ASP.NET Core - True Ultimate Guide

Section 4: Middleware - Notes

Middleware

At its core, middleware in ASP.NET Core is a series of components that form a pipeline through which every HTTP request and response flows. Each middleware component can:

1. **Examine** the incoming request.
2. **Modify** the request or response (if needed).
3. **Invoke** the next middleware in the pipeline or short-circuit the process and generate a response itself.

This pipeline allows you to modularize your application's logic and add features like authentication, logging, error handling, routing, and more in a clean and maintainable way.

Middleware Chain (Request Pipeline)

Imagine the request pipeline as a series of connected pipes. Each piece of middleware is like a valve in this pipeline, allowing you to control the flow of information and apply specific operations at different stages. The order you register your middleware matters, as they are executed sequentially.

app.Use vs. app.Run

These two methods are fundamental for adding middleware to your pipeline, but they have key differences:

- **app.Use(async (context, next) => { ... })**
    - **Non-Terminal Middleware:** This type of middleware typically performs some action and then calls the next delegate to pass control to the next middleware in the pipeline.
    - **Can Modify Request/Response:** It can change the request or response before passing it along.
    - **Examples:** Authentication, logging, custom headers, etc.
- **app.Run(async (context) => { ... })**
    - **Terminal Middleware:** This middleware doesn't call next; it ends the pipeline and generates the response itself.
    - **Often Used for the Final Response:** It's commonly used for handling requests that don't need further processing (e.g., returning a simple message).

- **Can't Modify Request:** Since it's the end of the line, it cannot modify the request before passing it on.

Code 1: The Consequence of Multiple app.Run Calls

```
app.Run(async (HttpContext context) => {
   await context.Response.WriteAsync("Hello");
});

app.Run(async (HttpContext context) => {
   await context.Response.WriteAsync("Hello again");
});

app.Run();
```

In this code, only the first app.Run middleware will be executed. It terminates the pipeline by writing "Hello" to the response, and the subsequent app.Run (which would write "Hello again") never gets a chance to run.

Code 2: Chaining Middleware with app.Use and app.Run

```
//middlware 1
app.Use(async (context, next) => {
   await context.Response.WriteAsync("Hello ");
   await next(context);
});

//middleware 2
app.Use(async (context, next) => {
   await context.Response.WriteAsync("Hello again ");
   await next(context);
});
```

```
//middleware 3
app.Run(async (HttpContext context) => {
   await context.Response.WriteAsync("Hello again");
});
```

This code demonstrates a correct way to chain middleware.

1. The first app.Use writes "Hello " to the response and then calls next to pass control to the next middleware.
2. The second app.Use writes "Hello again " and also calls next.
3. The final app.Run (which is terminal) writes "Hello again" and ends the pipeline. The result would be output of "Hello Hello again Hello again".

Key Points to Remember

- **Middleware Order is Crucial:** The order in which you register middleware matters, as they are executed in sequence.
- **Use app.Use for Non-Terminal Actions:** Use it for tasks like authentication, logging, or modifying headers/bodies.
- **Use app.Run to Terminate the Pipeline:** Employ it when you want to generate the final response.
- **Short-Circuiting:** Middleware can choose to short-circuit the pipeline (not call next) and return a response early if needed.

Custom Middleware in ASP.NET Cor

While ASP.NET Core provides a plethora of built-in middleware components, sometimes you need to create your own to address specific requirements unique to your application. Custom middleware allows you to:

- **Encapsulate logic:** Bundle related operations (e.g., logging, security checks, custom headers) into a reusable component.
- **Customize behavior:** Tailor the request/response pipeline to precisely match your application's needs.
- **Improve code organization:** Keep your middleware code clean and maintainable.

Anatomy of a Custom Middleware Class

1. **Implement IMiddleware:** This interface requires a single method: InvokeAsync(HttpContext context, RequestDelegate next). This is the heart of your middleware's logic.
2. **InvokeAsync or Invoke Method:**
    - context: The HttpContext provides access to the request and response objects.
    - next: The RequestDelegate allows you to call the next middleware in the pipeline.

