

.Net Core Fundamental

1. .Net Core Overview.....	2
2. ASP.Net Core.....	6
3. Project Structure.....	9
4. wwwroot Folder.....	14
5. Program.cs.....	15
6. Startup.cs.....	18
7. launchSettings.json.....	21
8. appSettings.json.....	25

.Net Core Overview

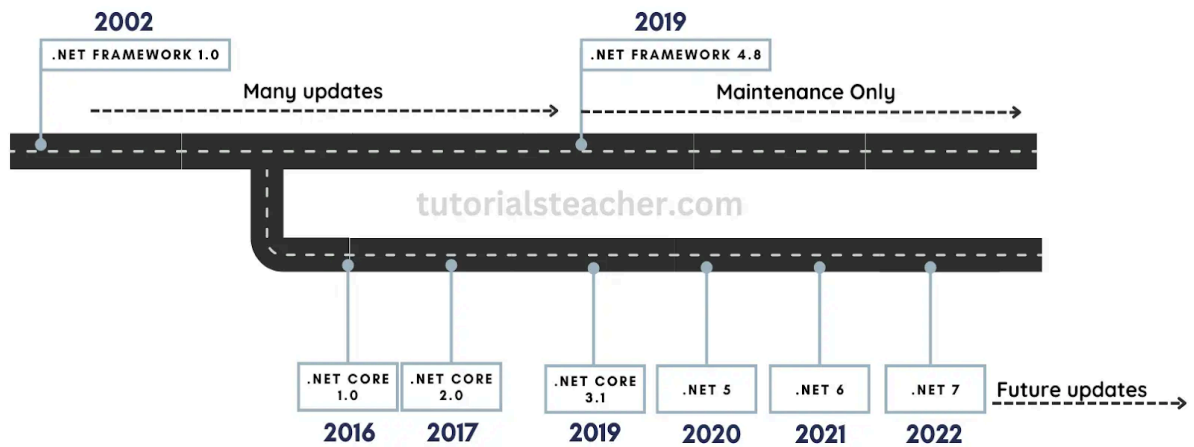
- .NET Core is a new version of .NET Framework, which is a free, open-source, general-purpose development platform maintained by Microsoft. It is a cross-platform framework that runs on Windows, macOS, and Linux operating systems.
- .NET Core Framework can be used to build different types of applications such as mobile, desktop, web, cloud, IoT, machine learning, microservices, game, etc.
- .NET Core is written from scratch to make it 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 it in 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 (.NET):

1. .NET Core (.NET) is an open-source and cross-platform. It provides the runtime environment where the ASP.NET Core Web Applications will run.
2. .NET Core is a runtime that executes applications that are built on it.
3. Install .NET Core Runtime to run applications and install .NET Core SDK to build applications.
4. .NET 7 is the latest stable version. .NET 8 is in Preview.

.NET Core Version History

Version	Visual Studio	Release Date	End of Support
.NET 7 - Latest Version	Visual Studio 2022 v17.4	Nov 8, 2022	
.NET 6 (LTS)	Visual Studio 2022	Nov 9, 2021	Nov 12, 2024
.NET 5	Visual Studio 2019	Nov 10, 2020	May 10, 2022
.NET Core 3.1 (LTS)	Visual Studio 2019	Dec 3, 2019	Dec 13, 2022
.NET Core 3.0	Visual Studio 2019	Sep 23, 2019	Mar 3, 2020
.NET Core 2.1 (LTS)	Visual Studio 2017, 2019	May 30, 2018	Aug 21, 2021
.NET Core 2.0	Visual Studio 2017, 2019	Aug 14, 2017	Oct 1, 2018
.NET Core 1.0	Visual Studio 2017	Jun 27, 2016	Jun 27, 2019



.Net Core Characteristic

- **Cross-Platform Compatibility:** .NET Core is designed to run on multiple operating systems, including Windows, macOS, and Linux, enabling developers to create applications that can be deployed across various platforms without significant modifications.
- **Open-Source:** Being open-source, .NET Core encourages community participation, transparency, and innovation. Developers can contribute to its development, report issues, and suggest enhancements, fostering a collaborative ecosystem.
- **High Performance:** .NET Core is optimized for performance, leveraging features like just-in-time (JIT) compilation, asynchronous programming, and runtime optimizations. This focus on performance ensures that applications built with .NET Core are efficient and responsive.
- **Modularity and Flexibility:** .NET Core offers a modular architecture, allowing developers to include only the required components in their applications. This modularity reduces overhead and improves performance by eliminating unnecessary dependencies.
- **Unified Platform:** With the introduction of .NET 5 (and later .NET 6), Microsoft unified the .NET platform, providing a single framework for building various types of applications, including web, desktop, mobile, and cloud-native applications.
- **Language Interoperability:** .NET Core supports multiple programming languages, including C#, F#, and Visual Basic.NET. This language interoperability enables developers to choose the language that best suits their skills and project requirements while still benefiting from the .NET ecosystem and runtime.
- **Modern Workload Support:** .NET Core is optimized for modern development scenarios, such as cloud-native applications, microservices, and containerized deployments. It includes built-in support for Docker containers, Kubernetes orchestration, and integration with cloud platforms like Azure.

- **Tooling and Ecosystem:** .NET Core provides a rich set of tools and libraries for application development, including the .NET CLI, Visual Studio IDE, Visual Studio Code, and various third-party tools and extensions. The ecosystem around .NET Core continues to expand, offering support for tasks such as testing, CI/CD, and monitoring.
- **Security and Reliability:** .NET Core emphasizes security and reliability, with features like built-in security mechanisms, memory management, and error handling. Additionally, Microsoft regularly releases updates and patches to address security vulnerabilities and improve stability.

ASP .NET Core

ASP.NET Core is the new and totally re-written version of the ASP.NET web framework. It is a free, open-source, and cross-platform framework for building cloud-based applications, such as web apps, IoT apps, and mobile backends. It is designed to run on the cloud as well as on-premises.

