**Introduction**

Hello! In this course, you will learn ASP.NET Core 6 from scratch. You will learn MVC, web services, how to use Entity Framework Core with ASP.NET, how to use ASP.NET's Identity, and much more. As a whole, I do expect you to have some experience with C#, Entity Framework Core, and web development in general. Please like and subscribe to support the channel for more free premium courses.

**Tools Overview**

In this lecture, I just quickly want to go over the tools I'll be using. First of all, I'm going to be using **Visual Studio 2022 Community**. It's the best so far, and I recommend you get it. You can do ASP.NET development with **Visual Studio Code** as well, but I recommend Visual Studio IDE because of the support.

I'm also going to be using **Chrome** as my browser and **Postman**. This is a great program to use when developing web services, and it's free, so make sure to get that as well.

**GitHub Repository and Commits**

Finally, I want to mention that I commit after each lecture, so you can access any file by cloning this repo. Here I am at the repo, and I’m going to go to **Code** and copy the link to show you how to clone it in Visual Studio. Paste it there and click **Clone**.

Now, go into the folder. If I go to **Git** and view the branch history, you will see all the commits. Let me switch to the **Detailed View**, for example. If you want to see the code for detecting HTTPS, just double-click the commit and view the files. You can also right-click to open the file nicely, or right-click and reset/delete changes to get back to that point without the later commits.

If you want to see some code from a later point, you can simply clone again. This is just a tip for you all.

**Setting Up a New Project**

Now, let’s get started. I’m going to create a new project. Go to **Create a New Project**, and choose **Web**. You would probably want to use the **MVC** template for websites or the **Web API** template for web services because they already handle a lot of the groundwork for you. However, for the course, I’m going to start with the **Core Empty** template and add what we need along the way.

I’ll call the project **Core**, and the rest of the settings are fine for me. Click **Next**. Select **.NET 6**, and make sure **HTTPS** is enabled. Then click **Create**.

**Exploring the Project**

Here we are! As you can see, there isn’t much here in the empty template. Let's first check out the **launchSettings.json** file inside the **Properties** folder. This file is used to configure the HTTP and HTTPS ports when the ASP.NET Core platform is started through IIS Express, which is how older versions of ASP.NET Core were deployed.

The **profiles** section describes different launch profiles, defining the configuration for different ways of running the application. The **projectName** section defines the configuration used by the **dotnet run** command. If you start the app either through the console command or by just launching it, this file is used.

**Configuring Launch Settings**

For now, I’m going to change the **http** port to **3000** and leave the **https** port as is, since I won’t be using it for now. I’ll also set **launchBrowser** to **false**.

To run the application, you can start it normally by clicking the **Run** button or by using the console. Let’s see how to use the console first. Right-click the project and select **Open in Terminal**. Now, type **dotnet watch**. If I go to **localhost:3000**, I can see that it works. This is coming from the **Program.cs** file, which I’ll explain in a bit.

**Running the Application**

The difference between **dotnet run** and **dotnet watch** is that **dotnet run** works better with the program files and settings, while **dotnet watch** refreshes the browser with a hot reload feature (which is new in Visual Studio 2022).

If you don’t see a change after editing the code, sometimes you’ll need to restart the application. Personally, I prefer using **Start Without Debugging** (Ctrl + F5), which runs the **dotnet watch** command anyway. Visual Studio is much faster now, so building the application and running it is quicker than before.

**Middleware and Services Overview**

The purpose of the ASP.NET Core platform is to receive HTTP requests and send responses. ASP.NET delegates this process to middleware components, arranged in a chain called the **request pipeline**. When a new HTTP request arrives, it goes through the middleware components, which inspect and modify the request if needed. Once the request has passed through the pipeline, the ASP.NET platform sends the response.

Some components generate responses for requests, while others provide supporting features, such as formatting data or reading and writing cookies. If no response is generated by the middleware components, ASP.NET will return a **404 Not Found** status code.

