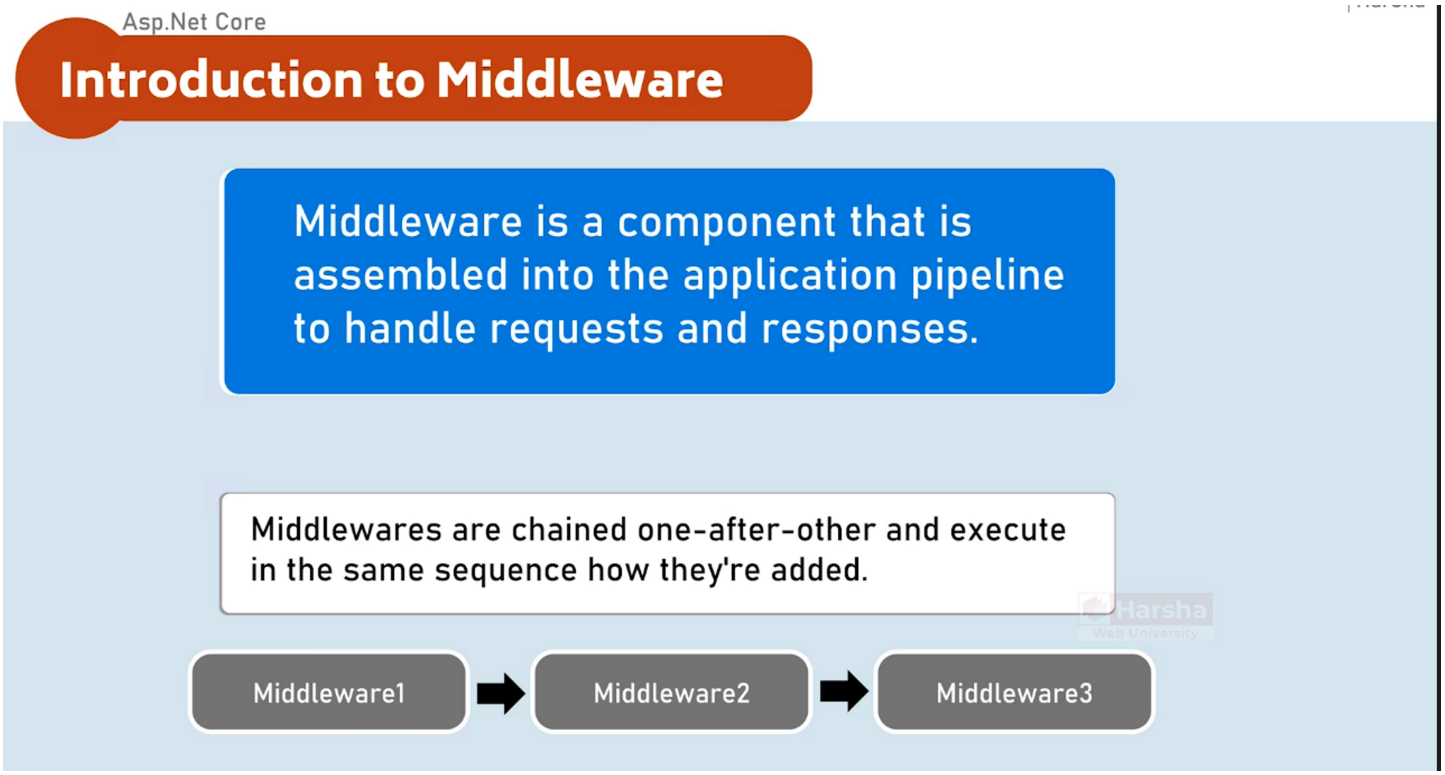
## Section 4: Middleware [MVC and Web API]

29 December 2024 22:34


Middleware is a component assembled into the application pipeline to handle requests and responses. To understand this, imagine a set of methods that execute sequentially whenever a request is received. These methods are called middleware.