Same as .NET Core, it was architected modular with minimum overhead, and then other more advanced features can be added as NuGet packages as per application requirement. This results in high performance, require less memory, less deployment size, and is easy to maintain.

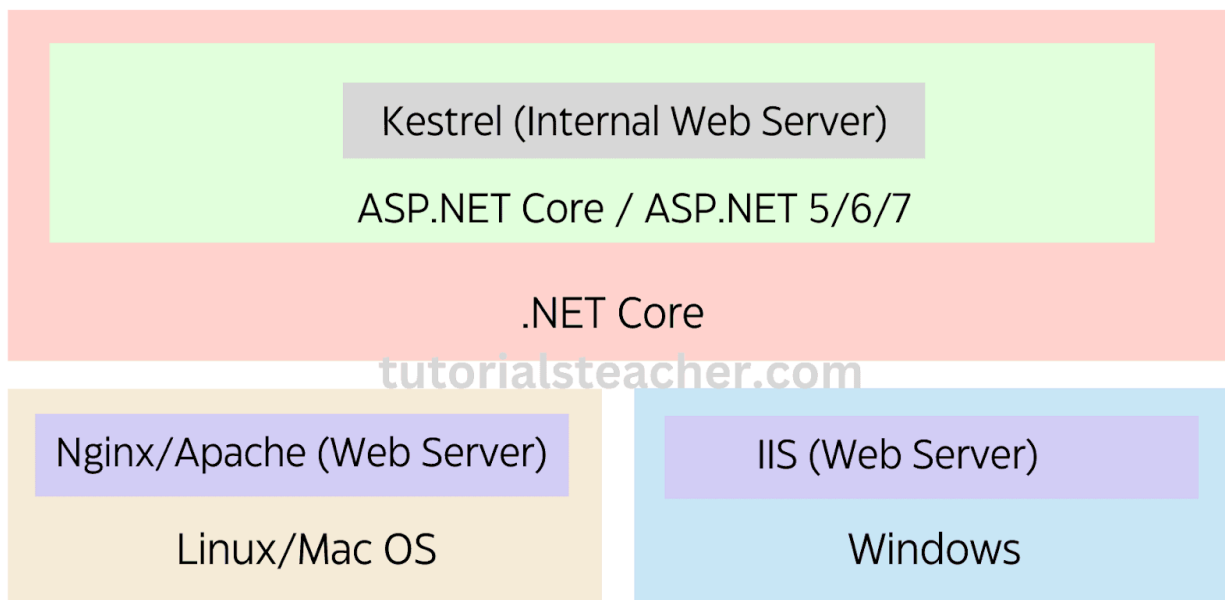
Why ASP.NET Core?

ASP.NET Core has the following advantages over traditional ASP.NET web framework:

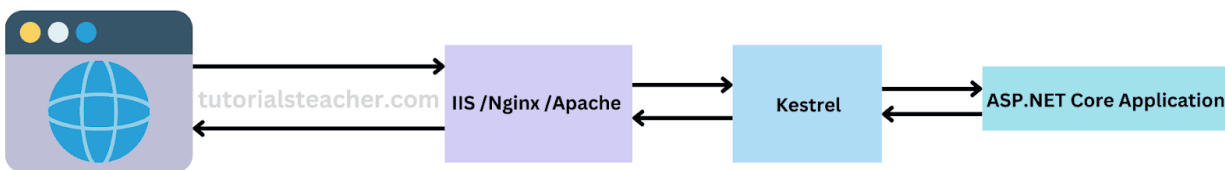
- **Supports Multiple Platforms:** ASP.NET Core applications can run on Windows, Linux, and Mac. So you don't need to build different apps for different platforms using different frameworks.
- **Fast:** ASP.NET Core no longer depends on System.Web.dll for browser-server communication. ASP.NET Core allows us to include packages that we need for our application. This reduces the request pipeline and improves performance and scalability.
- **IoC Container:** It includes the built-in IoC container for automatic dependency injection which makes it maintainable and testable.
- **Integration with Modern UI Frameworks:** It allows you to use and manage modern UI frameworks such as AngularJS, ReactJS, Umber, Bootstrap, etc. using Bower (a package manager for the web).
- **Hosting:** ASP.NET Core web application can be hosted on multiple platforms with any web server such as IIS, Apache etc. It is not dependent only on IIS as a standard .NET Framework.
- **Code Sharing:** It allows you to build a class library that can be used with other .NET frameworks such as .NET Framework 4.x or Mono. Thus a single code base can be shared across frameworks.
- **Side-by-Side App Versioning:** ASP.NET Core runs on .NET Core, which supports the simultaneous running of multiple versions of applications.
- **Smaller Deployment Footprint:** ASP.NET Core application runs on .NET Core, which is smaller than the full .NET Framework. So, the application which uses only a part of .NET CoreFX will have a smaller deployment size. This reduces the deployment footprint.

Cross-platform ASP.NET Core

ASP.NET Core applications run on the Windows, Mac or Linux OS using the .NET Core framework (now known as .NET 5/6/7). ASP.NET Core web application can be deployed on these OS because ASP.NET Core web application are self-hosted using internal web server called Kestrel. The platform specific web server such as IIS is used as external webserver that sends requests to the internal webserver Kestrel



The following image shows how the http requests will be handled for ASP.NET Core web applications..