Code Explanation

Let's dissect the code you provided:

```
// MyCustomMiddleware.cs
namespace MiddlewareExample.CustomMiddleware
{
  public class MyCustomMiddleware : IMiddleware
  {
    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
      await context.Response.WriteAsync("My Custom Middleware - Starts\n");
      await next(context);
      await context.Response.WriteAsync("My Custom Middleware - Ends\n");
    }
  }

  // Extension method for easy registration
  public static class CustomMiddlewareExtension
  {
    public static IApplicationBuilder UseMyCustomMiddleware(this IApplicationBuilder app)
    {
      return app.UseMiddleware<MyCustomMiddleware>();
    }
  }
```

}

- **MyCustomMiddleware Class:** This class implements IMiddleware. Its InvokeAsync method:
    - Writes "My Custom Middleware - Starts" to the response.
    - Calls next(context) to invoke the next middleware in the pipeline.
    - Writes "My Custom Middleware - Ends" to the response after the next middleware has finished.
- **CustomMiddlewareExtension Class:** This provides a convenient extension method UseMyCustomMiddleware to register your middleware in the Startup.Configure method.

```
// Program.cs (or Startup.cs)
using MiddlewareExample.CustomMiddleware;

// ...

builder.Services.AddTransient<MyCustomMiddleware>(); // Register as transient

app.Use(async (HttpContext context, RequestDelegate next) => {
   await context.Response.WriteAsync("From Midleware 1\n");
   await next(context);
});

app.UseMyCustomMiddleware(); // Use the extension method

app.Run(async (HttpContext context) => {
   await context.Response.WriteAsync("From Middleware 3\n");
});
```

How It Works

1. **Registration:** You register MyCustomMiddleware as a transient service so that ASP.NET Core can create instances of it when needed.

2. **Pipeline Integration:** The app.UseMyCustomMiddleware() extension method seamlessly adds your middleware to the pipeline.
3. **Execution Order:** Middleware components are executed in the order they are added to the pipeline. In this case, the order would be Middleware 1, MyCustomMiddleware, then Middleware 3.

Output

When you run the application, you'll see the following output in your browser:

From Midleware 1

My Custom Middleware - Starts

From Middleware 3

My Custom Middleware - Ends

This clearly demonstrates the flow of execution through the middleware chain.

Custom Conventional Middleware

ASP.NET Core middleware comes in two flavors: conventional and factory-based. Conventional middleware, as shown in your example, is a simple yet powerful way to encapsulate custom logic for processing HTTP requests and responses.

Key Characteristics

- **Class-Based:** Conventional middleware is implemented as a class.
- **Constructor Injection:** It receives dependencies (if any) through its constructor.
- **Invoke Method:** This is the heart of the middleware, containing the logic that handles each request.
- **RequestDelegate:** The Invoke method takes a RequestDelegate parameter (_next in your example). This delegate represents the next middleware in the pipeline.
- **Flexibility:** You have full control over the request and response objects within the Invoke method.

Code Breakdown: HelloCustomMiddleware

```
// HelloCustomMiddleware.cs

public class HelloCustomMiddleware

{
```

```
    private readonly RequestDelegate _next;

    public HelloCustomMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext httpContext)
    {
        if (httpContext.Request.Query.ContainsKey("firstname") &&
            httpContext.Request.Query.ContainsKey("lastname"))
        {
            string fullName = httpContext.Request.Query["firstname"] + " " +
httpContext.Request.Query["lastname"];
            await httpContext.Response.WriteAsync(fullName);
        }
        await _next(httpContext);
    }
}

// Extension method for easy registration
public static class HelloCustomModdleExtensions
{
    public static IApplicationBuilder UseHelloCustomMiddleware(this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<HelloCustomMiddleware>();
    }
}
```

Let's analyze each part:

1. **Constructor:** The constructor receives the RequestDelegate, which is stored for later use to invoke the next middleware in the pipeline.

