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

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

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

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

There are some limitations to the .NET Framework. For example, it only runs on the Windows OS Platform. Also, you need to use different .NET APIs for different windows devices such as Windows Desktop, Windows Store, Windows Phone, and Web Applications

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Machine:** (Windows, Mac, Linux)

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

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

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

The .NET Core can be installed in two ways:

1. **By Installing Visual Studio 2019/2022**
2. **By Installing .NET Core SDK**

**Key differences between ASP.NET Core and ASP.NET**

|  |  |  |
| --- | --- | --- |
| Feature | ASP.NET Core | ASP.NET |
| Platform | Cross-platform (Windows, Linux, macOS) | Windows only |
| Framework Size | Modular and lightweight | Larger and more complex |
| Performance | Generally faster and more efficient | Potentially slower due to larger size |
| Architecture | Built on .NET Core runtime | Built on full [.NET Framework](https://www.scholarhat.com/tutorial/net) |
| Model-View-Controller (MVC) | Integrated with Web API in a unified framework | Separate MVC framework and Web API |
| Configuration | More flexible and streamlined | Complex and XML-based configuration |
| Open Source | Yes | Primarily closed-source, with some open-source components |
| Microsoft Support | Fully supported by Microsoft | Limited support for older ASP.NET versions |
| Suitable for | Modern web apps, cloud deployments, microservices | Legacy applications, Windows-specific deployments |

**What Are Server GC and Workstation GC?**

.NET Core supports two garbage collection modes: workstation GC and server GC. The optimization of garbage collection depends on the particular application or workload and, therefore, requires the implementation of different modes.

The default garbage collection mode used by most .NET Core applications is the workstation GC, specifically for desktop applications, such as Windows Forms, WPF, and console applications.

Server Garbage Collector (GC) or Server GC, as it is called, is specifically designed for performance-critical server applications running on multi-core CPUs. Because Server GC mode isn't enabled by default, we can activate it in the application settings file.

You can set the GC mode in the project file or in the runtimeconfig.json file. The following code snippet shows how you can turn on Server GC for your ASP.NET Core application.

<PropertyGroup>

<ServerGarbageCollection>true</ServerGarbageCollection>

</PropertyGroup>

**What Is Startup Class And Program.cs In ASP.NET Core**

Program.cs is where the application starts. Program.cs file will work the same as the Program.cs file in traditional console application of .NET Framework. Program.cs file is the entry point of the application, which contains the Main method. The Main method **creates a web host instance** that uses **the Startup class to configure the application**.

Global.asax is no longer in the ASP.NET Core application. The Startup.cs file is a replacement of the Global.asax file in ASP.NET Core.

The Startup class is executed after the Program class. The Startup class has a constructor, ConfigureServices method, and Configure method. The constructor is executed first, then the ConfigureServices method, and finally the Configure method.

**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>();

});

}

**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();

});

}

}

In the Startup class, we actually do two things:

**ConfigureService Method**

The ConfigureService method is used to configure all the services that you want to use in this application. The ConfigureService method takes IServiceCollection as an input parameter. The IServiceCollection interface belongs to Microsoft.Extensions.DependencyInjection namespace. So, whenever we inject any kind of method to this IServiceCollection object that means we are injecting the service to the built-in dependency injection.

**Configure Method**

The Configure method is used to configure the HTTP request processing pipeline of the application. In other words, we can say that it will configure all the middleware that you want to use in your application. The Configure method takes two parameters i.e. IApplicationBuilder and IWebHostEnvironment instance. The IApplicationBuilder interface belongs to Microsoft.AspNetCore.Builder namespace and IWebHostEnvironment interface belong to Microsoft.AspNetCore.Hosting namespace.

The IApplicationBuilder Defines a class that provides the mechanisms to configure an application’s request pipeline. On the other hand, the IWebHostEnvironment Provides information about the web hosting environment an application is running in.

[**AddTransient, AddScoped and AddSingleton Services**](https://stackoverflow.com/questions/38138100/addtransient-addscoped-and-addsingleton-services-differences) **in Asp.Net Core**

##### AddTransient: Creates a new instance of the service every time it's requested. This is best for lightweight, stateless services.

