**WEEK 4**

**DAY 1 SESSION PLAN**

**Mentoring Session (6 hours):**

1. Introduction to .NET Core (2 hours)

○ What is .NET Core

○ Benefits of .NET Core

○ What is new in .NET Core

○ .NET Core vs .NET Framework

○ First .NET Core Application

○ Building .NET Core Applications

2. **Middleware and Static Files (2 hours)**

○ Understanding middleware in ASP.NET Core

○ Configuring and using middleware components

○ Serving static files (HTML, CSS, JavaScript)

○ Security considerations with static files

3. Introduction to Razor Pages (2 hours)

○ Overview of Razor Pages architecture

○ Advantages of Razor Pages over traditional MVC

○ Creating and configuring Razor Pages in a project

○ Folder structure and naming conventions

**Introduction to ASP.NET Core Framework**

In this article, I will give you a brief introduction to **ASP.NET Core Framework**. Nowadays, when it comes to software development, everyone is talking about **Free, Open-Source**, **and Cross-Platform** Development. As we all know, Microsoft is well known for its Windows-based products. Now, we are in the new age of software development. For this, Microsoft came into the market with a revolutionary product called ASP.NET Core.

**History of ASP.NET**

As we know, ASP.NET is the framework used to develop Data-Driven Web Applications for many years. Since then, the ASP.NET Framework has gone through a steady evolutionary change, and finally, the most decent evolution is ASP.NET Core (you can also call it .NET).

1. The ASP.NET Core (.NET) is not a continuous part of the legacy ASP.NET Framework. Instead, it is a completely new framework.
2. This Framework is an actual rewrite of the legacy ASP.NET Framework but with a much smaller and a lot more modular.
3. Some people think many things remain the same, but that is not completely true. The ASP.NET Core is actually a big fundamental change to the legacy ASP.NET Framework.

**What is ASP.NET Core?**

According to [**Microsoft**](https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet-core), ASP.NET Core is a Cross-Platform, Open-Source, High-Performance framework for building modern, cloud-based, and internet-connected applications that run on Windows, Linux, macOS, and Docker. It is a complete redesign of ASP.NET, with architectural changes that result in a smaller, more modular framework.

**Why ASP.NET Core?**

Nowadays, the ASP.NET Core framework is becoming more and more popular among developers. There are a number of reasons why modern developers are using it, and some of them are listed below:

**Cross-Platform:**

The ASP.NET Core Framework is designed from scratch to be Cross-Platform for both development and deployment. So, we don’t need to build different applications for different platforms using different frameworks. Let us discuss what Cross-Platform is from the ASP.NET Core point of view by comparing it with the earlier versions of the ASP.NET Framework.

The earlier versions of ASP.NET Framework applications can only run on Windows platforms. On the other hand, the ASP.NET Core applications can be developed and run on different platforms such as Windows, Mac, or Linux operating systems. We can host the earlier ASP.NET Framework applications only on IIS, whereas we can host the ASP.NET Core applications on IIS, Nginx, Docker, Apache, etc.

**Open Source:**

The ASP.NET Core framework is open source, which is the main reason for its popularity. The entire source code for ASP.NET Core Framework is available at https://github.com/dotnet/aspnetcore, and you can download the source code. Even if you want, you can also modify and compile your own version.

.NET is open source on GitHub and has over 100,000 contributions, including 3,700 companies. You can also contribute or download the source code from the ASP.NET Core Repository on GitHub.

The ASP.NET Core team is always there to support your effort in developing the application. It receives bug fixing and improvement updates on a regular basis, usually within a short time period. You don’t have to wait longer for updates.

**High Performance:**

ASP.NET Core is optimized for high performance. Its modular components and the ability to include only the necessary dependencies make it highly optimized for speed and scalability. Benchmarks demonstrate that applications developed with ASP.NET Core are significantly faster than those built with previous versions of ASP.NET.

**Unified MVC and Web API Framework:**

ASP.NET Core unifies the ASP.NET MVC and ASP.NET Web API into a single programming model, simplifying the development process for web applications and APIs. That means a single controller class can handle both.

**Lightweight and Modular:**

The framework is built to be modular, which means you can include only the necessary components in your application, reducing its overall footprint and enhancing performance.

**Built-in Dependency Injection:**

The framework includes a built-in dependency injection (DI) container, which makes it easy to manage service lifetimes and dependencies.

**Cloud-Ready:**

ASP.NET Core is designed to be cloud-ready, with features and configurations that facilitate deployment and scaling in cloud environments like Azure.

**Razor Pages:**

Razor Pages is a new feature in ASP.NET Core that makes it easier to build page-focused web applications. It is built on top of the existing ASP.NET Core MVC framework.

**Tag Helpers:**

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files, enhancing the productivity of developers working on the view layer.

**Middleware:**

ASP.NET Core uses middleware to handle requests and responses. Middleware components can be composed in a pipeline to process requests and responses, providing great flexibility and control over the application’s behavior.

**Side-by-Side Versioning:**

ASP.NET Core supports running multiple versions of the framework side-by-side on the same machine, which allows for seamless upgrades and backward compatibility.

**CLI Support:**

Using CLI (Command Line Interface) commands, you can develop, build, and publish .NET applications. Our upcoming articles discuss developing, building, and running .NET Core Applications using CLI.

**Testing:**

Applications developed using the ASP.NET Core framework can easily be tested and maintained. This is possible because it allows you to separate different parts of your application into different independent pieces and test them independently. Testing frameworks such as MSTtest, xUnit, and MOQ can be easily integrated into ASP.NET Core applications to simulate any test scenario.

**Support for Modern Client-side Frameworks:**

It offers support for client-side frameworks like Angular, React, and others and has built-in templates to get started with these frameworks easily.

**Robust Security Features:**

ASP.NET Core provides features to manage authentication, authorization, data protection, SSL enforcement, app secrets, CSRF protection, and more, which are essential for securing web applications and web services.