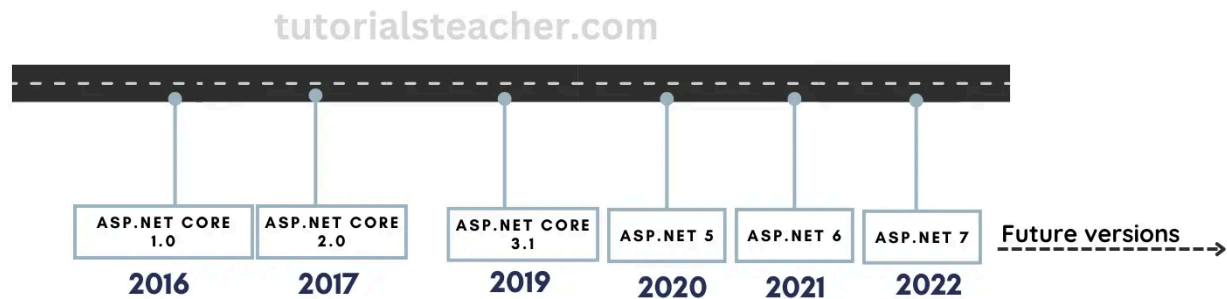


ASP.NET Core Version History

Microsoft launched ASP.NET web framework along with .NET Framework 1.0 in 2002. It was designed to run on Windows platform.

In 2016, Microsoft launched ASP.NET Core framework which can run on Windows, Mac, and Linux using .NET Core framework. It had many advantages over traditional ASP.NET framework.

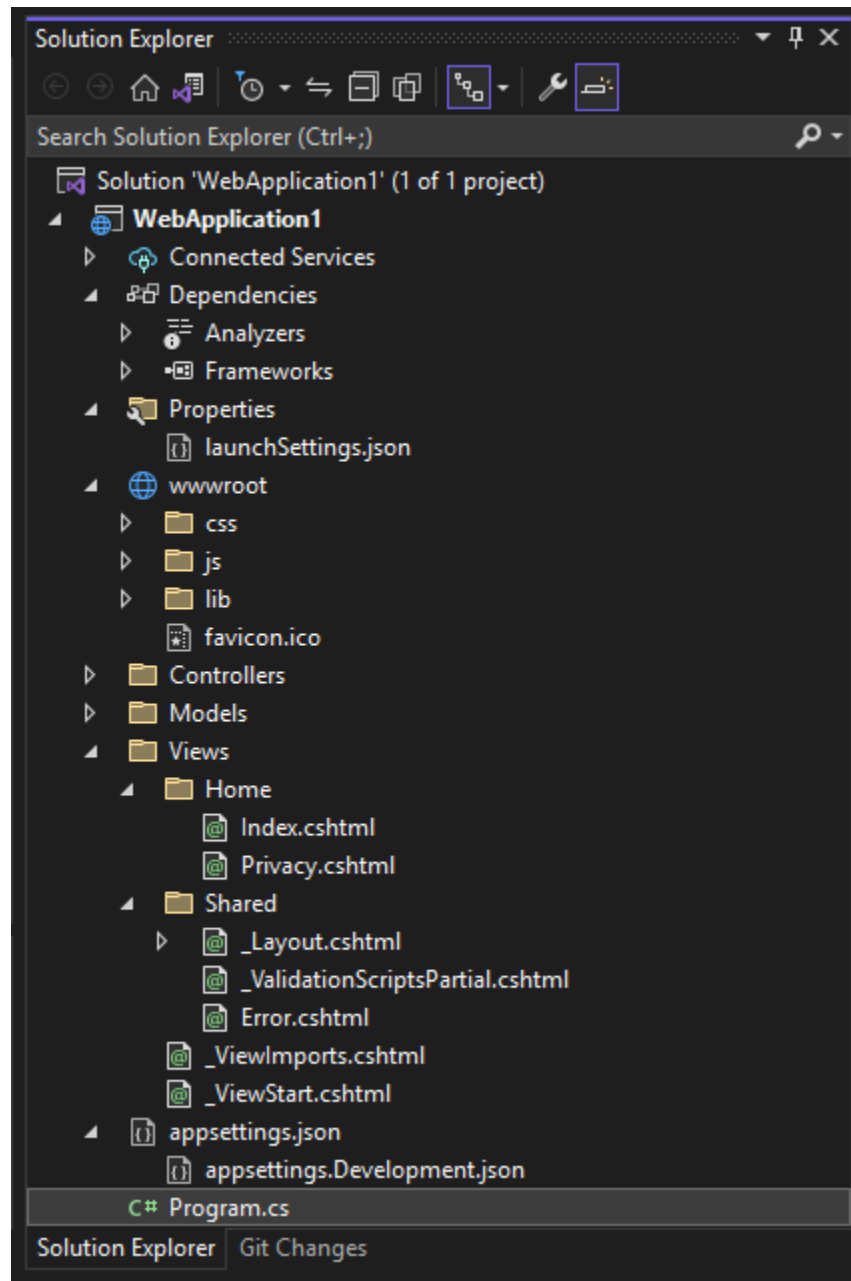
The next version of ASP.NET Core after version 3.1 was named as ASP.NET 5 which is unified framework for all types of application. So, ASP.NET 5 and later versions are ASP.NET Core framework only. They just named back to original name.



Version	Visual Studio	Release Date	End of Support
ASP.NET 7 - Latest Version	Visual Studio 2022 v17.4	Nov 8, 2022	May 14, 2024
ASP.NET 6 (LTS)	Visual Studio 2022	Nov 9, 2021	Nov 12, 2024
ASP.NET 5	Visual Studio 2019	Nov 10, 2020	May 10, 2022
ASP.NET Core 3.1 (LTS)	Visual Studio 2019	Dec 3, 2019	Dec 13, 2022
ASP.NET Core 3.0	Visual Studio 2019	Sep 23, 2019	Mar 3, 2020
ASP.NET Core 2.1 (LTS)	Visual Studio 2017, 2019	May 30, 2018	Aug 21, 2021
ASP.NET Core 2.0	Visual Studio 2017, 2019	Aug 14, 2017	Oct 1, 2018
ASP.NET Core 1.0	Visual Studio 2017	Jun 27, 2016	Jun 27, 2019