##### AddScoped: Create a same instance of the service within a request, but different across different requests.

##### AddSingleton: Creates a single instance of the service for the entire application, and then reuses that instance for each request.

builder.Services.AddScoped<IAuthService, AuthService>();

builder.Services.AddTransient<ICronJobService, CronJobService>();

builder.Services.AddSingleton<ILoggingService, LoggingService>();

**When to use which Service**

##### Singleton approach => We can use this for logging service and email service.

##### Scoped approach => This is a better option when you want to maintain a state within a request.

##### Transient approach =>  Use this approach for the lightweight service with little or no state.

##### 

##### Implementing Dependency Injection In .NET Core

##### To use dependency injection in .NET Core, we need to do the following:

##### Create an interface that defines the contract for our dependency.

##### Create a class that implements the interface.

##### Register the service with the DI container.

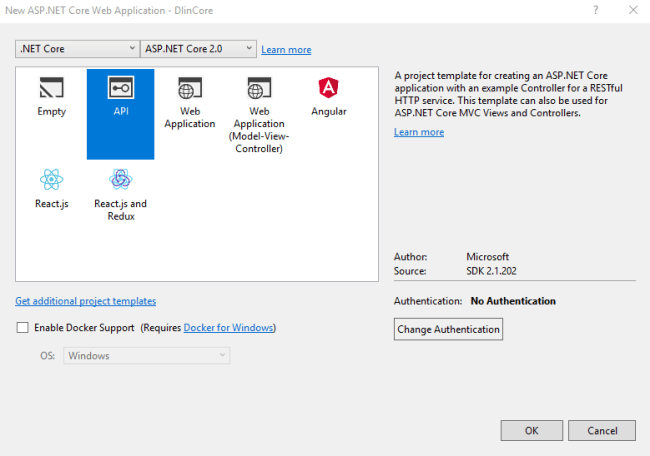
##### Inject the service into the class that needs it.

##### Here is an example of how to use dependency injection in .NET Core:

##### Consider a scenario where you want to fetch all the categories from the database and want to show that in the UI layer. So, you will create a service, i.e., a Web API which will be called by the UI layer. Now, in API, we need to create one GET method, which will call the repository, and the repository talks with the database. In order to call the repository, we need to create an instance of the same in the API GET method, which means it’s mandatory to create an instance of the repository for API. We can say the instance of the repository is the dependency of API. Let’s see how we can inject this dependency into our core Web API.

##### Open Visual Studio and create a new project

##### Select API as a template and press OK.



##### As we will fetch the categories, let’s create a category model with two fields - CategoryId and CategoryName.

namespace DIinCore

{

public class Category

{

public int CategoryId { get; set; }

public string CategoryName { get; set; }

}

}

##### Create an interface of the repository having the GetCategories method, which returns the list of category objects.

using System.Collections.Generic;

namespace DIinCore

{

public interface ICategoryRepository

{

List<Category> GetCategories();

}

}

##### Implement the preceding interface and return some sample data. As our target is to understand dependency injection, here, we are not going to fetch the data from the database but rather returning hard coded ones.

using System.Collections.Generic;

namespace DIinCore

{

public class CategoryRepository : ICategoryRepository

{

public List<Category> GetCategories()

{

List<Category> categories = new List<Category>();

Category category = new Category()

{ CategoryId = 1, CategoryName = "Category1" };

categories.Add(category);

category = new Category()

{ CategoryId = 2, CategoryName = "Category2" };

categories.Add(category);

return categories;

}

}

}

##### Assume that we are not aware of the dependency injection. Then, how will we expose the GET method from API? We used to create an instance of CategoryRepository and call the GetCategories method using that instance. So tomorrow, if there is a change in CategoryRepository, it will directly affect the GET method of API as it is tightly coupled with that.

[HttpGet]

public async Task<IActionResult> Get()

{

CategoryRepository categoryRepository = new CategoryRepository();

List<Category> categories = categoryRepository.GetCategories();

return Ok(categories);

}

##### With the .NET Framework, we used to use containers like LightInject, NInject, Unity, etc. But in .NET Core, Microsoft has provided an in-built container. We need to add the namespace, i.e., Microsoft.Extension.DependencyInjection.

##### So, in the startup class, inside the ConfigureServices method, we need to add our dependency into the service collection, which will dynamically inject whenever we want in the project. Also, we can mention which kind of instance we want to inject - the lifetime of our instance.

