include a Configure method and can optionally include ConfigureService method.

- **ConfigureServices() :**
  - The Dependency Injection pattern is used heavely in ASP.NET Core architecture. It includes built-in IoC container to provide dependent objects using constructors.
  - The ConfigureServices method is a place where you can register your dependent classes with the built-in IoC container. After registering 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.
  - ConfigureServices method includes IServiceCollection parameter to register services to the IoC container.
- **Configure() :**
  - The Configure method is a place where you can configure application request pipeline for your application using IApplicationBuilder instance that is provided by the built-in IoC container.
  - ASP.NET Core introduced the middleware components to define a request pipeline, which will be executed on every request. You include only those middleware components which are required by your application and thus increase the performance of your application.
  - The following is a default Configure method.

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

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

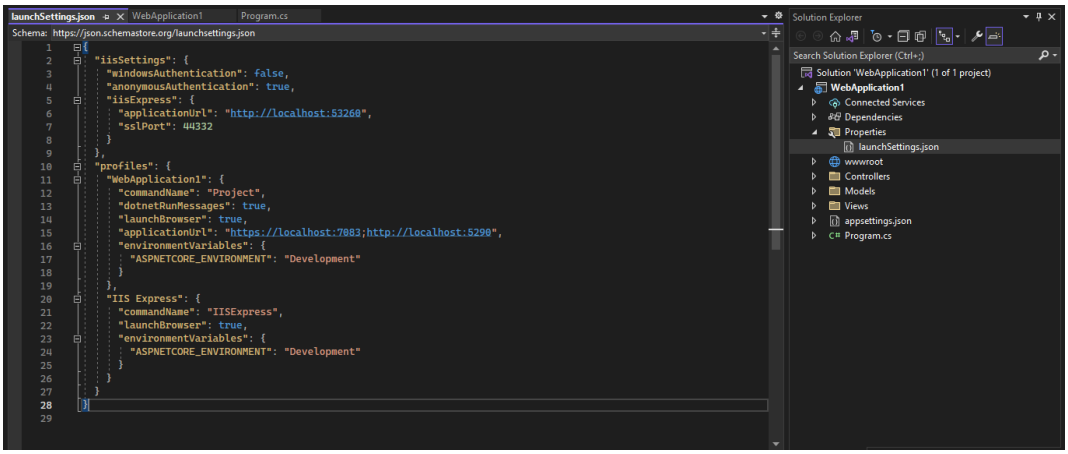
  - The Configure method includes three parameters IApplicationBuilder, IHostingEnvironment, and ILoggerFactory by default. These services are framework services injected by built-in IoC container.
  - At run time, the ConfigureServices method is called before the Configure method. This is so that you can register your custom service with the IoC container which you may use in the Configure method.

1.7 launchSettings.json

- The launchSettings.json file in ASP.NET Core is used primarily in the development environment for configuring how an application is launched and its initial environment settings.
- The settings within the LaunchSettings.json file will be used when we run or launch the ASP.NET core application either from Visual Studio or by using .NET Core CLI. launchSettings.json file is only used within the local development machine. This file is not required when we publish our ASP.NET Core Application to the Production Server.
- Some settings are as follows :
- **Profiles :**

  The file contains different profiles for launching the application. Each profile can specify different settings like the application URL, environment variables, command line arguments, etc. These profiles help set up different environments for development, like testing the application in different configurations without changing the code.

  In the below launchSettings.json file, within the profiles, we have two sections, i.e., IIS Express and WebApplication1.



  We can configure different profiles with different settings based on our development needs. When we run our ASP.NET Core application in Visual Studio or using the .NET CLI, we can select one of these profiles to specify how our application should be launched and debugged.

- **CommandName :**

  The value of the Command Name property of the launchSettings.json file can be any 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 is going to be used to host the application and handle the incoming HTTP Requests. For a better understanding, please have a look at the below table.

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

- **launchBrowser :**

  A boolean value determining whether a browser is launched when the application starts. That means this property determines whether to launch the browser and open the root URL or not. The value true indicates that it will launch the browser, and the value false means it will launch the web browser once it hosts the application.

- **environmentVariables :**

  It can be used to set environment variables that are important during the development phase. By default, it includes one configuration key called ASPNETCORE_ENVIRONMENT, using which we can specify the environment, such as Development, Production, and Staging.

- **dotnetRunMessages :**

  The dotnetRunMessages property in the launchSettings.json file of an ASP.NET Core project is a relatively less commonly used setting. Its primary function is to enable or disable the display of certain messages when the application is launched using the dotnet run command. The dotnetRunMessages property is typically a boolean value (true or false). Setting it to false suppresses certain run-time messages that are usually displayed when you start the application.

  It is particularly useful when you want a cleaner console output for better readability or when you are only interested in specific output messages. For instance, if your application generates custom logging messages during startup,

suppressing the standard dotnet run messages can make these custom messages more prominent.

- **applicationUrl :**

  The applicationUrl property specifies the application base URL(s) using which you can access the application. If you enable HTTPS while creating the project, you will get two URLs, i.e., one URL using HTTP protocol and another URL using HTTPS protocol. So, it specifies the URLs on which the application will listen when it's running. This is useful for testing applications on different ports or host names during development.