**Configuration and Environment Management:**

It has a robust configuration system can read settings from various sources, such as JSON files, environment variables, command-line arguments, etc. It also provides features to handle different environments, such as development, staging, and production, in a streamlined manner.

**Excellent Developer Tools:**

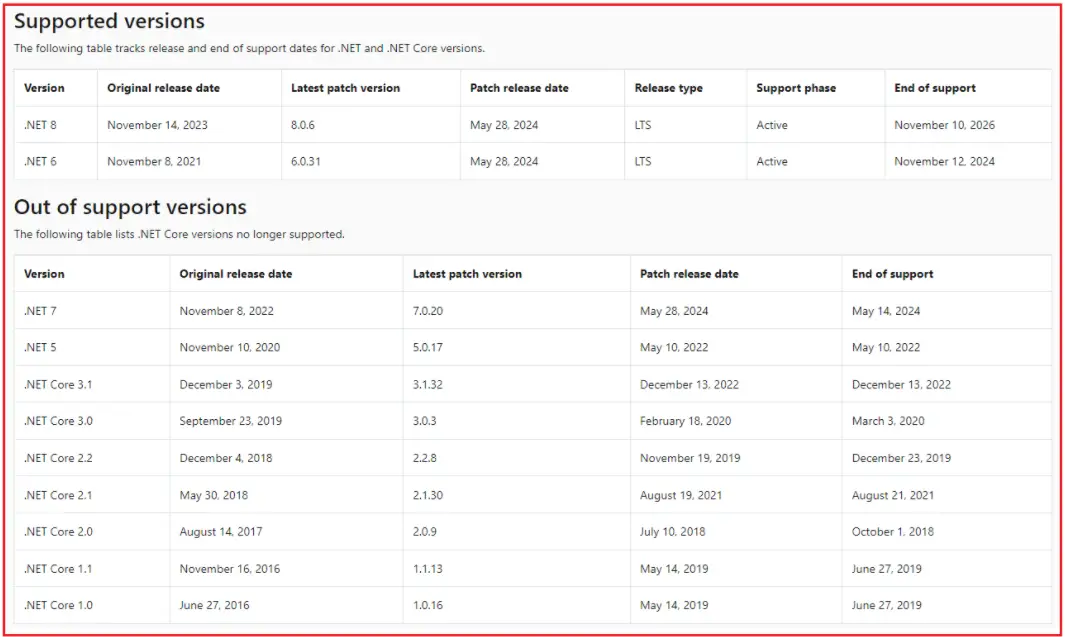
You have multiple options to develop ASP.NET Core applications. You can use either Visual Studio or Visual Studio Code or any third-party editor of your choice, such as ATOM, Sublime, etc.

ASP.NET Core also comes with a lot of preloaded tools. Anybody would love to work with these tools due to their simplicity and ease. Using Libmam (Libmam allows the installation of Bootstrap and jQuery easily), ReSharper (ReSharper provides on-the-fly code analysis and eliminates errors in C#, VB.NET, XAML, ASP.NET, and XML), etc.

**.NET Core Support Policy and Release Lifecycle:**

The latest version of .NET Core is .NET 8, released in November 2023. Beginning with .NET 5.0, Microsoft .NET core has been re-branded as .NET. So, from .NET 5, the word Core will not be used, and they will be called .NET 6, .NET 7, .NET 8, NET 9, etc.

A new major release of .NET Core is published every year in the month of November. Even-numbered releases are LTS releases with free support and patches for three years, and odd-numbered releases are STS releases with free support and patches for 18 months. For a better understanding of the .NET Core version, please look at the image below.

****

Here,

* **Long-Term Support (LTS) Releases:** These versions are supported for three years after their initial release. LTS releases are focused on stability and are intended for long-term usage in production environments.
* **Short-Term Support (STS) Releases**: These are supported for 18 months from the release date. They are also suitable for production but do not have the extended support period that LTS versions offer.

Customers can choose Long Term Support (LTS) releases or Standard Term Support (STS) releases. The quality of all releases is the same. The only difference is the length of support. LTS releases get free support and patches for 3 years. STS releases get free support and patches for 18 months. Please check the below URL for more detailed information:

[**https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core**](https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core)

**.NET Core Versions Overview**

**.NET Core 1.0**

Release Date: June 27, 2016

**Key Features:**

* Cross-Platform: For the first time, .NET applications run on Windows, macOS, and Linux.
* Modular Framework: Applications could include only the necessary libraries, reducing overhead.
* ASP.NET Core: A new, unified web framework for building modern web applications.
* Command-line tools: Provided robust command-line tools (CLI) for all supported platforms, which facilitated various development tasks outside of Visual Studio.
* Entity Framework Core: A lightweight, extensible, cross-platform version of Entity Framework for data access.

**Support Status:** End of support

**.NET Core 1.1**

Release Date: November 16, 2016

**Key Features:**

* Performance Improvements: Enhanced runtime performance and efficiency.
* Additional APIs: Expanded the set of APIs available to developers.
* Tooling Improvements: Better tools for development, including command-line tools.

**Support Status:** End of support

**.NET Core 2.0**

Release Date: August 14, 2017

**Key Features:**

* ASP.NET Core Razor Pages: Simplified the creation of page-centric web applications.
* Enhanced Compatibility: Improved compatibility with existing .NET Framework libraries.

**Support Status:** End of support

**.NET Core 2.1 (LTS)**

Release Date: May 30, 2018

**Key Features:**

* Long-Term Support (LTS): First LTS version, ensuring long-term stability and support.
* SignalR: Real-time web functionality for adding real-time features to web apps.
* HTTP Client Performance: Major improvements in HTTP client performance.

**Support Status:** End of support on August 21, 2021

**.NET Core 2.2**

Release Date: December 4, 2018

**Key Features:**

* Diagnostic Improvements: Enhanced diagnostic tools for monitoring and troubleshooting.
* Health Checks: Built-in health checks for ASP.NET Core apps.
* Azure SignalR Service: Enhanced integration with Azure services.

