



- **.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 licence. It is a .NET Foundation project.

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

**Consistent across Architectures:** 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 platform such as mobile, desktop, web, cloud, IoT, machine learning, microservices, game, etc.

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

**Modular Architecture:** .NET Core supports modular architecture approach using NuGet packages. There are different NuGet packages for various features that can be added to the .NET Core project as needed. Even the .NET Core library is provided as a NuGet package. The NuGet package for the default .NET Core application model is Microsoft.NETCore.App.

This way, it reduces the memory footprint, speeds up the performance, and easy to maintain.

**CLI Tools:** .NET Core includes CLI tools (Command-line interface) for development and continuous-integration.

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

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

**High Performance:** .NET Core was optimised for performance, with features like Just-In-Time (JIT) compilation, which translated code into machine instructions at runtime for better execution speed.

**Unified Platform:** After .NET Core 3.1, Microsoft unified the .NET platforms, bringing together .NET Core, .NET Framework, and Xamarin into a single platform called ".NET" starting from .NET 5. This unified platform aimed to provide consistent APIs and runtime behaviour across different application types.

## 2. 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, requires less memory, less deployment size, and is easy to maintain.

- **Why ASP.NET Core?**

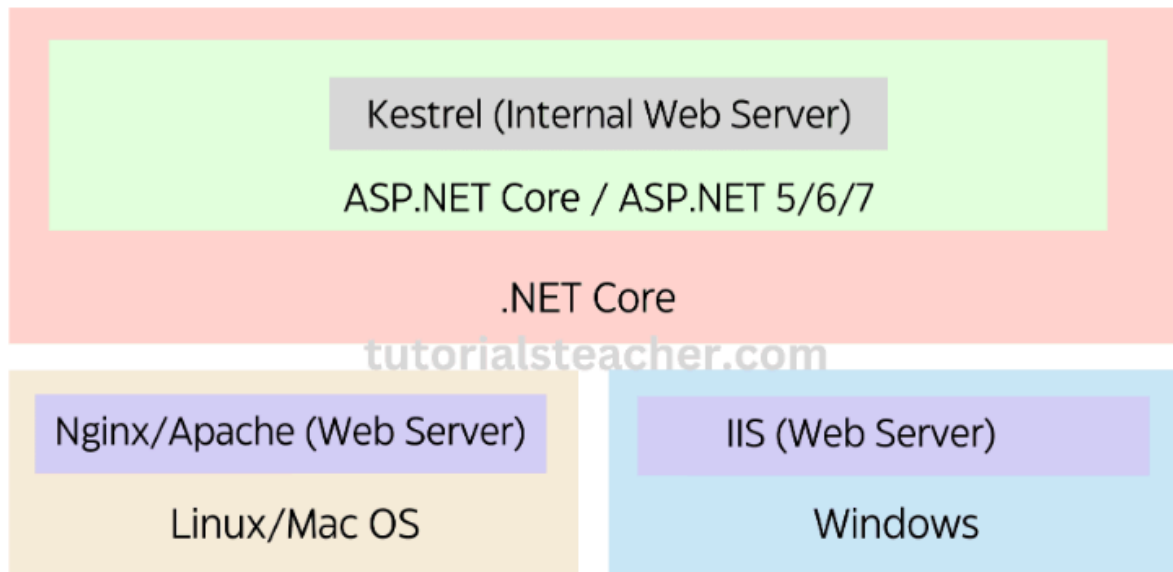
ASP.NET Core has the following advantages over traditional ASP.NET web framework:

- **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 applications can be deployed on these OS because ASP.NET Core web applications are self-hosted using an internal web server called Kestrel. The platform specific web server such as IIS is used as an external web server that sends requests to the internal web server Kestrel.



Cross-platform ASP.NET Core Framework

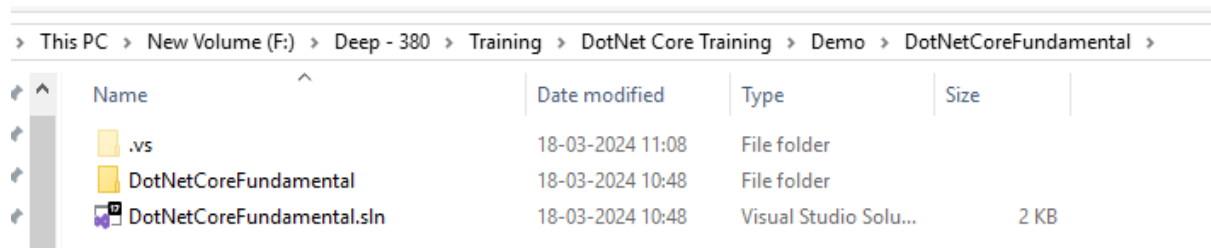
The following image shows how the http requests will be handled for ASP.NET Core web applications.