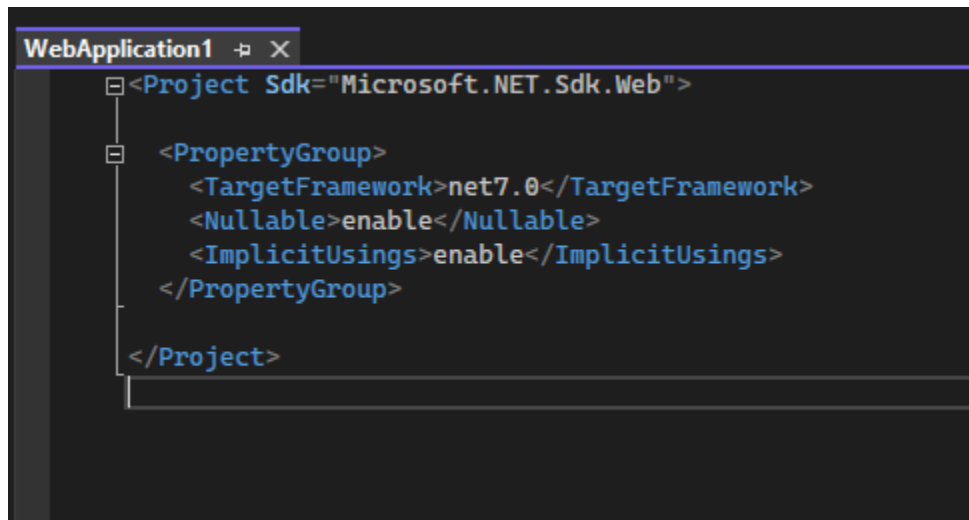
Project Structure

A default project structure of the ASP.NET Core MVC application in Visual Studio.



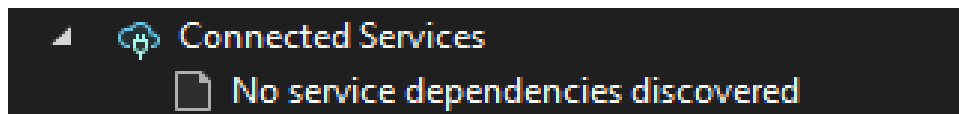
- **.csproj File**

Double click on the project name in Solution Explorer to open .csproj file in the editor. Right-click on the project and then click on Edit Project File in order to edit the .csproj file. As shown in the following image.



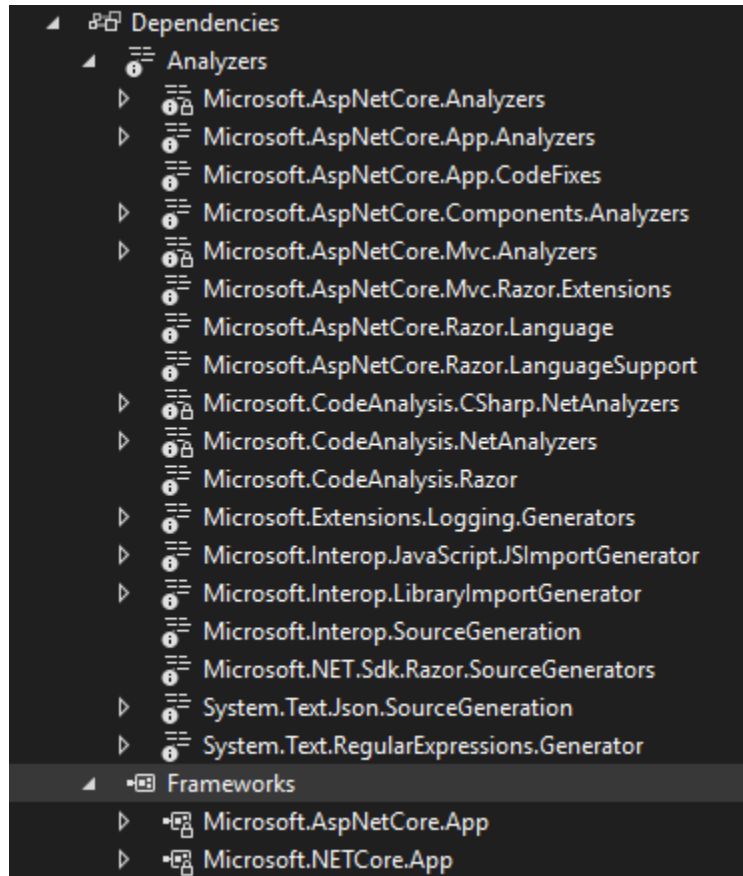
- **Connected Services**

The 'Connected Services' node contains the list of external services, APIs, and other data sources. It helps in the integration with various service providers, such as Azure, AWS, Google Cloud, and third-party services like authentication providers or databases. We are not using any service yet so it will be empty for now.



- **Dependencies**

The Dependencies node contains the list of all the dependencies that your project relies on, including NuGet packages, project references, and framework dependencies.



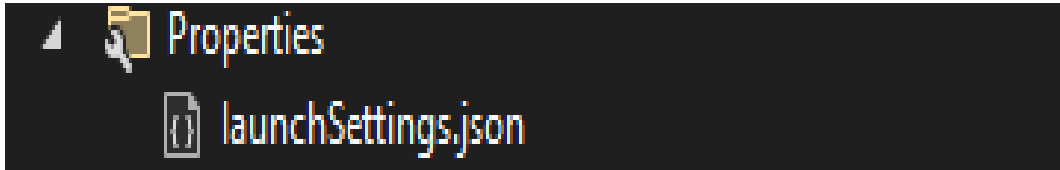
It contains two nodes, Analyzers and Frameworks.

Analyzers are extensions for static code analysis. They help you to enforce coding standards, identify code quality issues, and detect potential problems in your code. Analyzers can be custom rules or third-party analyzers provided by NuGet packages.

Frameworks node contains the target framework that your project is designed to run on. We have created an ASP.NET Core MVC application. So, it contains two frameworks, the .NET Core (Microsoft.NETCore.App) and ASP.NET Core (Microsoft.AspNetCore.App) framework. Click on any node and press F4 to see its version, file path, etc.