**Support Status:** End of support on December 23, 2019

**.NET Core 3.0**

Release Date: September 23, 2019

**Key Features:**

* Windows Desktop Support: Added support for Windows Forms and WPF, enabling desktop application development.
* C# 8.0 Support: New language features like nullable reference types and async streams.
* Blazor: Introduced Blazor, enabling C# to run in the browser using WebAssembly.
* .NET Core CLI Improvements: Enhanced command-line interface for better development experience.

**Support Status:** End of support on March 3, 2020

**.NET Core 3.1 (LTS)**

Release Date: December 3, 2019

**Key Features:**

* Long-Term Support (LTS): Ensured stability and long-term support.
* Razor Components: Enhanced Razor Components for building interactive web UIs.
* Performance Enhancements: Continued improvements in performance and reliability.

**Support Status:** Supported until December 2022

**.NET 5.0**

Release Date: November 10, 2020

**Key Features:**

* Unified Platform: Unified the .NET ecosystem, combining .NET Core and .NET Framework.
* Performance Improvements: Continued focus on high performance and scalability.
* C# 9.0 Support: Introduced new language features like records and improved pattern matching.

**Support Status:** Supported until May 2022

**.NET 6.0 (LTS)**

Release Date: November 8, 2021

**Key Features:**

* Long-Term Support (LTS): Guaranteed support and stability for long-term projects.
* .NET MAUI: Introduced .NET Multi-platform App UI for building cross-platform applications.
* Minimal APIs: Simplified API creation with a more concise syntax.
* Hot Reload: Allowed live code updates without restarting the application.

**Support Status:** Supported until November 2024

**.NET 7.0**

Release Date: November 8, 2022

**Key Features:**

* Enhanced .NET MAUI: Improved tools for building cross-platform applications.
* Performance Enhancements: Further optimizations in runtime performance.
* Updated Language Features: Support for C# 10 and F# 6 with new enhancements.
* Containerization: Better support for building and deploying containerized applications.

**Support Status:** Supported until May 2024

**.NET 8.0 (Planned LTS)**

Expected Release Date: November 2023

**Key Features:**

* Continued Performance Improvements: Ongoing enhancements in runtime and API performance.
* New Language Features: C# 11 with new features to improve developer productivity.
* Cloud-Native Enhancements: Improved support for developing and deploying cloud-native applications.
* .NET MAUI and Blazor Enhancements: Continued improvements to these frameworks for building modern web and mobile applications.

**Support Status:** Supported until November 2026

**What the ASP.NET Core Doesn’t Have?**

If you come from an ASP.NET background, you will not find the following things in ASP.NET Core.

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

**Differences Between .NET Framework and .NET Core Framework**

**.NET Framework:**

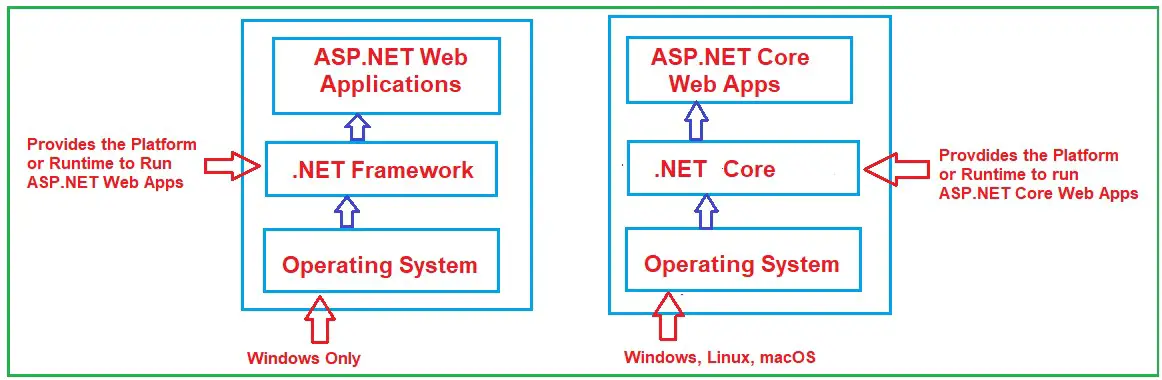
* .NET Framework is designed to run only on Windows operating systems. It is tightly integrated with Windows, which makes it optimized for desktop and server-based applications on this OS.
* .NET Framework comes as a large, monolithic package that includes all features, which can lead to unnecessary resource consumption for applications that do not need all its capabilities.
* .NET Framework does not match the performance improvements in .NET Core due to its older architecture and design constraints.
* .NET Framework is in maintenance mode, with Microsoft primarily releasing security updates and critical fixes. No new features are being added to the .NET Framework.
* .NET Framework is developed and maintained by Microsoft.
* .NET Framework is best for applications that are tightly integrated with the Windows ecosystem, such as desktop applications, or for maintaining large existing applications that were built with it.

**.NET Core:**

* .NET Core is a cross-platform framework that can run on Windows, Linux, and macOS. This makes it suitable for applications that need to operate across multiple operating systems.
* .NET Core is modular, allowing developers to include only the necessary packages via NuGet. This results in lighter applications that are optimized for performance and resource usage.
* .NET Core is optimized for high performance and scalability. It includes optimizations such as a smaller memory footprint and faster startup times, making it suitable for microservices and cloud applications.
* .NET Core is in development mode, with new features, performance improvements, and enhancements being added regularly.
* .NET Core is open source on GitHub and has over 100,000 contributions, and 3,700 companies have already contributed.
* .NET Core is ideal for building modern web applications, microservices, and applications that require cross-platform functionality.