**Services and Dependency Injection**

Services are objects that provide features in a web application. Any class can be used as a service, and there are no restrictions on the features that services provide. What makes services special is that they are managed by ASP.NET, and the **dependency injection** feature allows you to easily access services anywhere in the application, including middleware components. Dependency injection will be covered later on in the course.

**Program.cs File**

The **Program.cs** file contains the code executed when the application starts. It’s used to configure the ASP.NET platform and individual frameworks it supports. If you’ve used earlier versions of ASP.NET, you might remember that there used to be a **Startup.cs** file. Now, it’s all in the **Program.cs** file.

First, we have the **CreateBuilder** method, which sets up the basic features of the ASP.NET platform, such as creating services for configuration data and logging. Then, the **Build** method sets up the middleware components. The **Core Empty** template has set up one middleware component.

The **MapGet** method is used to define an endpoint (or route). Endpoints represent the request-handling code that gets executed when a URL match occurs. In this case, the route is the root of the domain, which returns a simple string like "Hello World."

Finally, the **Run** method starts listening for HTTP requests.

**Project File (csproj)**

The **csproj** file contains the information that the .NET Core platform uses to build the project and keep track of dependencies. This file is usually hidden but can be accessed by right-clicking the project and selecting **Edit Project File** or by simply double-clicking the project.

In most projects, you won’t need to edit the project file directly. The most common change is adding dependencies. For example, if you have dependencies in one project and create another project in the same solution, you can copy-paste the dependencies, and they will be installed in the new project as well.

**Adding a Dependency**

Let’s add a dependency now. Go to **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution**, search for **IdentityEntity**, select the project, and click **Install**. After installing, you can go back to the **csproj** file, and you’ll see the added dependencies listed there.

**Custom Middleware**

ASP.NET provides various middleware components. To understand it better, let’s create our own custom middleware. In the **Program.cs** file, above the **MapGet** method, use the **Use** method to create custom middleware.

Here’s an example:

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

if (context.Request.Method == HttpMethod.Get.Method

&& context.Request.Query["custom"] == "true") {

// Custom logic here

context.Response.ContentType = "text/plain";

await context.Response.WriteAsync("Custom Middleware");

await next();

}

await next.Invoke();

});

**Understanding the Custom Middleware Flow**

What's happening here exactly? When any GET request with the appropriate query string (?custom=true) comes in (for us, it's the root request because that's the only route we're handling), it first goes through the custom middleware. Then, it calls the next() method, which invokes the next middleware in the pipeline (in our case, app.MapGet()), which handles the request and produces a response.

**Testing the Middleware**

To see this in action, I’ll press **Ctrl + F5** (Start without debugging) and go to localhost:3000. Here is the text from the endpoint. But if I add the query string custom=true, I get the custom response.

**Understanding HTTP Context**

This code is easy to understand. The context is the HTTP context, which you can inspect by hovering over it. The context.Request and context.Response objects contain useful properties, such as Method, Query, and ContentType, which we’ve used here. You may also encounter middleware that calls next.Invoke() instead of just next(), and these are equivalent. The next() method is provided as a convenience by the compiler to produce concise code.

## Class-Based Middleware

Middleware can also be defined using classes. Let’s add a class called Middleware.

### Create a Middleware Class:

* Add a private RequestDelegate field called \_next.
* Create a constructor that accepts RequestDelegate and assigns it to \_next.
* Create an Invoke method that accepts HttpContext and writes the response.

public class Middleware(RequestDelegate next)

{

public async Task Invoke(HttpContext context) {

if (!context.Response.HasStarted) {

context.Response.ContentType = "text/plain; charset=utf-8";

await context.Response.WriteAsync("Class-based Middleware");

}

await next(context);

}

}

### Adding Middleware to the Pipeline

To add this middleware to the request pipeline, go to Program.cs and replace the existing middleware with the new class-based middleware:

app.UseMiddleware<Middleware>();

This will add the class-based middleware to the request pipeline.

### Testing Class-Based Middleware

Now, press **Ctrl + F5** to start the application. If you refresh the page, you’ll see "Class-based Middleware" when the query string custom=true is present. Otherwise, you’ll just get "Hello World".