ASP.NET Core Web Request Handling

### 3. Project Structure

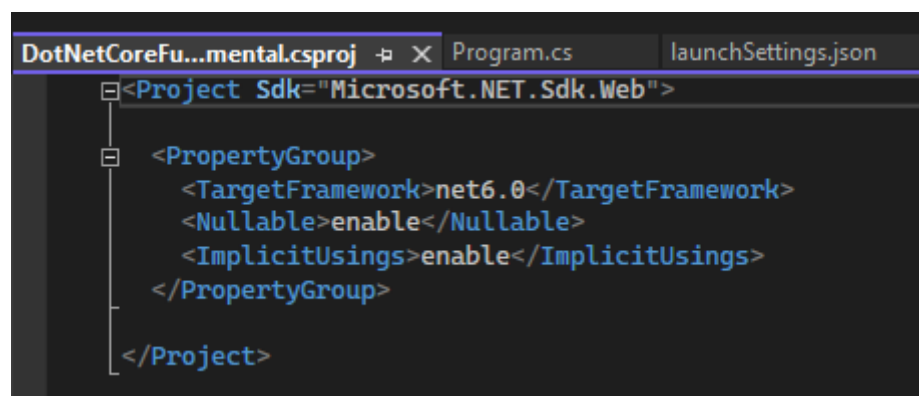
- **Solution File** :- Visual Studio creates the top-level solution file (.sln) that can contain one or more ASP.NET projects. Here, "Solution 'DotNetCoreFundamental'(1 of 1 project)" is the solution file. You can right-click on it and click on 'Open Folder in File Explorer' and see the 'DotNetCoreFundamental.sln' file is created. This is our solution file.



The screenshot shows a File Explorer window with the address bar path: This PC > New Volume (F:) > Deep - 380 > Training > DotNet Core Training > Demo > DotNetCoreFundamental >. The file list contains three items:

Name	Date modified	Type	Size
.vs	18-03-2024 11:08	File folder	
DotNetCoreFundamental	18-03-2024 10:48	File folder	
DotNetCoreFundamental.sln	18-03-2024 10:48	Visual Studio Solu...	2 KB

- **Project File** :- Under the solution node in the Solution Explorer, you can see our project node 'DotNetCoreFundamental'. All the files under this node will be for the 'DotNetCoreFundamental' project.



The screenshot shows the Visual Studio editor with the file 'DotNetCoreFundamental.csproj' open. The code content is as follows:

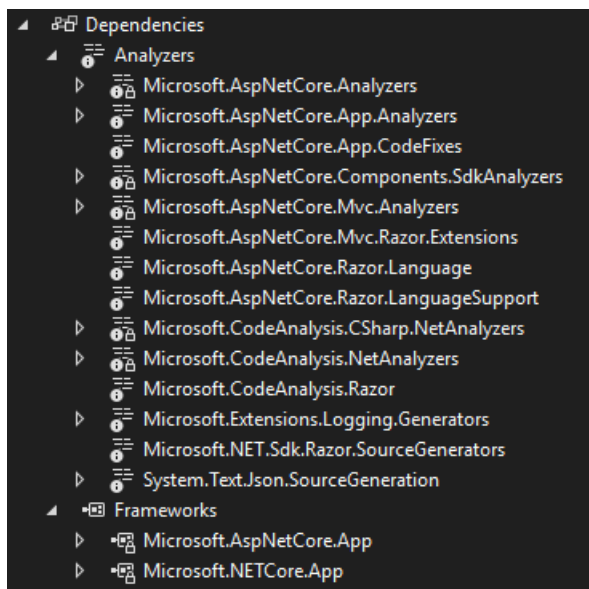
```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
</Project>
```

- **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.

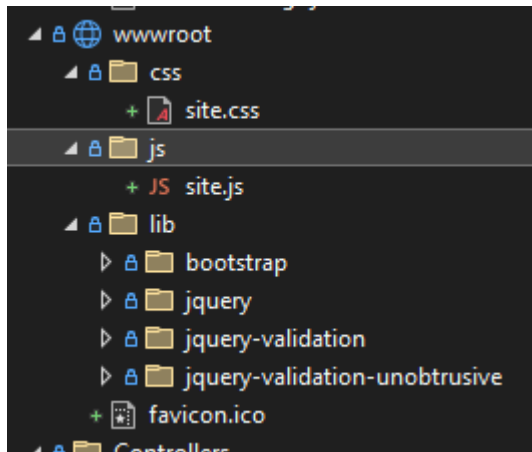
**Frameworks** node contains the target framework that your project is designed to run on. We have created an ASP.NET Core MVC application. So, it contains two frameworks, the .NET Core (Microsoft.NETCore.App) and ASP.NET Core (Microsoft.AspNetCore.App) framework. Click on any node and press F4 to see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