**Understanding .NET Core (.NET) vs ASP.NET Core:**

Many people are confused between ASP.NET Core and .NET Core. Please note that ASP.NET Core and .NET Core are not the same. They are different, just like ASP.NET and .NET Framework are different. ASP.NET Core is a fast, lightweight, modular, and open-source framework for creating Web Applications, Mobile Applications, and Web Services that run on Windows, Linux, and macOS. .NET Core is the Runtime Environment on which ASP.NET Core applications run. For a better understanding, please have a look at the following diagram:



**.NET Core (.NET):**

1. .NET Core (.NET) is the cross-platform software. It provides the Runtime Environment where the ASP.NET Core Web Applications will run.
2. Install .NET Core Runtime to run applications and install .NET Core SDK to build applications.
3. .NET 8 is the latest stable version.

**ASP.NET Core:**

1. ASP.NET Core is an Open-Source, Cross-Platform Framework using which we can develop different types of Web Applications, such as MVC, Web API, Razor Pages, etc.
2. If you want to develop and run .NET Core Applications, you need to install the .NET Core SDK, including the .NET Runtime. If you only want to run .NET Core Applications, you need only install the .NET Core Runtime.
3. ASP.NET Core 8 is the latest stable version.

**Note:** There is no separate versioning for ASP.NET Core; it is the same as the other .NET Core versions.

**.NET Core GitHub Repository:**

**Runtime: https://github.com/dotnet/runtime**  
**SDK: https://github.com/dotnet/sdk**  
**GitHub Repository: https://github.com/dotnet/aspnetcore**

In the next article, I will discuss the [**ASP.NET Core Environment Setup**](https://dotnettutorials.net/lesson/asp-net-core-environment-setup/) Required for Developing ASP.NET Core Web Applications using Visual Studio. In this article, I will try to give a brief introduction to the **ASP.NET Core Framework**. I hope this Introduction to ASP.NET Core Framework article will help you with your needs.

**ASP.NET Core Middleware with Examples**

In this article, I will discuss the **ASP.NET Core Middleware Components with Examples**. Please read our previous article discussing the [**ASP.NET Core AppSettings.json File**](https://dotnettutorials.net/lesson/asp-net-core-appsettings-json-file/) with Examples. In this article, we will discuss the following concepts related to the ASP.NET Core Middleware Components**.**

1. **What are ASP.NET Core Middleware Components?**
2. **Where Do We Use Middleware Components in the ASP.NET Core Application?**
3. **How Do We Configure Middleware Components in .NET Applications?**
4. **How Does the Middleware Component Works in the ASP.NET Core Application?**
5. **What is the Execution Order of Middleware Components in ASP.NET Core?**
6. **What are Request Delegates in ASP.NET Core?**
7. **What is the use of the Use and Run Extension Method in ASP.NET Core Web Application?**
8. **What is the Difference Between MapGet and Map Extension Methods in ASP.NET Core?**
9. **How Do We Configure Middleware Components Using the Run() Extension Method?**
10. **Adding Multiple Middleware Components into the Request Processing Pipeline**
11. **Configuring Middleware Components Using the Use Extension Method**
12. **Can we create a Terminal Middleware Component using the Extension method?**
13. **Differences Between Run, Map, and Use Extension Method in ASP.NET Core**

**What are ASP.NET Core Middleware Components?**

ASP.NET Core Middleware Components are the basic building blocks of the Request Processing Pipeline in ASP.NET Core Applications. The pipeline determines how the HTTP Requests and Responses will be processed in the ASP.NET Core Application. Middleware components are used to implement various functionalities like authentication, error handling, routing, and logging.

Middleware components are executed in the order they are added to the pipeline, and each middleware component in the ASP.NET Core Application performs the following tasks.

* Chooses whether to pass the HTTP Request to the next Middleware component registered in the request processing pipeline. This can be achieved by calling the next() method within the Middleware. In this article, we will discuss how to register the Middleware component and call the next() method.
* Can perform certain tasks before and after the next component is invoked in the pipeline.

**Note:** In ASP.NET Core, many built-in Middleware components are already available for us to use directly. We can also create our own middleware components per our business requirements. The most important point you need to remember is that a given Middleware component should only have a specific purpose, i.e., a single responsibility.

**Where Do We Use Middleware Components in the ASP.NET Core Application?**

Middleware components are used in the request processing pipeline of an ASP.NET Core application. They handle cross-cutting concerns such as authentication, logging, error handling, routing, and more. Some examples of using Middleware components in the ASP.NET Core application are as follows.

Advertisements

* **UseAuthentication** adds the authentication middleware to the ASP.NET Core pipeline. This middleware is responsible for validating the credentials provided in requests and setting the user’s context.
* **UseHttpsRedirection** Middleware automatically redirects HTTP requests to HTTPS. This is important for ensuring that all communications between the client and the server are encrypted and secure.
* **UseDeveloperExceptionPage** Middleware component provides a detailed error page in the browser during development that shows stack traces, query parameters, cookies, and headers when an exception occurs for easier debugging.
* **UseExceptionHandler**Middleware component captures and handles exceptions that occur during the request processing, allowing us to provide custom error-handling logic for production environments**.** It can help manage errors in production environments by redirecting to error pages or returning standard error responses.
* **UseStaticFiles** Middleware enables our application to serve static files, like images, JavaScript, and CSS files. It’s essential for delivering the static content of our web applications directly to clients.
* **UseAuthorization** adds authorization middleware to the ASP.NET Core pipeline. It checks whether the authenticated user has permission to access a given resource or endpoint, which is necessary for enforcing security policies in our application.
* **UseRouting** configures the routing middleware, enabling endpoint routing to map incoming requests to the appropriate endpoint handlers based on route patterns defined in the application.



**How Do We Configure Middleware Components in .NET Applications?**

From the .NET 6, we need to configure the **Middleware Components** within the **Main()** method of the Program class, which is present inside the **Program.cs** class file. As we already discussed, the application execution will start with the Main method in the ASP.NET Core Web Application. When we create a new ASP.NET Core Empty Web Application, the Program class is created with the Main method by default, as shown in the image below.



In the above Main method, we have configured two middleware components, MapGet and Run, by default. If you want to configure a middleware component, you need to do so within the **Main()** method of the **Program** class using the **WebApplication** instance.