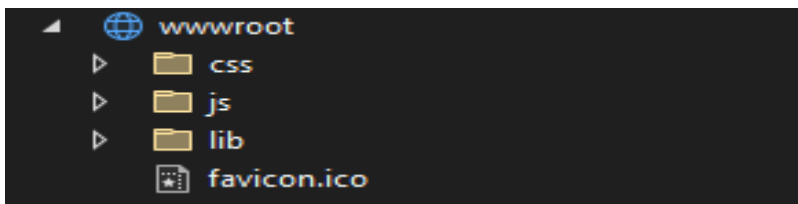
- **Properties**

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**

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.



- **Controllers, Models, Views**

- **Model:** Model represents the shape of the data. A class in C# is used to describe a model. Model objects store data retrieved from the database.

Model represents the data.

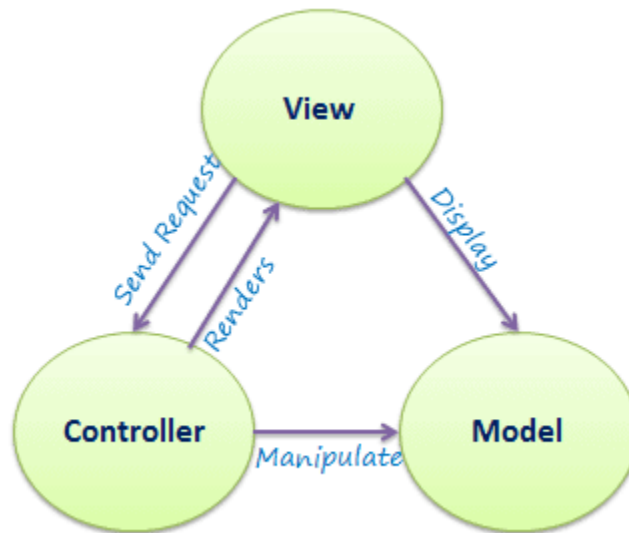
- **View:** View in MVC is a user interface. View display model data to the user and also enables them to modify them. View in ASP.NET MVC is HTML, CSS, and some special syntax (Razor syntax) that makes it easy to communicate with the model and the controller.

View is the User Interface.

- **Controller:** The controller handles the user request. Typically, the user uses the view and raises an HTTP request, which will be handled by the controller. The controller processes the request and returns the appropriate view as a response.

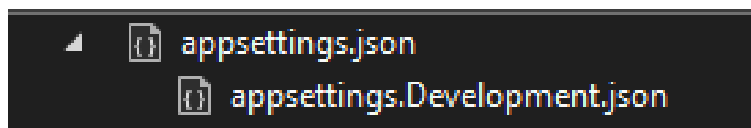
Controller is the request handler.

The following figure illustrates the interaction between Model, View, and Controller.



- **appsettings.json**

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



- **program.cs**

The last file 'program.cs' is an entry point of an application. ASP.NET Core web application is a console application that builds and launches a web application.

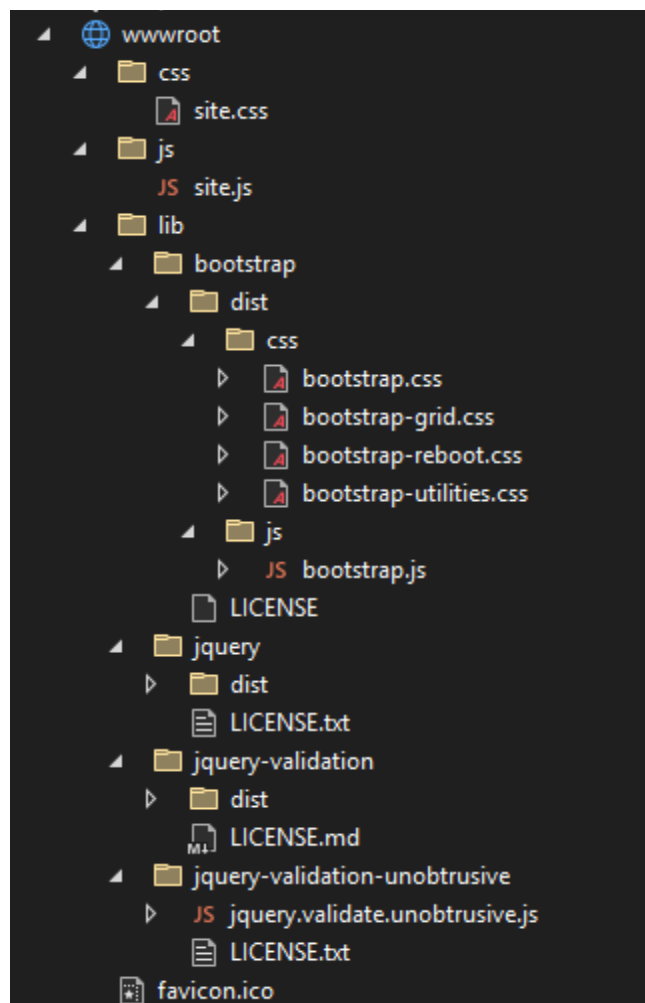


wwwroot Folder

By default, the **wwwroot** folder in the ASP.NET Core project is treated as a web root folder. Static files can be stored in any folder under the web root and accessed with a relative path to that root.

In the standard ASP.NET application, static files can be served from the root folder of an application or any other folder under it. This has been changed in ASP.NET Core. Now, only those files that are in the web root - wwwroot folder can be served over an http request. All other files are blocked and cannot be served by default.

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.