For example, when a request is received, the first middleware executes, followed by the second, third, and so on. After all middleware has executed, the response is sent back to the browser. Middleware is chained in the sequence they are created or added.

Initially, the application pipeline is empty, and you add middleware one by one. Each middleware performs a single operation, following the single responsibility principle. For example:

The first middleware handles HTTPS redirection.
The second enables static file redirection.
The third manages authentication.
The fourth manages authorization.
This approach makes it easy to understand each middleware's purpose and to modify the sequence. For example, removing one middleware doesn't affect others' functionality.

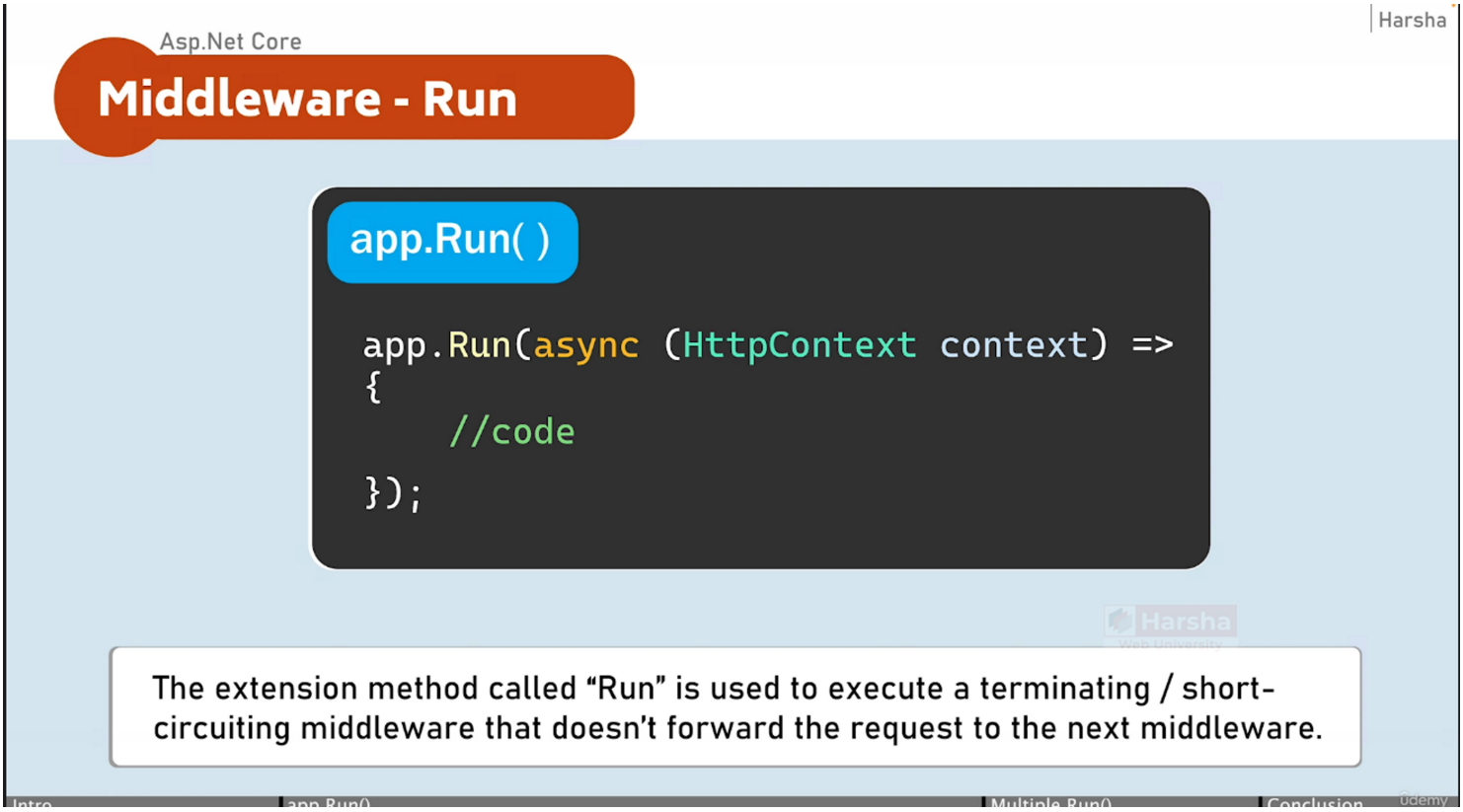
Middleware can be created in two ways:

As a request delegate (anonymous method or lambda expression) for simple operations.
As a separate middleware class for more complex functionality.
Additionally, not all middleware must forward requests to the next in the pipeline. Middleware that doesn't forward requests is called terminal middleware or short-circuiting middleware.

We’ll explore middleware development and execution in further lectures in this section.

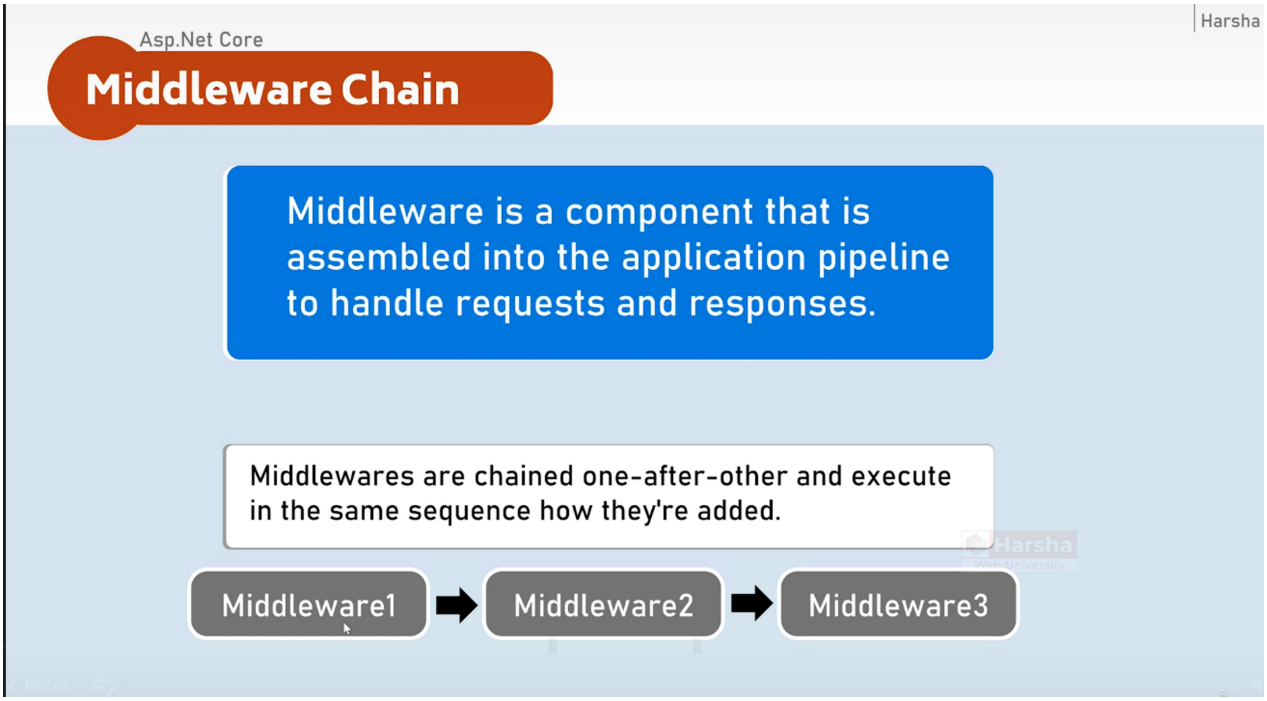
# Run


Asp.Net Core
Run


Harsha
Asp.Net Core
Middleware - Run
app.Run( )
app.Run(async (HttpContext context) =>
{
//code
});
The extension method called "Run" is used to execute a terminating / short-circuiting middleware that doesn't forward the request to the next middleware.
Udemy