**Modifying HTTP Responses After Middleware**

Middleware components can modify the HTTP response object after the next() function has been called. To demonstrate this:

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

{

await next();

await context.Response.WriteAsync("Status Code: " + context.Response.StatusCode);

});

This new middleware immediately calls the next() method to pass the request along the pipeline, and after that, it adds a string to the response body. This allows middleware to make changes to the response both before and after it is passed along the pipeline.

**Testing Response Modifications**

Press **Ctrl + F5** to start the application, and when you refresh the page, you’ll see the status code added after the response has been generated.

## Short-Circuiting the Pipeline

Components that generate complete responses can choose not to call the next() function, which causes the request to be stopped and the pipeline short-circuited. For example:

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

if (context.Request.Path == "/short") {

await context.Response.WriteAsync("Request short-circuited");

}

else {

await next();

}

});

In this case, if the URL path is /short, the request is short-circuited and no further middleware is invoked.

**Testing Short-Circuiting**

Start the project, and if you go to the root URL, you get "Hello World". If you go to /short, you get "Request Short-Circuited", and the pipeline stops there.

## Using the Map Method for URL-Specific Middleware

The Map method is used to create a section of the pipeline that processes requests for specific URLs, creating a separate sequence of middleware components.

Here’s how to use the Map method:

app.Map("/branch", branchApp =>

{

branchApp.Use(async (context, next) =>

{

await context.Response.WriteAsync("Branch Middleware");

});

});

This allows you to define middleware that only processes requests for a specific route (/branch in this case).

**Testing URL-Specific Middleware**

Press **Ctrl + F5** to start the project, and navigate to /branch to see the "Branch Middleware" response.

## Terminal Middleware

Terminal middleware components don’t forward requests to other middleware in the pipeline and always mark the end of the request processing. The Run method is often used for terminal middleware.

app.Map("/branch", branchApp =>

{

branchApp.Run(async context =>

{

await context.Response.WriteAsync("Terminal Middleware");

});

});

This stops the pipeline once the response is generated, and no further middleware is executed.

## Class-Based Middleware as Terminal Middleware

Class-based middleware can be written so that it can be used both as regular middleware and as terminal middleware. To do this, ensure that the Invoke method checks whether the next delegate is null:

public class Middleware2()

{

private readonly RequestDelegate? \_next;

public Middleware2(RequestDelegate? next) : this() {

\_next = next;

}

public async Task Invoke(HttpContext context) {

if (!context.Response.HasStarted) {

context.Response.ContentType = "text/plain; charset=utf-8";

await context.Response.WriteAsync("Class-based Middleware2\r");

}

if (\_next != null) {

await \_next(context);

}

}

}

**Testing Class-Based Terminal Middleware**

Press **Ctrl + F5** to start the project, and test the /branch route. The response should display "Class-based Terminal Middleware2" if it’s terminal middleware.

## The Options Pattern in Middleware

The options pattern is commonly used to configure middleware components in ASP.NET Core. Here’s how it works:

**Define an Options Class:**

Create a class that holds configuration options for middleware. For example:

public class FruitOptions {

public string Name { get; set; } = "Apple";

public string Color { get; set; } = "Green";

}

**Register the Options Class** in Program.cs:

builder.Services.Configure<FruitOptions>(options => {

options.Name = "Apple";

options.Color = "Green";

});

**Use the Options in Middleware**:

Middleware can access these options using dependency injection.

**Using the Options Pattern with Middleware**

To implement the options pattern for configuring middleware, we will follow these steps:

**Add Configuration to Program.cs**:

After defining the FruitOptions class, we need to configure it in Program.cs so that the application can inject it into middleware.

builder.Services.Configure<FruitOptions>(options =>

{

options.Name = "Apple";

options.Color = "Green";

});

## Create Middleware that Uses the Options:

In the middleware, we will access the FruitOptions using dependency injection. The middleware can then use the configured options to alter its behavior or response.