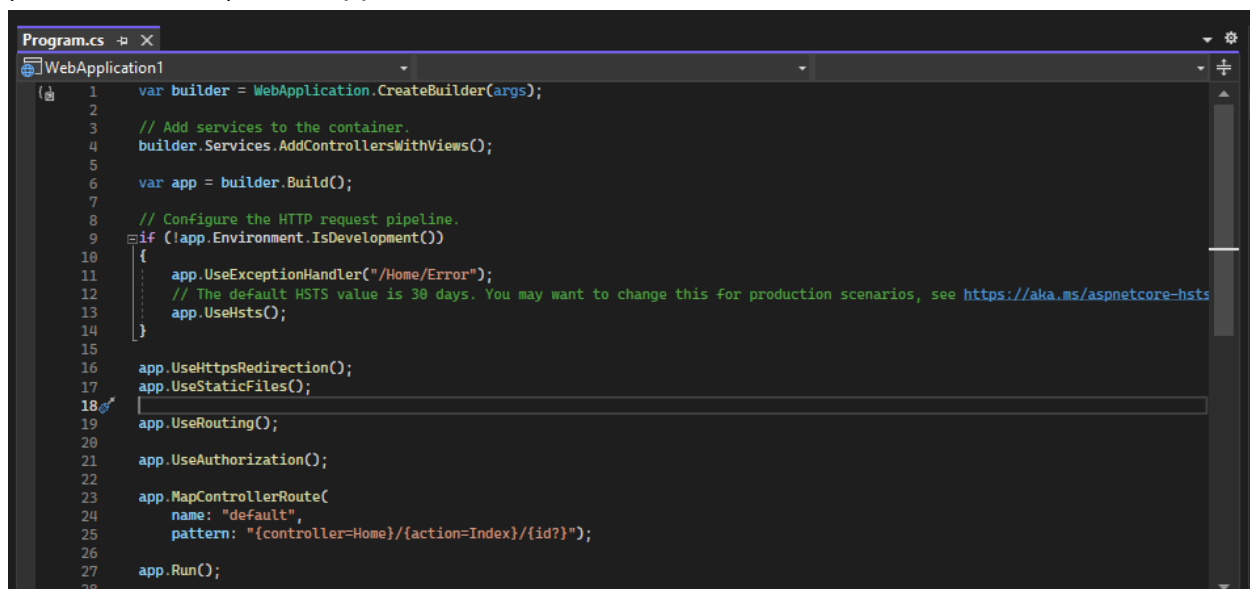
You can access static files with base URL and file name. For example, we can access above app.css file in the css folder by `http://localhost:<port>/css/app.css`.

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.

Every ASP.NET Core web applications starts like a console application then turn into web application. When you press F5 and run the application, it starts the executing code in the Program.cs file.

The following is the default Program.cs file created in Visual Studio for ASP.NET 7 (ASP.NET Core) MVC application.



```
1 var builder = WebApplication.CreateBuilder(args);
2
3 // Add services to the container.
4 builder.Services.AddControllersWithViews();
5
6 var app = builder.Build();
7
8 // Configure the HTTP request pipeline.
9 if (app.Environment.IsDevelopment())
10 {
11     app.UseExceptionHandler("/Home/Error");
12     // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts
13     app.UseHsts();
14 }
15
16 app.UseHttpsRedirection();
17 app.UseStaticFiles();
18
19 app.UseRouting();
20
21 app.UseAuthorization();
22
23 app.MapControllerRoute(
24     name: "default",
25     pattern: "{controller=Home}/{action=Index}/{id?}");
26
27 app.Run();
28
```

Let's understand program.cs code step-by-step:

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. You 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.

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

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 your application can use MVC architecture.

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

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.

Now, using this `WebApplication` object `app`, you 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");
}
```

Note that 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 incoming request. It turns a console application into an MVC application based on the provided configuration.

So, `program.cs` contains codes that sets up all necessary infrastructure for your application.

Startup.cs

This class is described by its name: startup. It is the entry point of the application. It configures the request pipeline which handles all requests made to the application. The inception of the startup class is in the OWIN (Open Web Interface for .NET) application that is a specification to reduce dependency of applications on the server. It contains application configuration related items.

It is not necessary that the class name be "Startup". The ASP.net core application is a Console app and we have to configure a web host to start listening. The "Program" class does this configuration.

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = new WebHostBuilder()
            .UseKestrel()
            .UseContentRoot(Directory.GetCurrentDirectory())
            .UseStartup<Startup>()
            .Build();
        host.Run();
    }
}
```

Startup Class Constructor

We can also specify the constructor of the startup class. The startup class has constructor with one or three parameters.

```
public Startup(IHostingEnvironment env)
{
}

public Startup(IApplicationBuilder appenv, IHostingEnvironment env, ILoggerFactory lo
{
}
```

- **IApplicationBuilder** is an interface that contains properties and methods related to current environment. It is used to get the environment variables in application.
- **IHostingEnvironment** is an interface that contains information related to the web hosting environment on which application is running. Using this interface method, we can change behavior of application.
- **IloggerFactory** is an interface that provides configuration for the logging system in Asp.net Core. It also creates the instance of logging system.

The startup class contains two methods: **ConfigureServices** and **Configure**.

ConfigureServices Method

- Declaration of this method is not mandatory in startup class. This method is used to configure services that are used by the application. When the application is requested for the first time, it calls ConfigureServices method. This method must be declared with a public access modifier, so that environment will be able to read the content from metadata.
- ASP.net core has built-in support for Dependency Injection. We can add services to DI container using this method. Following are ways to define ConfigureServices method in startup class.

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

In case of specific class or library of project that would like to add to DI container, we will use the IServiceCollection. In the above example, I have just add MVC service to the ConfigureServices method.

Configure Method

- This method is used to define how the application will respond on each HTTP request i.e. we can control the ASP.net pipeline. This method is also used to configure middleware in HTTP pipeline. This method accept IApplicationBuilder as a parameter. This method may accept some optional parameter such as IHostingEnvironment and ILoggerFactory. Whenever any service is added to ConfigureServices method, it is available to use in this method.