For a better understanding, please modify the Main Method of the Program class as follows. Here, we have configured a few Middleware Components using the WebApplication instance (using the variable app), such as **UseDeveloperExceptionPage()**, **UseRouting()**, and **MapGet().**

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

// Initializes the configuration for the web application

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

// Builds the web application based on the configured settings

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

// Middleware configuration section:

// Activates the Developer Exception Page in Development environment to show detailed error messages

**if** **(**app.Environment.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

// Adds the routing middleware to the request processing pipeline which is required to use the routing capabilities

app.UseRouting**()**;

// Endpoint configuration section:

// Maps HTTP GET requests to the root URL "/" to a method returning "Hello World!"

app.MapGet**(**"/", **()** =**>** "Hello World!"**)**;

// Maps HTTP GET requests to "/greet" URL to a method returning a greeting message

app.MapGet**(**"/greet", **()** =**>** "Hello from the /greet endpoint!"**)**;

// Maps HTTP GET requests to "/greet/{name}" URL to a method that uses a route parameter

app.MapGet**(**"/greet/{name}", **(string** name**)** =**>** $"Hello, {name}!"**)**;

// Starts the web application which begins listening for incoming requests

app.Run**()**;

**}**

**}**

**}**

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

// Initializes the configuration for the web application

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

// Builds the web application based on the configured settings

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

// Middleware configuration section:

// Activates the Developer Exception Page in Development environment to show detailed error messages

**if** **(**app.Environment.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

// Adds the routing middleware to the request processing pipeline which is required to use the routing capabilities

app.UseRouting**()**;

// Endpoint configuration section:

// Maps HTTP GET requests to the root URL "/" to a method returning "Hello World!"

app.MapGet**(**"/", **()** =**>** "Hello World!"**)**;

// Maps HTTP GET requests to "/greet" URL to a method returning a greeting message

app.MapGet**(**"/greet", **()** =**>** "Hello from the /greet endpoint!"**)**;

// Maps HTTP GET requests to "/greet/{name}" URL to a method that uses a route parameter

app.MapGet**(**"/greet/{name}", **(string** name**)** =**>** $"Hello, {name}!"**)**;

// Starts the web application which begins listening for incoming requests

app.Run**()**;

**}**

**}**

**}**

Before understanding the above Built-in Middleware Components (**UseDeveloperExceptionPage()**, **UseRouting()**, and **MapGet()**) in ASP.NET Core Applications, let us first understand How Exactly These Middleware Components are Executed and in what order in ASP.NET Core Web Applications.

**How Does the Middleware Component Works in the ASP.NET Core Application?**

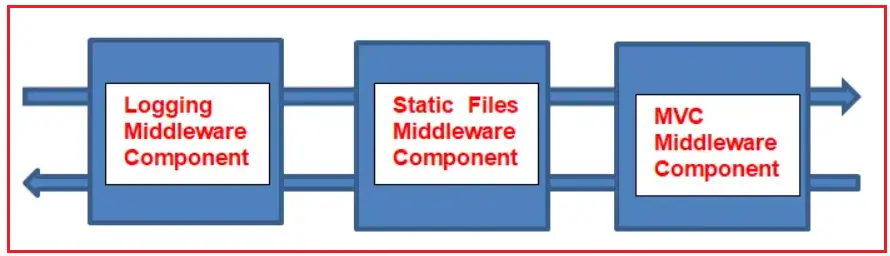
In the ASP.NET Core Web Application, the Middleware Component can access the incoming HTTP Request and outgoing HTTP Response. So, a Middleware Component in ASP.NET Core Web Application can

* Handle the incoming HTTP request by generating an HTTP response.
* Process the incoming HTTP request, modify it, and then pass it to the Next Middleware Component that is configured in the Request Processing Pipeline.
* Process the outgoing HTTP response, modify it, and then pass it on to either the previous middleware component or to the ASP.NET Core Web Server.

For better understanding, please have a look at the following diagram, which shows how the middleware components are used in the Request Processing Pipeline of the ASP.NET Core Web Application. As shown in the image below, we have configured 3 Middleware Components to the Application Request Processing Pipeline to handle HTTP Requests and Responses.

We have a Logging Middleware Component. This component logs the request time and then passes the HTTP Request to the next middleware component, i.e., the Static Files Middleware component in the Request Pipeline, for further processing.

As previously discussed, a Middleware Component in an ASP.NET Core Web Application may also handle the HTTP Request by generating an HTTP Response. The Middleware Component may also decide not to call the Next Middleware Component in the Request Processing Pipeline, a concept called **Short-Circuiting** the Request Processing Pipeline.

[](https://go.ezodn.com/ads/charity/proxy?p_id=5c4134cd-7791-4646-61b9-9eb0fa4802ab&d_id=118560&imp_id=4642671958688463&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)