##### AddTransient: Creates a new instance of the service every time it's requested. This is best for lightweight, stateless services.

##### AddScoped: Create a same instance of the service within a request, but different across different requests.

##### AddSingleton: Creates a single instance of the service for the entire application, and then reuses that instance for each request.

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

namespace DIinCore

{

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

// Registering a singleton service

services.AddSingleton<ICategoryRepository, CategoryRepository>();

// Other dependency injections can be added here using AddTransient or AddScoped

// services.AddTransient<ICategoryRepository, CategoryRepository>();

// services.AddScoped<ICategoryRepository, CategoryRepository>();

services.AddMvc();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseMvc();

}

}

}

##### So far, we have added our dependency to the collection. It’s time to inject where we need it, i.e., in the Web API. Our GET method is inside the CategoryController, and we want an instance of categoryrepository. So, let’s create a CategoryController constructor which expects the ICategoryRepository type. From this parameterized constructor, set the private property of type ICategoryRepository, which will be used to call GetCategories from the GET method.

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

using System.Threading.Tasks;

namespace DIinCore.Controllers

{

[Route("api/Category")]

public class CategoryController : Controller

{

private ICategoryRepository categoryRepository { get; set; }

public CategoryController(ICategoryRepository categoryRepository)

{

this.categoryRepository = categoryRepository;

}

[HttpGet]

public async Task<IActionResult> Get()

{

List<Category> categories = categoryRepository.GetCategories();

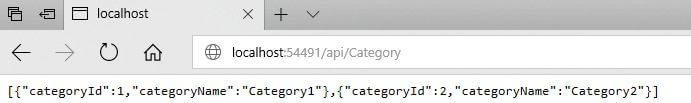
return Ok(categories);

}

}

}

Run the application, and we will be able to see the result of the GET method of CategoryController. Now, even though we haven’t created an instance of CategoryRepository, which is expected by CategoryController, we are able to call the GET method successfully. The instance of CategoryRepository has been resolved dynamically, i.e., our Dependency Injection.

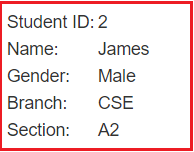


##### ****What is MVC?****

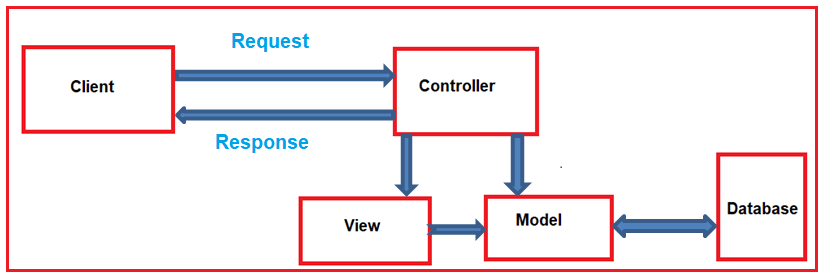
MVC stands for Model View and Controller. It is an architectural design pattern that means this design pattern is used at the architecture level of an application. So, the point that you need to remember is MVC is not a programming language, MVC is not a Framework, it is a design pattern. When we design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

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

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



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



In the MVC design pattern, the controller component, who actually handles the incoming request. In order to handle the request, the controller components do several things are as follows. The controller component creates the model that is require   
ed by a view. The model is the component in the MVC design pattern which basically contains classes :?Li;.lij8lo9l’[;pl

\|/that are used to store the domain data or you can say business data.

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

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

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

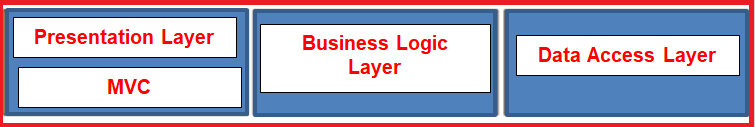
**Where MVC is used in the real-time three-layer application?**

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

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

The MVC design pattern is basically used to implement the Presentation Layer of the

plication. Please have a look at the following diagram.