- **sslPort :**

  This property specifies the HTTPS Port number to access the application in the case of an IIS Express Server. The value 0 means you cannot access the application using HTTPS protocol.

- **windowsAuthentication :**

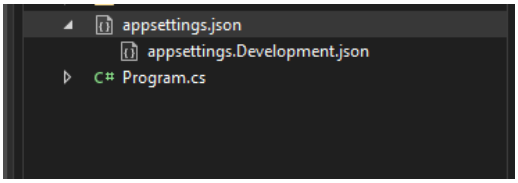
  This property will specify whether Windows Authentication is enabled for your application. If true, it means Windows Authentication is enabled, and false means it is not enabled.

- **anonymousAuthentication :**

  This property will specify whether Anonymous Authentication is enabled for your application or not. If true, it means Anonymous Authentication is enabled, and false means it is not enabled.

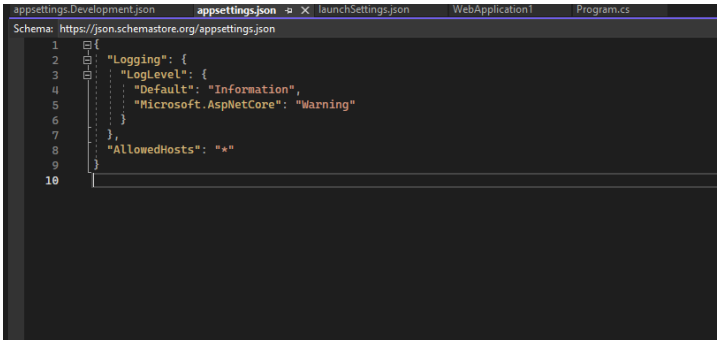
1.8 appSettings.json

- The appsettings.json file in an ASP.NET Core application is a JSON formatted file that stores configuration data. In this file, you can keep settings like connection strings, application settings, logging configuration, and anything else you want to change without recompiling your application. The settings in this file can be read at runtime and overridden by environment-specific files like appsettings.Development.json or appsettings.Production.json.



- The appsettings.json file is the application configuration file used to store configuration settings such as database connection strings, any application scope

global variables, etc. If you open the ASP.NET Core appsettings.json file, you see the following code created by Visual Studio by default.



- The appsettings.json file typically resides in the root directory of your ASP.NET Core project. You can add multiple appsettings.json files with different names, such as appsettings.development.json, appsettings.production.json, etc., to manage configuration for different environments (e.g., development, production, staging).
- Here's what each section typically means:
    - Logging: Defines the logging level for different components of the application.
    - AllowedHosts: Specifies the hosts that the application will listen to.
- **Configuration Sections :** The file is typically divided into sections that represent different aspects of the application settings. For example, you might have sections like ConnectionStrings, Logging, or custom sections for your application-specific settings.
- **Security Consideration :** It's important not to store sensitive information, like passwords or secret keys, directly in appsettings.json. Instead, use secure storage like environment variables, Azure Key Vault, or other secure configuration providers.
- Key Features of ASP.NET Core AppSettings.json file :
    - **Hierarchical Configuration :** Settings are organized hierarchically, which can be accessed using a colon (:) as a separator in C#.
    - **Environment-Specific Settings :** ASP.NET Core supports environment-specific settings using files like appsettings.Development.json, appsettings.Staging.json, and appsettings.Production.json. The appropriate file is selected based on the current environment.
    - **Safe Storage of Sensitive Data :** Sensitive data like connection strings and API keys can be stored in appsettings.json, but for higher security, it's recommended to use user secrets in development and secure services like

Azure Key Vault in production.

- **Reloadable :** The configuration can be set up to be reloadable, meaning changes to the appsettings.json file can be read without restarting the application.

- 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
- ASP.NET Core AppSettings.json File Real-Time Examples
  - **Database Connection Strings :** In this example, a connection string is defined for a database. The application can access this connection string to interact with the database.

```
{
    "ConnectionStrings": {
            "DefaultConnection":
            "Server=myServerAddress;Database=myDataBase;User
            Id=myUsername;Password=myPassword;"
    }
}
```

  - **Logging Configuration :** In this example, the logging levels are configured for the application and for specific namespaces within the application.

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

  - **External Service Configuration** : This example shows how to store API settings for external services, like base URLs and API keys.

```
{
    "ExternalService": {
        "BaseUrl": "https://api.example.com/",
        "ApiKey": "Your-Api-Key-Here"
    }
}
```

- **Custom Application Settings :** Custom settings specific to the application, like page size for pagination or support email addresses.

```
{
    "ApplicationSettings": {
        "PageSize": 20,
        "SupportEmail": "support@example.com"
    }
}
```

- **Authentication Settings :** Settings related to authentication, like JWT settings in this case.

```
{
    "Authentication": {
        "Jwt": {
            "Key": "Your-Secret-Key",
            "Issuer": "YourIssuer",
            "Audience": "YourAudience"
        }
    }
}
```

- **API Rate Limiting :** Configurations for API rate limiting.

```
{
    "RateLimiting": {
        "EnableRateLimit": true,
        "MaxRequestsPerSecond": 5
    }
}
```