2. **Invoke Method:**
    - It checks if the query string contains both "firstname" and "lastname" parameters.
    - If so, it combines the values into a fullName string and writes it to the response.
    - **Crucially:** It calls await _next(httpContext); to continue the middleware chain. This line ensures that the request is passed on to subsequent middleware components, even if a full name is generated.
    - By design, any code after this line, such as the comment "//after logic", would not execute for requests containing both "firstname" and "lastname", as the await _next(httpContext); line immediately transfers control to the next middleware in the pipeline.
3. **UseHelloCustomMiddleware Extension:** This extension method simplifies the registration process by hiding the details of instantiating and using your custom middleware class.

Program.cs (or Startup.cs): Using the Middleware

```
// ... other middleware ...

app.UseMyCustomMiddleware();

app.UseHelloCustomMiddleware();

// ...
```

How It Works

1. When a request arrives, ASP.NET Core traverses the middleware pipeline.
2. It reaches HelloCustomMiddleware, which checks for the specific query parameters.
3. If the parameters are present, the middleware generates a personalized greeting.
4. Regardless of whether it generates the greeting, the middleware calls next(context) to pass the request along to the next middleware component in the pipeline.

Key Points

- **Simplicity:** Conventional middleware is easy to write and understand.
- **Control:** You have fine-grained control over how the request is processed and how the response is generated.
- **Extension Methods:** Use extension methods to make middleware registration clean and readable.

The Ideal Order of Middleware Pipeline

1. **Exception/Error Handling:**
    - **Purpose:** Catches and handles exceptions that occur anywhere in the pipeline.
    - **Examples:** UseExceptionHandler, UseDeveloperExceptionPage (for development environments).
2. **HTTPS Redirection:**
    - **Purpose:** Redirects HTTP requests to HTTPS for security.
    - **Example:** UseHttpsRedirection.
3. **Static Files:**
    - **Purpose:** Serves static files like images, CSS, and JavaScript directly to the client.
    - **Example:** UseStaticFiles.
4. **Routing:**
    - **Purpose:** Matches incoming requests to specific endpoints based on their URLs.
    - **Examples:** UseRouting, UseEndpoints.
5. **CORS (Cross-Origin Resource Sharing):**
    - **Purpose:** Enables secure cross-origin requests from different domains.
    - **Example:** UseCors.
6. **Authentication:**
    - **Purpose:** Verifies user identities and establishes a user principal.
    - **Example:** UseAuthentication.
7. **Authorization:**
    - **Purpose:** Determines whether a user is allowed to access a particular resource or perform a certain action.
    - **Example:** UseAuthorization.
8. **Custom Middleware:**
    - **Purpose:** Your application-specific middleware components to handle tasks like logging, feature flags, etc.

Reasoning Behind the Order

- **Early Exception Handling:** Catching exceptions early prevents them from propagating and causing further issues down the pipeline.

- **Security First:** HTTPS redirection, authentication, and authorization are essential for securing your application.
- **Performance Optimization:** Static files, response caching, and compression are placed early to optimize the response generation process.
- **Routing as a Foundation:** Routing determines how requests are handled by your application's core logic.
- **CORS for Flexibility:** CORS allows your application to be consumed by a wider range of clients.
- **Custom Middleware:** Your custom middleware can be placed strategically within the pipeline to apply logic at the appropriate stage.

Flexibility and Exceptions

While this is the general recommended order, there might be exceptions based on your application's specific needs. For instance:

- **Health Checks:** You might want to place health check middleware very early in the pipeline to quickly determine the application's status without executing other middleware components.
- **Specialized Middleware:** Some middleware components may have specific ordering requirements documented by their providers.