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

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

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

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

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

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



**What is Program Class?**

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

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

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

var app = builder.Build();

app.UseRouting();

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

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

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

app.Run();

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

**What is Kestrel Web Server?**

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

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

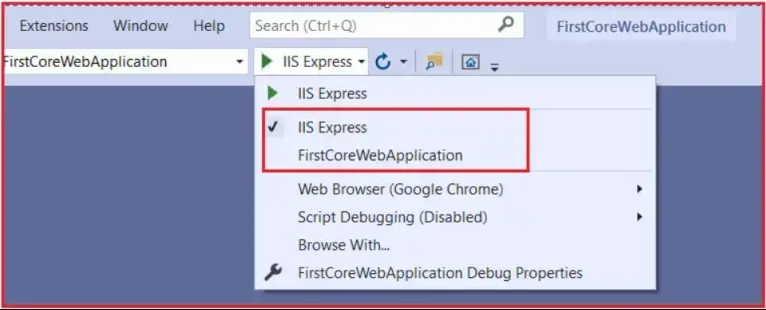
Kestrel is a lightweight, cross-platform web server built specifically for ASP.NET Core applications. It’s designed to be a fast, scalable, and efficient web server that can handle incoming HTTP requests and serve content to clients. Kestrel is the default web server that comes with ASP.NET Core, and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

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

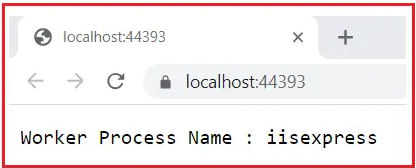
Before using the Kestrel Web Server to run our application, open the [**launchSettings.json**](https://dotnettutorials.net/lesson/asp-net-core-launchsettings-json-file/) file from your application’s Properties folder. Once you open the launchSettings.json file, you will find the following code by default.



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



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



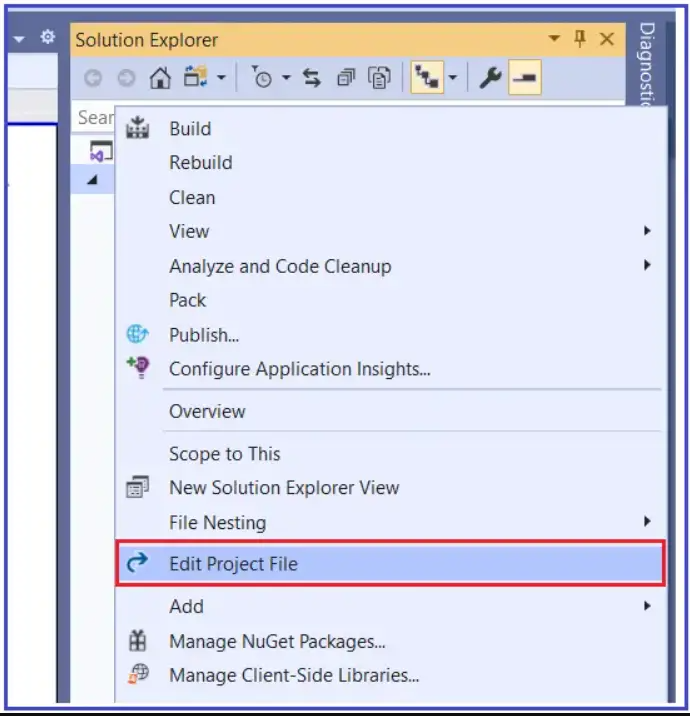
**What is the Hosting Model in ASP.NET Core?**

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

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

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

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



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

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

<PropertyGroup>

<TargetFramework>net6.0</TargetFramework>

<Nullable>enable</Nullable>

<ImplicitUsings>enable</ImplicitUsings>

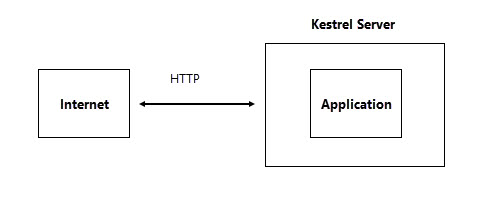
<AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>

</PropertyGroup>

< Project>

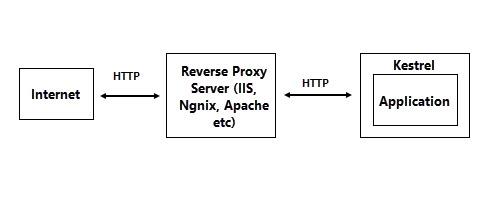
**Internal Web Server**

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



**External Web Server**

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



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

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

##### ****What is the ASP.NET Core AppSettings.json File?****

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.