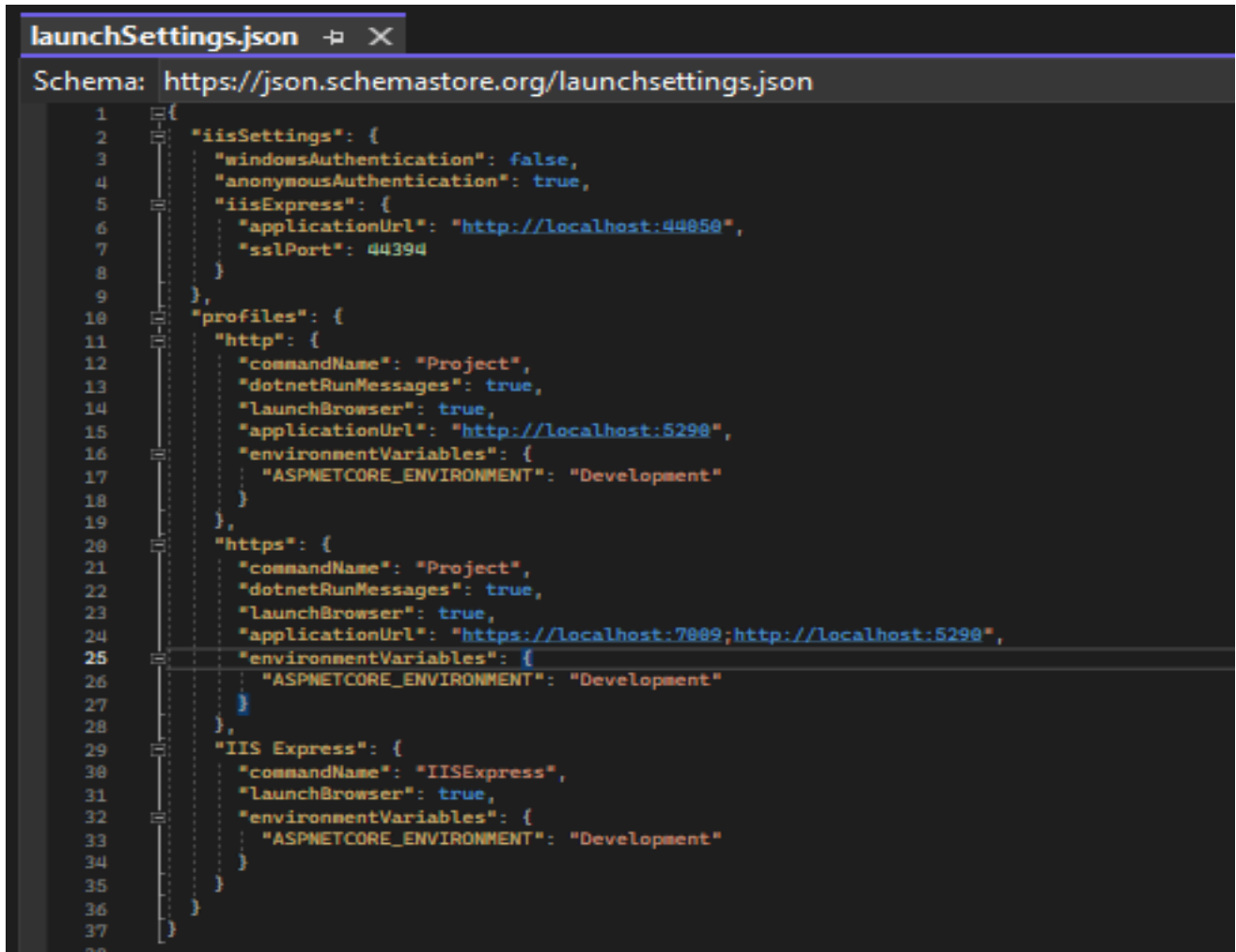
```
public void Configure(IApplicationBuilder app)
{
    app.UseMvc();

    app.Run(context => {
        return context.Response.WriteAsync("Hello Readers!");
    });
}
```

The Startup class is mandatory. With the help of this class we can configure the environment in our ASP.net Core application. We can use Constructor and two different methods: ConfigureServices and Configure for setting up the environment. This class creates services and injects services as dependencies so the rest of the application can use these dependencies. The ConfigureServices used to register the service and Configure method allow us to add middleware and services to the HTTP pipeline. This is the reason ConfigureServices method calls before Configure method.

LaunchSettings.json

- In ASP.NET Core, the launchSettings.json file is a configuration file used to configure various aspects of how our application should be launched and debugged during development. This file is typically found in the “Properties” folder of our ASP.NET Core project and is used primarily by the Visual Studio IDE and the .NET CLI (Command-Line Interface).
- 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. The most important point you need to remember is that this 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.



```
1 {
2   "iisSettings": {
3     "windowsAuthentication": false,
4     "anonymousAuthentication": true,
5     "iisExpress": {
6       "applicationUrl": "http://localhost:44050",
7       "sslPort": 44394
8     }
9   },
10  "profiles": {
11    "http": {
12      "commandName": "Project",
13      "dotnetRunMessages": true,
14      "launchBrowser": true,
15      "applicationUrl": "http://localhost:5290",
16      "environmentVariables": {
17        "ASPNETCORE_ENVIRONMENT": "Development"
18      }
19    },
20    "https": {
21      "commandName": "Project",
22      "dotnetRunMessages": true,
23      "launchBrowser": true,
24      "applicationUrl": "https://localhost:7009;http://localhost:5290",
25      "environmentVariables": {
26        "ASPNETCORE_ENVIRONMENT": "Development"
27      }
28    },
29    "IIS Express": {
30      "commandName": "IISExpress",
31      "launchBrowser": true,
32      "environmentVariables": {
33        "ASPNETCORE_ENVIRONMENT": "Development"
34      }
35    }
36  }
37 }
```

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.
- **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 ASP.NET Core Hosting Model 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.

- **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.

- **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.

When to Use ASP.NET Core LaunchSettings.json File?