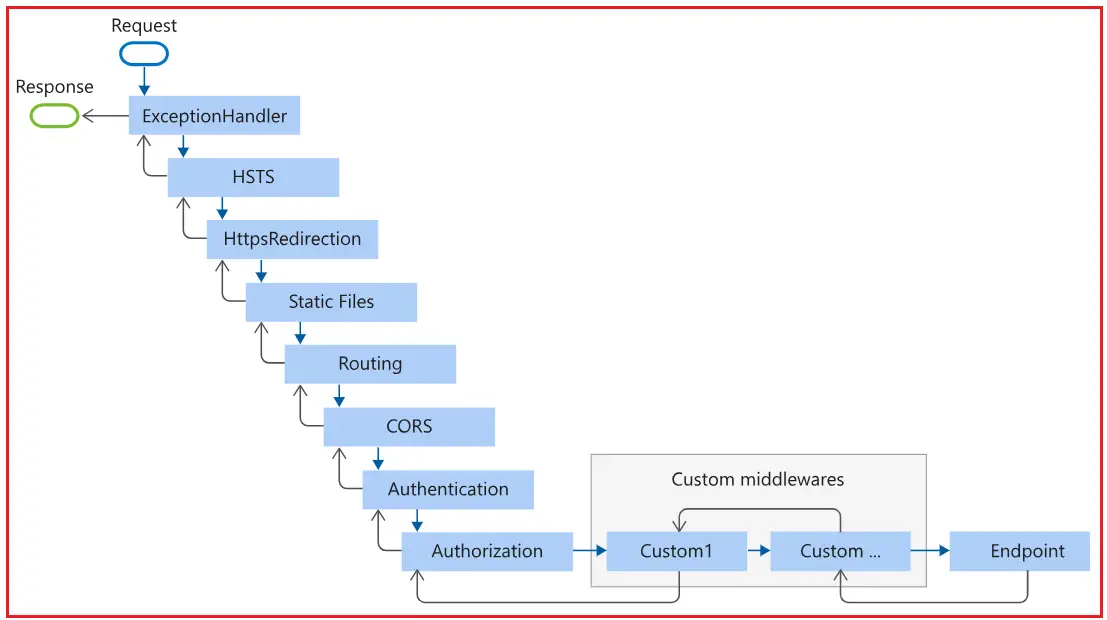
For example, we have a Static Files Middleware Component. Suppose the incoming HTTP request comes for some static files such as images, CSS files, JavaScript, etc. In that case, this Static Files Middleware component can handle the request and then Short-Circuiting the Request Processing Pipeline by not calling the Next Middleware Component in the pipeline, i.e., the MVC Middleware Component.

The ASP.NET Core Middleware Components can access the HTTP Request and Response in the pipeline. So, a middleware component can also process the outgoing response. For example, the logging middleware component, in our case, may log the time when the response is sent back to the client.

**What is the Execution Order of Middleware Components in ASP.NET Core?**

It is very important to understand the execution order of Middleware Components in ASP.NET Core Web Applications. These components are executed in the same order as they are added to the Request Processing Pipeline for incoming requests and in reverse order for outgoing responses. So, we need to take proper care when adding the Middleware Components to the Request Processing Pipeline. If we add them in the wrong order, then we might get unexpected behavior.

The following diagram shows the complete request processing pipeline for ASP.NET Core MVC and Razor Pages apps. You have full control over reordering existing middleware or injecting new custom middleware as necessary for your scenarios.



The order in which middleware components are added to the Program.cs file defines the order in which the middleware components are invoked on requests and the reverse order for the response. The order is critical for security, performance, and functionality.

**What are Request Delegates in ASP.NET Core?**

In ASP.NET Core, Request Delegates are used to build the Request Processing Pipeline, i.e., to handle each incoming HTTP request. You can configure the Request Delegates using the Run, Map, and Use Extension Methods.

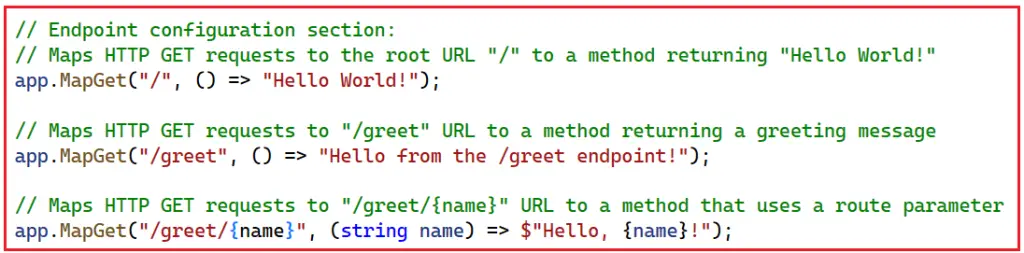
You can specify a Request Delegate using an in-line anonymous method (called in-line middleware) or specify the Request Delegates using a reusable method (named method). These reusable methods and in-line anonymous methods are called Middleware Components. Each Middleware Component in the Request Processing Pipeline is responsible for invoking the Next Middleware Component in the Pipeline or Short-Circuiting the Pipeline by not calling the Next Middleware Component.

**What is the use of the Use and Run Extension Method in ASP.NET Core Web Application?**

In the ASP.NET Core Web Application, we can use the “**Use**” and “**Run**” Extension Methods to register the Inline Middleware Component. The “**Run**” extension method allows us to add the terminating middleware (the middleware that will not call the Next Middleware Components in the Request Processing Pipeline). On the other hand, the “Use” extension method allows us to add the middleware components that are called the next middleware component in the request processing pipeline.

If you observe the Main method of the Program class, using the **WebApplication** instance (**app**) along with the extension methods such as **Use and Run**, we configure the Middleware Components.

If you further look at the Main method, then you will also see that using the MapGet extension method, we have specified multiple URL patterns like “/”,”/ greet”,”/ greet/{name} “. So, based on the URL pattern, the corresponding middleware component will be executed.



Instead of MapGet, we can also use the Map extension method, as shown below.

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

// Initializes the configuration for the web application

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

// Builds the web application based on the configured settings

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

// Middleware configuration section:

// Activates the Developer Exception Page in Development environment to show detailed error messages

**if** **(**app.Environment.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

// Adds the routing middleware to the request processing pipeline which is required to use the routing capabilities

app.UseRouting**()**;

// Endpoint configuration section:

// Maps HTTP GET requests to the root URL "/" to a method returning "Hello World!"

app.Map**(**"/", **()** =**>** "Hello World!"**)**;

// Maps HTTP GET requests to "/greet" URL to a method returning a greeting message

app.Map**(**"/greet", **()** =**>** "Hello from the /greet endpoint!"**)**;

// Maps HTTP GET requests to "/greet/{name}" URL to a method that uses a route parameter

app.Map**(**"/greet/{name}", **(string** name**)** =**>** $"Hello, {name}!"**)**;

// Starts the web application which begins listening for incoming requests

app.Run**()**;

**}**

**}**

**}**

namespace FirstCoreWebApplication

{

public class Program

{

public static void Main(string[] args)