**AppSettings.json**

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

"ConnectionStrings": {

"DefaultConnection": "Server=LAPTOP-LSTNN0NC\\SQLEXPRESS; Database=DotNetPools; Trusted\_Connection=True; MultipleActiveResultSets=true"

},

"WebAPIBaseUrl": "http://localhost:1012/api"

}

Access The Value From **AppSettings.json file**

public class ViewUserController : Controller

{

private readonly IConfiguration \_configuration;

public ViewUserController(IConfiguration configuration)

{

\_configuration = configuration;

}

string apiBaseUrl = \_configuration.GetValue<string>("WebAPIBaseUrl");

}

**Middleware**

Middleware in ASP.NET Core is a software component responsible for processing requests and generating responses in the web application pipeline. It sits between the server side and the application and is designed to handle cross-cutting concerns, such as authentication, caching, logging, and routing.

The primary purpose of middleware is to provide a modular way of processing HTTP requests and responses, allowing developers to add, remove, or reorder middleware components in the pipeline based on their specific needs. This makes it easy to customize the web application's behavior without modifying the core application logic.

In addition, middleware can perform various tasks, such as modifying request or response headers, handling errors and exceptions, and executing asynchronous code. Middleware can also perform custom processing of requests and responses, such as generating dynamic content or formatting data.

Overall, middleware plays a critical role in the architecture of ASP.NET Core applications, allowing developers to write modular, flexible, and extensible web applications that can be easily customized and scaled.

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

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

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

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

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

## **Default Page in ASP.NET Core**

## We want the **index.html** page to be our default page

Step 1 : Add one HTML Page with the name **index.html**within the wwwroot folder.

Step 2 : //Setting the Default Files in program.cs file

app.UseDefaultFiles();

Now, we want the **MyCustomPage1.html** page to be our default page instead of the **index.html** page. To do this, you need to modify the Main() method of the Program class as follows.

Step 1 : Add one HTML Page with the name **CustomPage1.html**within the wwwroot folder.

Step 2 :

//Specify the MyCustomPage1.html as the default page

//First Create an Instance of DefaultFilesOptions

DefaultFilesOptions defaultFilesOptions = new DefaultFilesOptions();

//Clear any DefaultFileNames if already there

defaultFilesOptions.DefaultFileNames.Clear();

//Add the default HTML Page to the DefaultFilesOptions Instance

defaultFilesOptions.DefaultFileNames.Add("MyCustomPage1.html");

app.UseDefaultFiles(defaultFilesOptions);

## **.NET Background Tasks**

Background tasks are the elements of an application that work behind the scenes without input or interaction from the end user. They usually involve processes that run for long periods or require continuous updates to the host application’s parameters.

As background tasks can run concurrently with foreground operations, developers typically assign them to run do so to improve performance. One common example is to periodically clean up unused data after a specified duration, similar to clearing a web browser’s cache to improve loading speed.

**IHostedService**

One of the easiest ways to start implementing background tasks into your ASP.NET application is through the IHostedService interface. This interface allows you to run background tasks at specific intervals continuously, which you can designate with every application instance or set up as a standalone project. However, IHostedService is typically relegated to short-running tasks.

The IHostedService interface in ASP.NET Core defines a background task or service as part of the application's lifetime. It’s typically used for monitoring, logging, or data processing tasks that must run continuously, even when the application is not processing requests. Classes that implement the IHostedService interface are added to the application's service collection using dependency injection, and they are started and stopped automatically by the application's host.