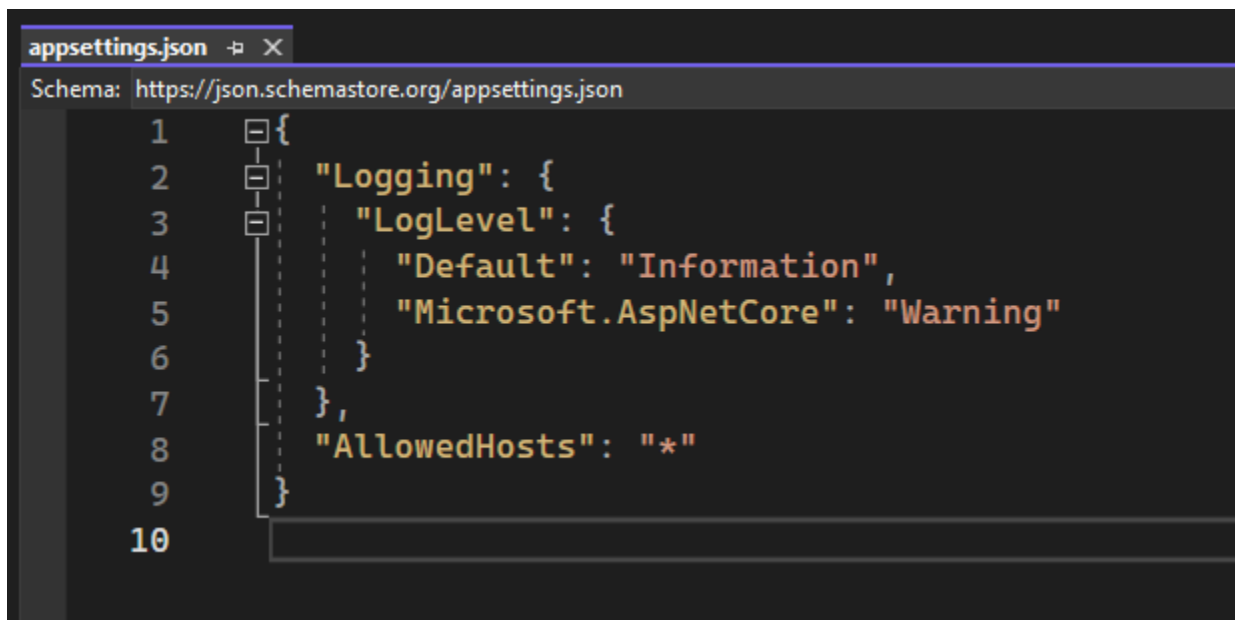
The `launchSettings.json` file in ASP.NET Core is used primarily in the development environment for configuring how an application is launched and its initial environment settings. Understanding when and why to use this file can help streamline your development workflow. Here are some scenarios and purposes for using the `launchSettings.json` file:

- **Defining Environment Variables:** It's common to use different settings for development, staging, and production environments. The `launchSettings.json` file allows you to define environment variables, like **ASPNETCORE_ENVIRONMENT**, which can be set to Development, Staging, or Production. This helps in testing how your application behaves under different configurations without changing the code.
- **Setting Up Multiple Launch Profiles:** You can set up different launch profiles for various scenarios, like one for launching with IIS, another for Kestrel, or a profile for a Docker container. This flexibility allows you to switch contexts and test your application in different environments quickly.
- **Customizing Application URLs:** For development purposes, you might want to run your application on specific ports or with certain hostnames. The `launchSettings.json` file allows you to define these URLs to test your application as it would appear on different domains or ports.

- **Controlling Browser Launch and URLs:** If your application is web-based, you can configure whether a browser should automatically launch when you start your application. You can also specify which URL the browser should open to.
- **Suppressing Command-Line Messages:** Using the `dotnetRunMessages` property, you can control the verbosity of the messages displayed in the console when running the application using `dotnet run`. This can be useful for decluttering your development environment.
- **Development-Time Settings Only:** It's important to remember that `launchSettings.json` is only for development settings. These settings do not apply when the application is deployed to production. For production, you would typically use other configuration methods like `appsettings.json`, environment variables on the server, or Azure App Service settings.
- **Testing Under Different Conditions:** If you want to test how your application behaves under different configurations, such as with different connection strings, API endpoints, or feature flags, you can quickly switch between these configurations.
- **Integrating with IDEs:** When working with IDEs like Visual Studio, `launchSettings.json` provides an easy way to configure the IDE's behavior when debugging and running your application.

appSettings.json

- In the Asp.Net Core application we may not find the web.config file. Instead we have a file named "appsettings.json". So to configure the settings like database connections, Mail settings or some other custom configuration settings we will use the "appsettings.json".
- The appsettings in Asp.net core have different configuration sources as shown below.
 - appsettings.json File
 - Environment Variable
 - User Secrets
 - Command Line Arguments
- 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.



```
appsettings.json  ▸ ×
Schema: https://json.schemastore.org/appsettings.json
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft.AspNetCore": "Warning"
6      }
7    },
8    "AllowedHosts": "*"
9  }
10
```

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.

Add New Keys

The appsettings.json file can be configured with Key and Value pair combinations. So, the Key will have single or multiple values.

```
• {  
•   "ConnectionStrings": {  
•     "MyDatabase": "Server=localhost;Initial  
Catalog=MySampleDatabase;Trusted_Connection=Yes;Multiple  
ActiveResultSets=true"  
•   }  
• }
```

Read Values from appsettings.json

To read the values from appsettings.json there are several ways to read. One of the simple and easy way to read the appsettings in Asp.net core is by using the IConfiguration using the namespace Microsoft.Extensions.Configuration.

In the below code, we have two method to read the value using IConfiguration extension. Inject the IConfiguration in the controller constructor and use the variable to get the section.

`"config.GetSection("ConnectionStrings").GetSection("MyDatabase").Value"`
will return the value as
`"Server=localhost;Initial Catalog=MySampleDatabase;Trusted_Connection=Yes;MultipleActiveResultSets=true"`.

Similarly,

`"config.GetValue<string>("ConnectionStrings:MyDatabase")"`
will return the value as
`"Server=localhost;Initial Catalog=MySampleDatabase;Trusted_Connection=Yes;MultipleActiveResultSets=true"`.

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
  }
}
```