For example, here’s a class-based middleware that writes the configured fruit name and color to the response:

public class FruitMiddleware

{

private readonly RequestDelegate \_next;

private readonly IOptions<FruitOptions> \_options;

public FruitMiddleware(RequestDelegate next, IOptions<FruitOptions> options)

{

\_next = next;

\_options = options;

}

public async Task InvokeAsync(HttpContext context)

{

context.Response.ContentType = "text/plain; charset=utf-8";

// Use the options to generate a custom response

var fruit = \_options.Value.Name;

var color = \_options.Value.Color;

await context.Response.WriteAsync($"Fruit: {fruit}, Color: {color}");

// Call the next middleware

await \_next(context);

}

}

### Register the Middleware in the Pipeline:

In Program.cs, you need to add the middleware to the request pipeline. Ensure that the middleware is added after the services are configured.

app.UseMiddleware<FruitMiddleware>();

### Testing the Options Pattern in Middleware

Press **Ctrl + F5** to start the project. When you visit the site, the response should include:

Fruit: Apple, Color: Green

This demonstrates how the options pattern is used to configure middleware and inject configuration into it.

### Dependency Injection in Middleware

The options pattern often works alongside **dependency injection** (DI) in ASP.NET Core. In this case, we used IOptions<T> to inject configuration values into the middleware. DI is a powerful tool that allows components to be loosely coupled and tested independently.

You can also inject other services into middleware components. For example:

public class LoggingMiddleware

{

private readonly RequestDelegate \_next;

private readonly ILogger<LoggingMiddleware> \_logger;

public LoggingMiddleware(RequestDelegate next, ILogger<LoggingMiddleware> logger)

{

\_next = next;

\_logger = logger;

}

public async Task InvokeAsync(HttpContext context)

{

\_logger.LogInformation("Request received at {Time}", DateTime.UtcNow);

await \_next(context);

}

}

In this example, we injected an ILogger into the middleware to log information about incoming requests.

**Registering Services for Dependency Injection**

To make sure the services are available for DI, they need to be registered in the ConfigureServices method of Program.cs. This can include:

* Configuring options using builder.Services.Configure<T>().
* Registering other services like logging, databases, etc.

For example, to register logging:

builder.Services.AddLogging();

## Middleware Order and Execution Flow

The order in which middleware components are added to the pipeline is important. Middleware is executed in the order it is added to the pipeline. When a request comes in, it is passed through each middleware component in sequence, and the response is sent back through the components in reverse order.

Here’s an example of how the order works:

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

{

// Middleware 1

await context.Response.WriteAsync("Middleware 1: Start\n");

await next(); // Pass to the next middleware

await context.Response.WriteAsync("Middleware 1: End\n");

});

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

{

// Middleware 2

await context.Response.WriteAsync("Middleware 2: Start\n");

await next(); // Pass to the next middleware

await context.Response.WriteAsync("Middleware 2: End\n");

});

app.Run(async context =>

{

// Final terminal middleware

await context.Response.WriteAsync("Terminal Middleware\n");

});

The output for a request would be:

Middleware 1: Start

Middleware 2: Start

Terminal Middleware

Middleware 2: End

Middleware 1: End

Notice that middleware components can modify both the request (before next() is called) and the response (after next() is called). The Run method is used for terminal middleware, which doesn’t call next().

## Handling Errors in Middleware

You can handle errors within middleware to catch unhandled exceptions in the pipeline. For example, you can add an error-handling middleware that logs exceptions and sends a custom error response.

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

try {

await next();

}

catch (Exception ex) {

context.Response.StatusCode = 500;

await context.Response.WriteAsync($"Error: {ex.Message}");

}

});

This middleware catches any exceptions thrown by the downstream components and ensures that the response code is set to 500, followed by the error message.

## Middleware for Static Files

ASP.NET Core provides built-in middleware for serving static files (e.g., images, CSS, JavaScript) directly from the server. You can add this middleware using the UseStaticFiles() method.

To serve static files from a directory:

app.UseStaticFiles(); // Enable static file serving from wwwroot

This will serve files from the wwwroot folder by default. You can also configure custom directories or virtual directories.