{

// Initializes the configuration for the web application

var builder = WebApplication.CreateBuilder(args);

// Builds the web application based on the configured settings

var app = builder.Build();

// Middleware configuration section:

// Activates the Developer Exception Page in Development environment to show detailed error messages

if (app.Environment.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

// Adds the routing middleware to the request processing pipeline which is required to use the routing capabilities

app.UseRouting();

// Endpoint configuration section:

// Maps HTTP GET requests to the root URL "/" to a method returning "Hello World!"

app.Map("/", () => "Hello World!");

// Maps HTTP GET requests to "/greet" URL to a method returning a greeting message

app.Map("/greet", () => "Hello from the /greet endpoint!");

// Maps HTTP GET requests to "/greet/{name}" URL to a method that uses a route parameter

app.Map("/greet/{name}", (string name) => $"Hello, {name}!");

// Starts the web application which begins listening for incoming requests

app.Run();

}

}

}

**What is the Difference Between MapGet and Map Extension Methods in ASP.NET Core?**

In ASP.NET Core, both MapGet and Map are extension methods used for routing and mapping HTTP requests to specific middleware, but they have distinct purposes and use cases:

**MapGet Method:**

* MapGet is specifically designed to handle HTTP GET requests. It’s used to define endpoints that respond only to GET requests.
* We typically use MapGet to retrieve data from the server without modifying its state, such as fetching a list of items or getting details about a specific item.
* The MapGet method usually takes a route pattern and a request delegate (a handler function). The handler function is executed when a GET request matches the specified route pattern.

**Map Method:**

* Map is a general method for handling all types of HTTP requests (such as GET, POST, PUT, DELETE, etc.). It’s more flexible than MapGet.
* **We need to** use Map when we need to set up endpoints that might handle multiple types of HTTP requests or when we have custom logic to determine the type of request to be handled.
* Similar to MapGet, Map also takes a route pattern and a request delegate. However, since Map can handle various HTTP methods, the request delegate often contains logic to differentiate between these methods.

**Note:** Once we progress in this course, we will discuss handling GET, POST, PUT, and DELETE requests using the Map method.

**How Do We Configure Middleware Components Using the Run() Extension Method?**

Now, let us see how to configure Middleware Components using Run Extension. So, modify the Main method of the Program class as follows to add a new custom Inline Middleware Component using the Run Extension Method.

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

// Initializes the configuration for the web application

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

// Builds the web application based on the configured settings

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

//Configuring New Inline Middleware Component using Run Extension Method

app.Run**(async** **(**context**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Getting Response from First Middleware"**)**;

**})**;

// Starts the web application which begins listening for incoming requests

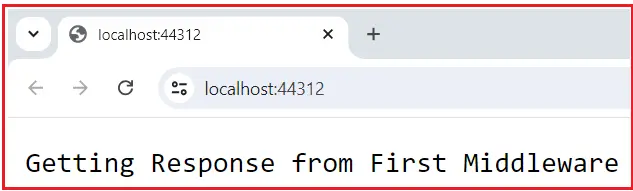
app.Run**()**;

**}**

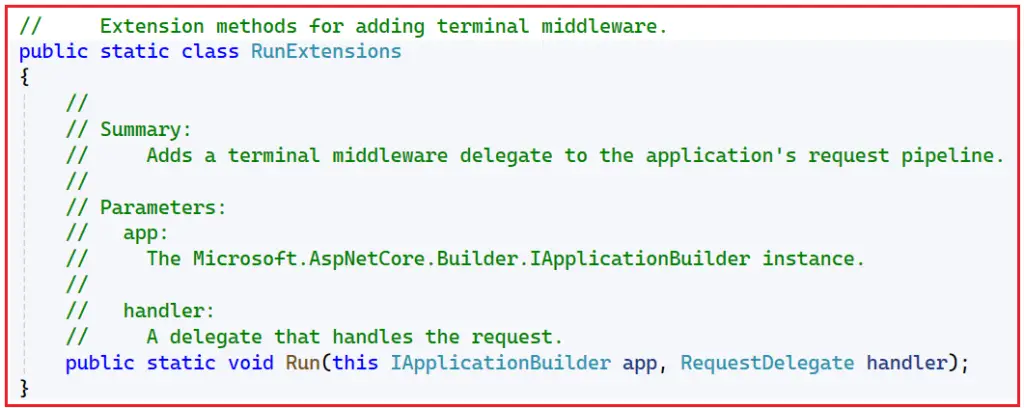
**}**

**}**

Now, run the application, and you should get the expected output, as shown in the image below.



If you go to the definition of the **Run Extension Method**, you will see the following signature. It basically adds a terminal middleware delegate to the application’s request pipeline.



Here, you can see that it clearly says that this is an Extension Method used for adding terminal middleware components. Terminal Middleware is the last middleware component. If you are new to the extension method, please read the article below, where we discussed the extension methods in detail.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=5c4134cd-7791-4646-61b9-9eb0fa4802ab&d_id=118560&imp_id=8588545762725972&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

[**https://dotnettutorials.net/lesson/extension-methods-csharp/**](https://dotnettutorials.net/lesson/extension-methods-csharp/)

You can also see from the above definition of the Run() Extension method that it takes an input parameter of **RequestDelegate**. Now, if you go to the definition of RequestDelegate, then you will see the following.



As you can see in the above image, the RequestDelegate is a delegate that takes an input parameter of type **HttpContext** object. If you are new to delegates, I strongly recommend that you read the following article, in which we discussed them in detail.

[**https://dotnettutorials.net/lesson/delegates-csharp/**](https://dotnettutorials.net/lesson/delegates-csharp/)

As we already discussed, the middleware components in an ASP.NET Core Web Application can access both HTTP requests and responses because of the above HttpContext object. In our example, we are passing an anonymous method or delegate to the Run Extension method and, moreover, passing the HTTP Context object as an input parameter. The following diagram shows the above.



Instead of passing the request delegate inline as an anonymous method, we can define it in a separate method and pass it here. For a better understanding, please modify the Program class code as follows.