Example (Program.cs or Startup.cs):

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

var app = builder.Build();


if (app.Environment.IsDevelopment())

{

   app.UseDeveloperExceptionPage();

}


app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();

app.UseAuthorization();
```

```
// ... your custom middleware ...


app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers(); // Or MapRazorPages(), MapGet(), etc.
});
```

By adhering to this recommended order, you'll create a well-structured and efficient ASP.NET Core application that's easier to maintain, debug, and secure.

UseWhen()

UseWhen() is a powerful extension method in ASP.NET Core's IApplicationBuilder interface. It allows you to conditionally add middleware to your request pipeline based on a predicate (a condition). This means you can create dynamic pipelines where specific middleware components are executed only when certain conditions are met.

Syntax

```
app.UseWhen(
    context => /* Your condition here */,
    app => /* Middleware configuration for the branch */
);
```

- **context:** The HttpContext object representing the current request.
- **Predicate (Condition):** A function that takes the HttpContext and returns true if the middleware branch should be executed, false otherwise.
- **Middleware Configuration:** An action that configures the middleware components that should be executed if the condition is true. This is where you use app.Use(), app.Run(), or other middleware registration methods.

How UseWhen() Works

1. **Predicate Evaluation:** When a request comes in, the UseWhen() method first evaluates the predicate function against the HttpContext.
2. **Branching (if true):** If the predicate returns true, the middleware branch specified in the configuration action is executed. The request flows through this branch, potentially undergoing modifications or generating a response.
3. **Rejoining the Main Pipeline:** After the branch is executed (or skipped if the predicate was false), the request flow rejoins the main pipeline, continuing with the next middleware components registered after the UseWhen() call.

Code Example: Explained

```
app.UseWhen(
   context => context.Request.Query.ContainsKey("username"),
   app => {
      app.Use(async (context, next) =>
      {
         await context.Response.WriteAsync("Hello from Middleware branch");
         await next();
      });
   });

app.Run(async context =>
{
   await context.Response.WriteAsync("Hello from middleware at main chain");
});
```

- **Condition:** The predicate context.Request.Query.ContainsKey("username") checks if the query string contains a parameter named "username".
- **Branch Middleware:** If the "username" parameter is present, the branch middleware is executed. It writes "Hello from Middleware branch" to the response and then calls next to allow the rest of the pipeline to continue.
- **Main Pipeline:** The final app.Run middleware is part of the main pipeline. It writes "Hello from middleware at main chain" to the response.

Output

- If the request contains the "username" query parameter (e.g., /path?username=John), the output will be:
- Hello from Middleware branch

Hello from middleware at main chain

- If the request does not contain the "username" parameter (e.g., /path), the output will be:

Hello from middleware at main chain

When to Use UseWhen()

- **Conditional Features:** Enable or disable certain features based on the request (e.g., logging only for certain users, applying caching rules based on query parameters).
- **Dynamic Pipelines:** Create pipelines that adapt to different requests (e.g., different authentication middleware for specific routes).
- **A/B Testing:** Route a subset of users through alternative middleware branches for experimentation.
- **Debugging and Diagnostics:** Apply diagnostic middleware only in development environments.

Key Points to Remember:

Conceptual Understanding:

1. **The Pipeline:** Middleware forms a pipeline for HTTP requests and responses. Each component can inspect, modify, or terminate the flow.
2. **Order Matters:** Middleware is executed in the order it's registered. Think carefully about the sequence.
3. **Types of Middleware:**
    - **Built-in:** ASP.NET Core offers middleware for authentication, routing, static files, etc.
    - **Custom:** You can create your own to add specific logic to your app.

app.Use vs. app.Run:

1. **app.Use:** For non-terminal middleware. It calls next to pass control to the next component.
2. **app.Run:** For terminal middleware. It ends the pipeline and generates a response.

Custom Middleware:

1. **Two Ways:**
    - **Conventional:** Class-based, using the Invoke method and constructor injection.
    - **Factory-Based:** Uses a delegate to create the middleware instance.
2. **Benefits:** Encapsulates logic, improves code organization, and allows you to tailor your application's behavior.

Recommended Order: (Not strict, but a good guideline)

1. Exception Handling
2. HTTPS Redirection
3. Static Files
4. Routing
5. CORS
6. Authentication
7. Authorization
8. Custom Middleware
9. MVC/Razor Pages/Minimal APIs

Bonus Points:

- **Short-Circuiting:** Middleware can choose not to call next and return a response early.
- **UseWhen:** Conditionally add middleware branches based on request criteria.
- **Middleware Ordering Flexibility:** Understand the reasons behind the recommended order, but also know when to deviate from it based on your application's specific requirements.