Asp.Net Core

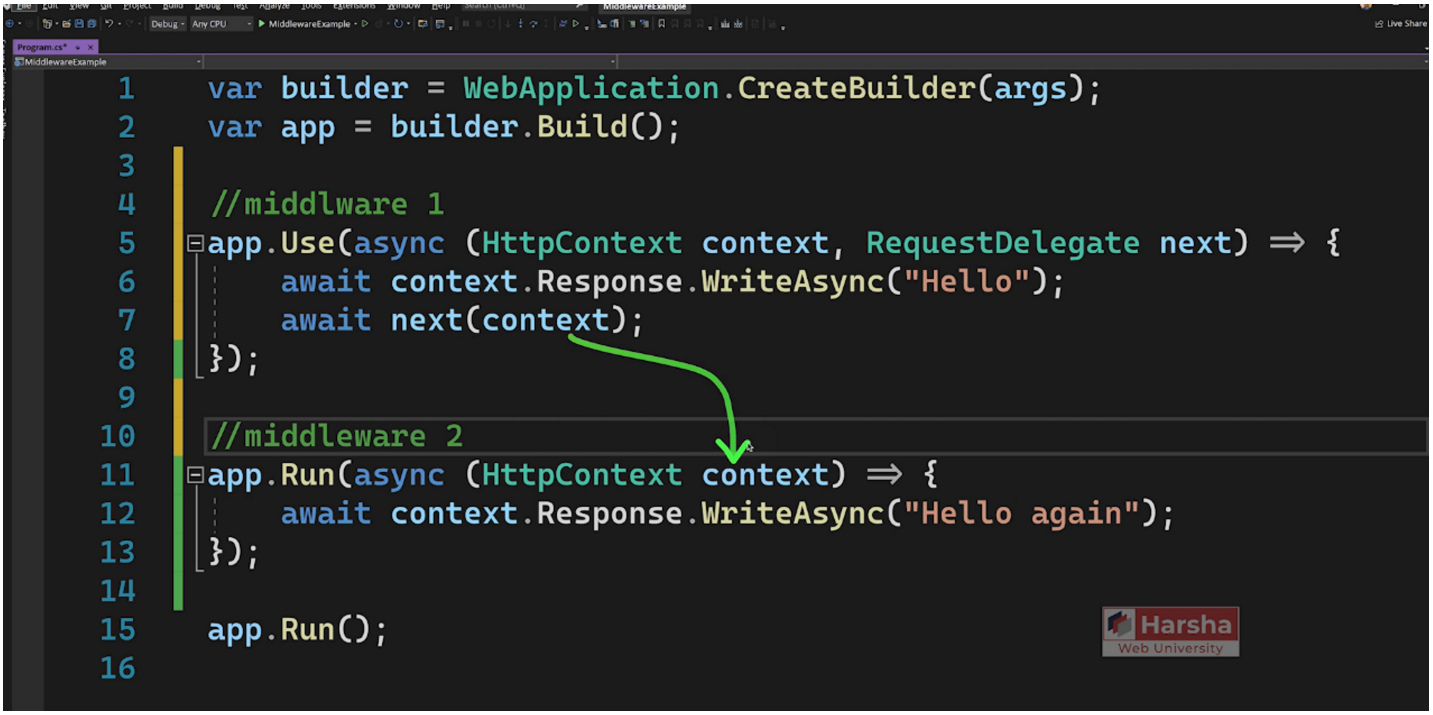
# Middleware Chain

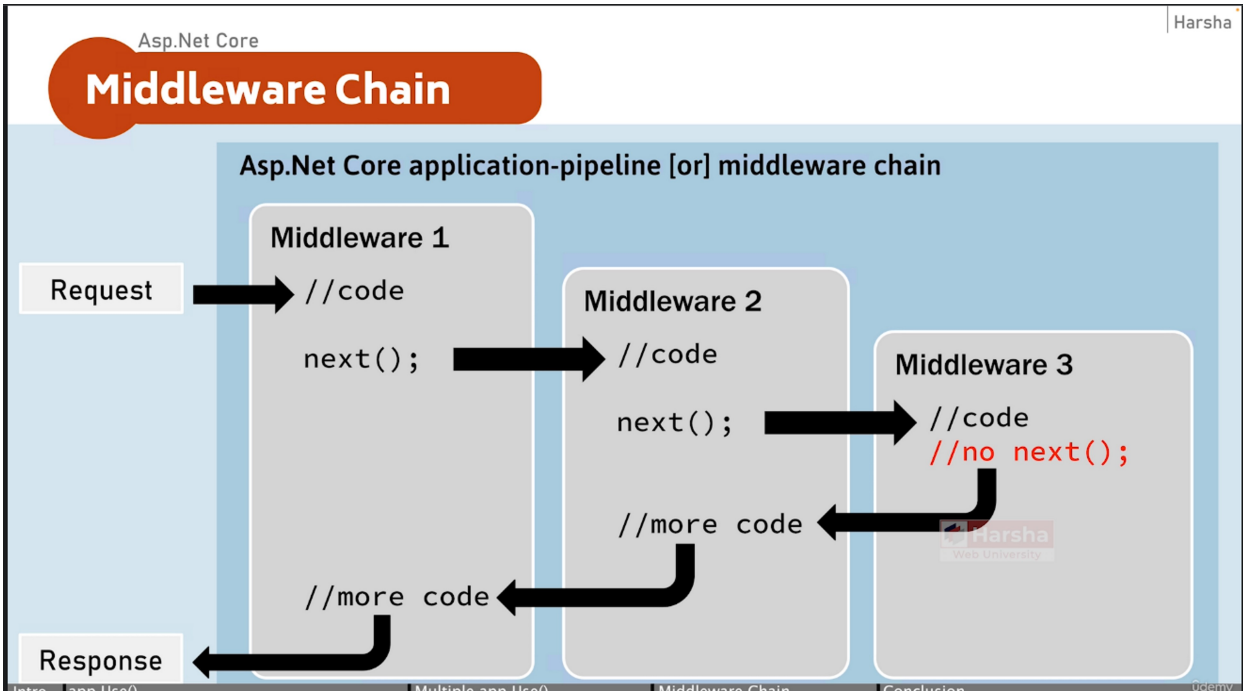



Middlewares can be chained one after another. When a request is received, the first middleware executes and may forward the request to the next middleware (Middleware 2). Middleware 2 can also forward the request to Middleware 3, and so on. However, forwarding the request to the subsequent middleware is optional. Let's see how to implement this.

You can use app.use instead of app.run. The app.use method creates middleware that can either forward the request to subsequent middleware or short-circuit the chain, terminating it. In this case, the lambda expression should receive two arguments:

HttpContext: Contains properties like Request, Response, Session, etc.
RequestDelegate: Represents the next middleware in the pipeline.






Middleware performs individual responsibilities, such as enabling HTTPS redirection or authorization. If a middleware (e.g., Middleware 3) doesn't call the next method, it is called terminating middleware. After Middleware 3 completes execution, it returns to Middleware 2 at the point where next was called, executing any code after next. Once Middleware 2 finishes, it returns to Middleware 1, where code after its next call is executed. Finally, the combined response is sent back to the browser. This is how the middleware chain executes.