The IHostedService interface defines two methods: StartAsync and StopAsync. The StartAsync method is called when the application starts and is used to start the background task or service. The StopAsync method is called when the application is stopped or restarted. It’s used to stop the background task or service, releasing acquired resources.

public class SampleBackgroundTaskService : IHostedService

{

private Timer \_timer = null;

public Task StartAsync(CancellationToken cancellingToken)

{

\_timer = new Timer(ActionToBePerformed, null, TimeSpan.Zero,

TimeSpan.FromSeconds(5));

return Task.CompletedTask;

}

void ActionToBePerformed(object state)

{

Debug.WriteLine("Working behind the scenes...");

}

public Task StopAsync(CancellationToken cancellingToken)

{

\_timer?.Change(Timeout.Infinite, 0);

return Task.CompletedTask;

}

}

After creating your background service class, you have to add the SampleBackgroundTaskService to the ConfigureServices method in your project’s startup.cs file. This code will enable the application to recognize and call the background service task.

using Microsoft.AspNetCore.Mvc;

public void ConfigureServices(IServiceCollection services)

{

services.AddHostedService<SampleBackgroundTaskService>();

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

}

**BackgroundService**

The BackgroundService class is an abstract base class that simplifies the implementation of background tasks. It provides a single method to override. BackgroundService was introduced as an extension for long-running or concurrent tasks.

To rewrite the same function from the SampleBackgroundTaskService above using BackgroundService, set up the class as shown in the following code:

using System.Diagnostics;

public class SampleBackgroundTaskService : BackgroundService

{

protected async override Task ExecuteAsync(CancellationToken cancellingToken)

{

while (!cancellingToken.IsCancellationRequested)

{

Debug.WriteLine("Working behind the scenes...");

await Task.Delay(5000, cancellingToken);

}

}

}

**Create the App using .NET CLI**

**dotnet new webapp --output aspnetcoreapp --no-https**

The preceding command creates a new web app project in a directory named aspnetcoreapp. The project doesn't use HTTPS.

**Run the app using .NET CLI**

**cd aspnetcoreapp**

**dotnet run**

The run command produces output like the following example:

**Output**

Building...

info: Microsoft.Hosting.Lifetime[14]

Now listening on: http://localhost:5109

info: Microsoft.Hosting.Lifetime[0]

Application started. Press Ctrl+C to shut down.

info: Microsoft.Hosting.Lifetime[0]

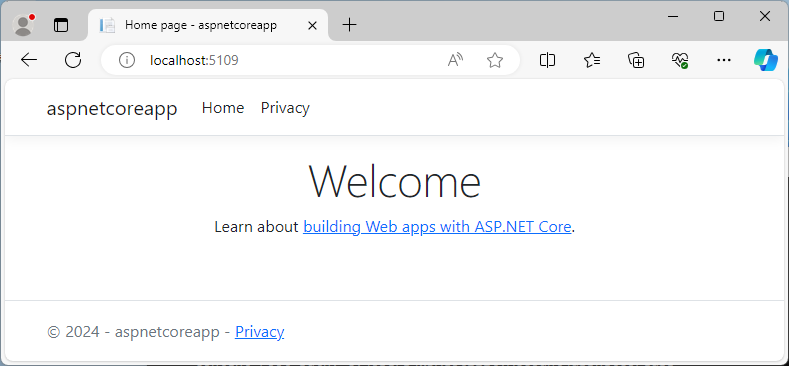
Hosting environment: Development

info: Microsoft.Hosting.Lifetime[0]

Content root path: C:\aspnetcoreapp

Open a browser and go to the URL shown in the output. In this example, the URL is http://localhost:5109.

T he browser shows the home page.



**How does ASP.NET Core handle concurrency and parallelism?**

ASP.NET Core provides several mechanisms for handling concurrency and parallelism depending on the application's specific requirements. Some common mechanisms used in ASP.NET Core applications are:

**Asynchronous programming:** ASP.NET Core supports asynchronous programming by using the async and awaits keywords. Asynchronous programming allows multiple tasks to be executed concurrently without blocking the main thread, improving the application's responsiveness.

**Parallel programming:** ASP.NET Core supports parallel programming using the Parallel class and the Task Parallel Library (TPL). Parallel programming allows multiple tasks to be executed concurrently across multiple processors, improving the application's performance.

**Locking and synchronization:** ASP.NET Core provides several mechanisms for locking and synchronization, including the lock keyword, the Interlocked class, and the Monitor class. These mechanisms allow multiple threads to access shared resources safely and prevent race conditions.

**Concurrency control:** ASP.NET Core supports concurrency control through transactional memory and the optimistic concurrency control (OCC) pattern. Concurrency control ensures that multiple threads can access and modify shared resources without interfering with each other.