- **Properties** :- The Properties node includes launchSettings.json file which includes Visual Studio profiles of debug settings. launchSettings.json helps developers to configure the debugging and launch profiles of their ASP.NET (also known as ASP.NET Core) applications for different environments such as development, staging, production, etc.. The following is a default launchSettings.json file.

```
DotNetCoreFun...mental.csproj  Program.cs  launchSettings.json  DotNetCoreFundamental
Schema: https://json.schemastore.org/launchsettings.json
1  {
2    "iisSettings": {
3      "windowsAuthentication": false,
4      "anonymousAuthentication": true,
5      "iisExpress": {
6        "applicationUrl": "http://localhost:28182",
7        "sslPort": 44395
8      }
9    },
10   "profiles": {
11     "DotNetCoreFundamental": {
12       "commandName": "Project",
13       "dotnetRunMessages": true,
14       "launchBrowser": true,
15       "applicationUrl": "https://localhost:7188;http://localhost:5248",
16       "environmentVariables": {
17         "ASPNETCORE_ENVIRONMENT": "Development"
18       }
19     },
20     "IIS Express": {
21       "commandName": "IISExpress",
22       "launchBrowser": true,
23       "environmentVariables": {
24         "ASPNETCORE_ENVIRONMENT": "Development"
25       }
26     }
27   }
28 }
29
```

- **wwwroot** :- 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. All the css, JavaScript, and external library files should be stored here which are being referenced in the HTML file.



- **Controllers, Models, and Views** :- The Controllers, Models, and Views folders include controller classes, model classes and cshtml or vbhtml files respectively for MVC application.
- **appSettings.json** :- The appsettings.json file is a configuration file commonly used in .NET applications, including ASP.NET Core and ASP.NET 5/6, to store application-specific configuration settings and parameters. It allows developers to use JSON format for the configurations instead of code, which makes it easier to add or update settings without modifying the application's source code.



- **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.

```
public static void Main(string[] args)
{
    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.ms/aspnetcore-hsts
        app.UseHsts();
    }

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

    app.UseRouting();

    app.UseAuthorization();

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

    app.Run();
}
```



## 4. 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 the above app.css file in the css folder by **http://localhost:<port>/css/app.css**.
- Remember, you need to include a middleware for serving static files using **app.UseStaticFiles()** method in the program.cs file.

## 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.

Every ASP.NET Core web application starts like a console application then turns into a 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 6 (ASP.NET Core) MVC 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.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();

// Used for serving files of wwwroot folder
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.
- The `CreateBuilder()` method sets up 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.

- 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.
- The `builder.Build()` method returns the object of `WebApplication` using which you can configure the request pipeline using middleware and a 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.
- Note that the method starts with "Use" word which means it configures the middleware.
- 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. res the static files, routing, and authorization middleware respectively.
- Finally, the `app.run()` method runs the application, start listening to 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.

## 6. Startup.cs

ASP.NET Core apps use a Startup class, which is named Startup by convention. The Startup class:

Optionally includes a ConfigureServices method to configure the app's services. A service is a reusable component that provides app functionality. Services are registered in ConfigureServices and consumed across the app via dependency injection (DI) or ApplicationServices.

Includes a Configure method to create the app's request processing pipeline. ConfigureServices and Configure are called by the ASP.NET Core runtime when the app starts:

### Startup.cs

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

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

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

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

        app.UseRouting();
```

```

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}

```

### **Program.cs**

```

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>();
            });
}

```

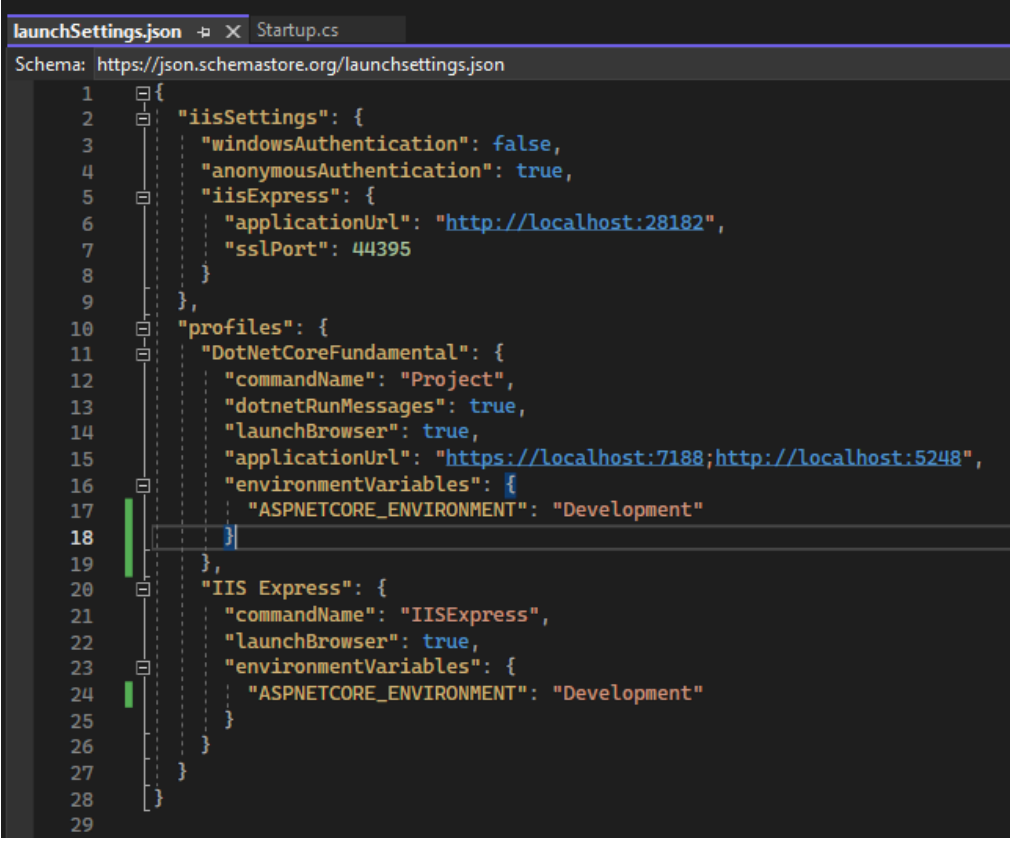
## 7. launchSettings.json

launchSettings.json is a configuration file used in ASP.NET Core applications to define how the application should be launched and debugged. This file is typically found in the Properties folder of the project.

Here's an explanation of the key components and settings within launchSettings.json:

- **profiles:** This section contains profiles for different environments like development, staging, and production. Each profile defines settings such as the application URL, environment variables, and debugging options.
- **applicationUrl:** Specifies the URL where the application will be hosted when launched in the specified environment. This is useful for defining different URLs for different environments or specifying custom ports.
- **commandName:** Defines the command to execute when launching the application. For ASP.NET Core web applications, the command is usually set to "web", indicating that the application should be launched as a web application.
- **launchBrowser:** Specifies whether the default web browser should be launched when starting the application. This is helpful for automatically opening the browser to view the application.
- **environmentVariables:** Allows you to set environment variables that will be available to the application when launched. This can be used to configure the application for different environments or to provide sensitive configuration data securely.
- **inspectUri:** Defines the URL where the debugger can attach to the application for debugging purposes. This is useful when debugging the application using an external debugger.

Here's a basic example of a launchSettings.json file:



```
1  {
2    "iisSettings": {
3      "windowsAuthentication": false,
4      "anonymousAuthentication": true,
5      "iisExpress": {
6        "applicationUrl": "http://localhost:28182",
7        "sslPort": 44395
8      }
9    },
10   "profiles": {
11     "DotNetCoreFundamental": {
12       "commandName": "Project",
13       "dotnetRunMessages": true,
14       "launchBrowser": true,
15       "applicationUrl": "https://localhost:7188;http://localhost:5248",
16       "environmentVariables": {
17         "ASPNETCORE_ENVIRONMENT": "Development"
18       }
19     },
20     "IIS Express": {
21       "commandName": "IISExpress",
22       "launchBrowser": true,
23       "environmentVariables": {
24         "ASPNETCORE_ENVIRONMENT": "Development"
25       }
26     }
27   }
28 }
29
```

launchSettings.json X Startup.cs  
Schema: <https://json.schemastore.org/launchsettings.json>

## 8. appsettings.json

appSettings.json is a configuration file commonly used in ASP.NET Core applications to store application settings in a structured format. It's a JSON file that allows developers to store various configuration values such as database connection strings, API keys, logging settings, and other parameters that control the behaviour of the application.

Here's a basic example of what an appSettings.json file might look like:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "System": "Error"
    }
  },
  "ConnectionStrings": {
    "DefaultConnection":
"Server=myServerAddress;Database=myDataBase;User=myUsername;Password=m
yPassword;"
  },
  "AppSettings": {
    "ApiKey": "your-api-key",
    "MaxItemsPerPage": 10
  }
}
```

- **Logging:** This section defines logging settings for the application. It specifies the default log level for different log sources like Microsoft and System.
- **ConnectionStrings:** This section typically stores database connection strings. In this example, there is one connection string named DefaultConnection, which contains the connection details for a database.
- **AppSettings:** This section is custom and can store any application-specific settings. In this example, it contains an API key (ApiKey) and a setting for the maximum number of items per page (MaxItemsPerPage).