Advertisements

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

// Initializes the configuration for the web application

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

// Builds the web application based on the configured settings

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

//Configuring New Inline Middleware Component using Run Extension Method

app.Run**(**FirstMiddleware**)**;

// Starts the web application which begins listening for incoming requests

app.Run**()**;

**}**

//This method signature must be the same as the RequestDelegate signature

**private** **static** **async** Task FirstMiddleware**(**HttpContext context**)**

**{**

//Using context object, we can access both Request and Response

**await** context.Response.WriteAsync**(**"Getting Response from First Middleware"**)**;

**}**

**}**

**}**

In this example, the FirstMiddleware method is defined separately and then passed to the Run extension method. Now, run the application, and you should see the same output.

**Adding Multiple Middleware Components into the Request Processing Pipeline:**

Let us modify the Program class as follows to add two custom inline middleware components to the Request processing pipeline.

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

// Initializes the configuration for the web application

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

// Builds the web application based on the configured settings

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

//Configuring First Middleware Component using Run Extension Method

app.Run**(async** **(**context**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Getting Response from First Middleware"**)**;

**})**;

//Configuring Second Middleware Component using Run Extension Method

app.Run**(async** **(**context**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Getting Response from Second Middleware"**)**;

**})**;

// Starts the web application which begins listening for incoming requests

app.Run**()**;

**}**

**}**

**}**

Now, we have two middleware components registered using the Run() extension method. Run the application, and you will get the following output.

The output is coming from the first middleware component. When we register a middleware component using the Run() extension method, that component becomes a terminal component, which means it will not call the next middleware component in the request processing pipeline.

Then, the question that should come to your mind is how to call the next middleware component in the request processing pipeline. The answer is to register the middleware component using the Use extension method.

**Configuring Middleware Components Using the Use Extension Method**

Let us modify the Program class code as follows to register the Middleware component using the Use extension method. As you can see below, the First middleware component is registered using the Use extension method, and the second middleware component is registered using the Run extension method.

As you can see in the code below, in the first Use Extension method, we pass two input parameters to the anonymous method, i.e., context and next. Then, we call the next method, which will call the next middleware component registered in the Request Processing Pipeline.

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

// Initializes the configuration for the web application

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

// Builds the web application based on the configured settings

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

//Configuring First Middleware Component using Use Extension Method

app.Use**(async** **(**context, next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Getting Response from First Middleware"**)**;

**await** next**()**;

**})**;

//Configuring Second Middleware Component using Run Extension Method

app.Run**(async** **(**context**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"\nGetting Response from Second Middleware"**)**;

**})**;

// Starts the web application which begins listening for incoming requests

app.Run**()**;

**}**

**}**

**}**

Now run the application, and you will see the output as expected, which is coming from both middleware components, as shown in the image below.

**Advertisements**

**Understanding the Use Extension Method in ASP.NET Core:**

If you go to the definition of the Use extension method, you will see the following signature.

This method has also been implemented as an extension of the IApplicationBuilder interface. This is the reason why we are able to invoke this method using the IApplicationBuilder instance. As you can see from the above definition, this method takes two input parameters. The first parameter is the HttpContext context object, through which it can access both the HTTP request and response. The second parameter is the Func type, i.e., it is a generic delegate that can handle the request or call the next middleware component in the request processing pipeline.

**Can we create a Terminal Middleware Component using the Extension method?**

Yes, it is possible to create a terminal middleware component using the Use extension method in ASP.NET Core. A terminal middleware component handles the request completely and does not call the next() delegate to pass the request to the next middleware component. In this case, we need to specify the context object and request delegate explicitly. For a better understanding, please modify the Program class as follows.

Advertisements

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

// Initializes the configuration for the web application

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

// Builds the web application based on the configured settings

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

//Configuring First Middleware Component using Use Extension Method

//This will also act as a terminal Middleware Component

app.Use**(async** **(**HttpContext context, RequestDelegate next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Getting Response from First Middleware"**)**;

// Not calling 'next' since we want to terminate the pipeline here.

**})**;

//Configuring Second Middleware Component using Run Extension Method

app.Run**(async** **(**context**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"\nGetting Response from Second Middleware"**)**;

**})**;

// Starts the web application which begins listening for incoming requests

app.Run**()**;

**}**

**}**

**}**

**Differences Between Run, Map, and Use Extension Method in ASP.NET Core**

In ASP.NET Core, Run, Map, and Use are extension methods for configuring the middleware pipeline. Each method serves a distinct purpose in controlling the flow of HTTP requests in the request processing pipeline. Understanding the differences between these extension methods is very important for effective pipeline management and middleware configuration.

**Run Extension Method:**

* The Run extension method terminates the middleware pipeline. It does not accept the next parameter because it is expected to be the final point in the middleware chain.
* The Run is used when no further processing should occur after the current middleware. Once a middleware calls Run, no subsequent middleware is executed. It is typically used at the end of the middleware pipeline.

**Map Extension Method:**

The Map extension method creates an endpoint routing middleware component. It allows us to specify middleware that should only run for specific request paths.

It is useful for creating separate pipelines for different URL paths of the incoming request. This helps handle different routes within the same application.

**Use Extension Method:**

* The Use extension method is the most flexible. It allows us to pass requests to the next middleware in the pipeline. It takes a function that can do work before and/or after the next middleware.
* It is commonly used to add middleware that performs actions before calling the next middleware component to process the request or to do additional processing after the next middleware has been completed.

**Note:** If you use the Map and Run middleware, you will see that the Map endpoint might not write the output to the Response stream.

In the next article, I will discuss the [**ASP.NET Core Request Processing Pipeline**](https://dotnettutorials.net/lesson/asp-net-core-request-processing-pipeline/) with Examples. In this article, I explain **How to use Middleware Components in the ASP.NET Core** **Application** to handle the request processing pipeline with an example. I hope you enjoy this article on ASP.NET Core Middleware Components with Examples