Using these mechanisms, developers can build ASP.NET Core applications that are more responsive, scalable, and efficient, handling multiple requests and tasks concurrently and in parallel. However, using these mechanisms carefully and appropriately is important, as concurrency and parallelism can introduce new challenges, such as race conditions, deadlocks, and thread starvation.

**How do you implement caching in ASP.NET Core?**

Response caching in ASP.NET Core is a technique used to improve the performance and scalability of web applications by caching the ASP.NET Core MVC responses returned by the server for a specific period. Caching the response can help reduce the number of requests made to the server, as clients can reuse the cached response instead of requesting the same resource again.

Response caching works by adding a caching layer between the client and the server. When a client requests a resource, the caching layer checks whether the response for the request has been cached. If the response is cached, the caching layer returns the cached response to the client. If the response is not cached, the request is forwarded to the server, and the server generates the response and caches it for future use.

In ASP.NET Core, response caching can be implemented using the [ResponseCache] attribute, which can be applied to an action method in a controller. The attribute allows developers to specify the caching behavior, such as the duration of the cache, the location of the cache, and the cache key. By default, the caching location is on the client side, but it can also be set to a distributed or proxy cache depending on the needs of the application.

Response caching can significantly impact the performance and scalability of web applications, particularly for resources that are expensive to generate, such as database queries or API calls. However, it’s important to use response caching judiciously, as caching can also lead to stale data being returned to clients. Therefore, setting appropriate caching policies and ensuring the cache is invalidated when the underlying data changes are crucial.

**What’s the difference between middleware and a filter in ASP.NET Core?**

In ASP.NET Core, middleware and filters are two mechanisms used for processing requests and responses.

Middleware is a software component between the web server (like Apache) and the application and processes requests and responses during the application development. Middleware can be used for various tasks, such as authentication, logging, and error handling. Middleware is executed in a pipeline, and each middleware component can modify the request or response before passing it to the next component in the pipeline.

Conversely, filters are used to perform cross-cutting concerns on controllers and actions in an MVC application. Filters can be used for authorization, validation, and caching tasks. Filters are executed before and after the action method, and they can modify the request or response or short-circuit the request processing if necessary.

The main difference between middleware and filters is their scope and the way they are executed. Middleware is executed globally and can be used for any request or response. In contrast, filters are executed only for specific controllers or actions and can be used to modify the request or response before or after the action method.

**What is Core CLR?**

CoreCLR (Common Language Runtime, now renamed to .NET Runtime) is the runtime environment executing ASP.NET Core applications. It is the open-source implementation of the .NET runtime, developed by Microsoft and available on multiple platforms, including Windows, Linux, and macOS.

CoreCLR provides a managed execution environment for ASP.NET Core applications, including memory management, garbage collection, type safety, and security. It also supports just-in-time (JIT) compilation, which compiles code at runtime to native machine code, allowing for faster execution.

CoreCLR is designed to be modular, with various components such as the garbage collector, JIT compiler, and primitive data type system implemented as separate modules. This modularity allows for more flexibility and customization in building and deploying .NET Core applications.

CoreCLR is a critical component of the .NET platform, providing the necessary runtime infrastructure for developing and executing .NET applications across different platforms.

**Have you worked with Docker on ASP.NET Core projects?**

The Docker platform allows developers to package and deploy applications in lightweight, portable containers. In the context of ASP.NET Core, Docker provides a way to package and deploy ASP.NET Core applications and their dependencies in a self-contained, isolated container that can run on any platform that supports Docker.

Using Docker in ASP.NET Core, developers can create Docker images of their applications, which can be deployed to any environment that supports Docker. This makes it easy to deploy ASP.NET Core applications consistently and reliably, without worrying about differences in the underlying infrastructure.

Overall, Docker is a powerful tool for developing, deploying, and managing ASP.NET Core applications, providing a portable, flexible, and scalable environment for building modern applications.

**Explain the difference between app.Run and app.Use in ASP.NET Core.**

**app.Use** method adds a middleware delegate to the application's request pipeline. When you want to pass the context to the next middleware then prefer app.Use method.

**app.Run** method adds a terminal middleware delegate to the application's request pipeline. When you want to terminate the pipeline then prefer to use the app.Run method.