For example:

app.UseStaticFiles(new StaticFileOptions

{

FileProvider = new PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), "Content")),

RequestPath = "/static"

});

This configuration serves static files from the Content folder and makes them available at /static in the URL path.

## Customizing Middleware

If you need to pass options or configurations to middleware, you can customize the middleware constructor to accept those options. For example:

public class CustomHeaderMiddleware

{

private readonly RequestDelegate \_next;

private readonly string \_headerValue;

public CustomHeaderMiddleware(RequestDelegate next, string headerValue)

{

\_next = next;

\_headerValue = headerValue;

}

public async Task InvokeAsync(HttpContext context)

{

context.Response.Headers.Add("X-Custom-Header", \_headerValue);

await \_next(context);

}

}

You can configure the middleware like so:

app.UseMiddleware<CustomHeaderMiddleware>("MyHeaderValue");

This middleware will add a custom header to every response.

**Conclusion**

ASP.NET Core middleware is a powerful and flexible way to modify HTTP requests and responses. You can define middleware as simple inline functions or more complex classes, use dependency injection for configuration, handle errors, and serve static files. The order in which middleware is added to the pipeline is crucial, and it can be tailored to fit the needs of your application.

By using middleware effectively, you can create highly customized web applications with robust request handling, error management, and other necessary features.

Here's a cleaner and more structured version of your text, divided into meaningful sections and paragraphs:

**Importing Services and Setting Up Endpoints**

To start, define the formatter by importing the necessary services:

formatter = ...

import context and request services.

This allows us to retrieve the required services. For example, using iResponseFormatter with a custom endpoint:

await formatter.CustomEndpoint(...);

In Program.cs, register the endpoint as follows:

app.MapGet("/endpoint", ...);

Running the application (Ctrl + F5), you can navigate to /endpoint and confirm the custom endpoint functionality.

**Exploring Dependency Injection: Singleton, Scoped, and Transient Services**

ASP.NET Core supports different service lifecycles:

1. **Singleton**: One instance per application.
2. **Transient**: A new instance for each dependency resolution.
3. **Scoped**: One instance per request.

**Example: Transient Service**

Create a GuidService implementing IResponseFormatter:

public class GuidService : IResponseFormatter {

private Guid \_guid = Guid.NewGuid();

public Task FormatAsync(...) { ... }

}

Register it in Program.cs using AddTransient:

services.AddTransient<IResponseFormatter, GuidService>();

**Demonstration**

* Duplicate the CustomEndpoint class, rename it Middleware2, and set a new URL, e.g., /middleware2.
* Observe how refreshing /middleware2 generates new GUIDs for each request, as transient services create new instances.

**Scoped Services**

Scoped services provide a shared instance within a request. Modify the service registration:

services.AddScoped<IResponseFormatter, GuidService>();

Refresh the endpoint and observe that GUIDs remain consistent across components within the same request.

**Handling Scoped Service Exceptions**

Attempting to use a scoped service outside a request context throws an exception:

Cannot resolve scoped service from root provider.

To fix this, ensure the service is used within an HTTP request scope.

**Configuration Management in ASP.NET Core**

ASP.NET Core supports multiple configuration files for different environments:

1. **appsettings.json**: General settings.
2. **appsettings.{Environment}.json**: Overrides specific settings based on the environment.

For example:

"Logging": {

"LogLevel": {

"Default": "Information"

}

}

In appsettings.Development.json:

"Logging": {

"LogLevel": {

"Default": "Debug"

}

}

Switching environments (e.g., to Production) changes the logging level automatically.

**Accessing Configuration in Code**

To retrieve configuration values, use the IConfiguration service:

app.MapGet("/config", (IConfiguration config) => {

string defaultLogLevel = config["Logging:LogLevel:Default"];

return defaultLogLevel;

});

Navigating to /config displays the current logging level based on the active environment.

**Advanced Configuration: Options Pattern**

The options pattern simplifies managing configurations by binding strongly-typed classes to configuration data. Define a class (FruitOptions) and use it in middleware. This ensures clean and maintainable code.

Let me know if you'd like additional refinements or assistance with the code.