Asp.Net Core

# Middleware Chain

app.Use( )

```
app.Use(async (HttpContext context, RequestDelegate next) =>
{
    //before logic
    await  next(context);
    //after logic
});
```

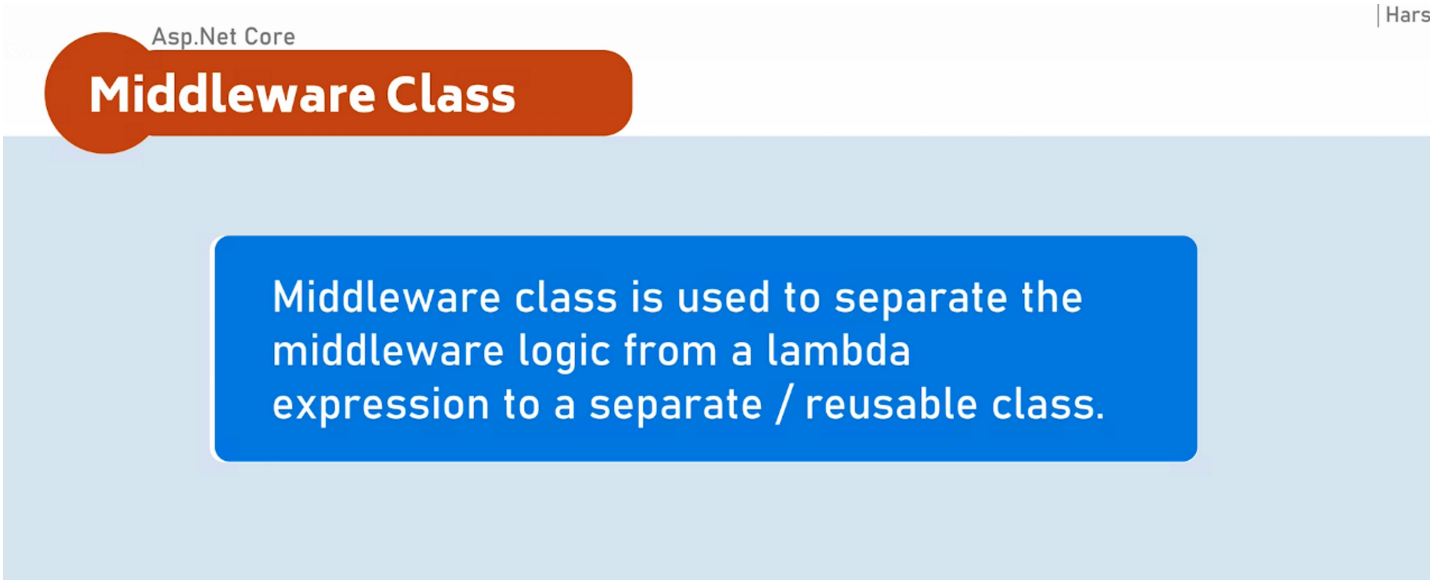


The extension method called "Use" is used to execute a non-terminating / short-circuiting middleware that may / may not forward the request to the next middleware.

Asp.Net Core

# Custom Middleware Class

If your middleware needs to execute a large amount of code with significant responsibility, it's better to separate it into a custom middleware class in a separate file instead of writing it all in the Program.cs file. This approach keeps your code organized and modular.




By default, a middleware class must implement the IMiddleware interface to be recognized as middleware. The IMiddleware interface requires the implementation of the InvokeAsync method, which executes when a request reaches that middleware.

The InvokeAsync method receives two arguments:

Context: Provides access to properties like Request and Response.

Next: Optionally invokes the subsequent middleware.

Middleware is of RequestDelegate type by default. The "before logic" executes before invoking the next middleware, and the "after logic" executes after the subsequent middleware completes and returns to the calling point. This "before and after logic" behavior has been covered in the previous lecture.

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

now, we have to register this custom middleware class as a service. Go to Program.cs file and before calling this 'build()' method here.

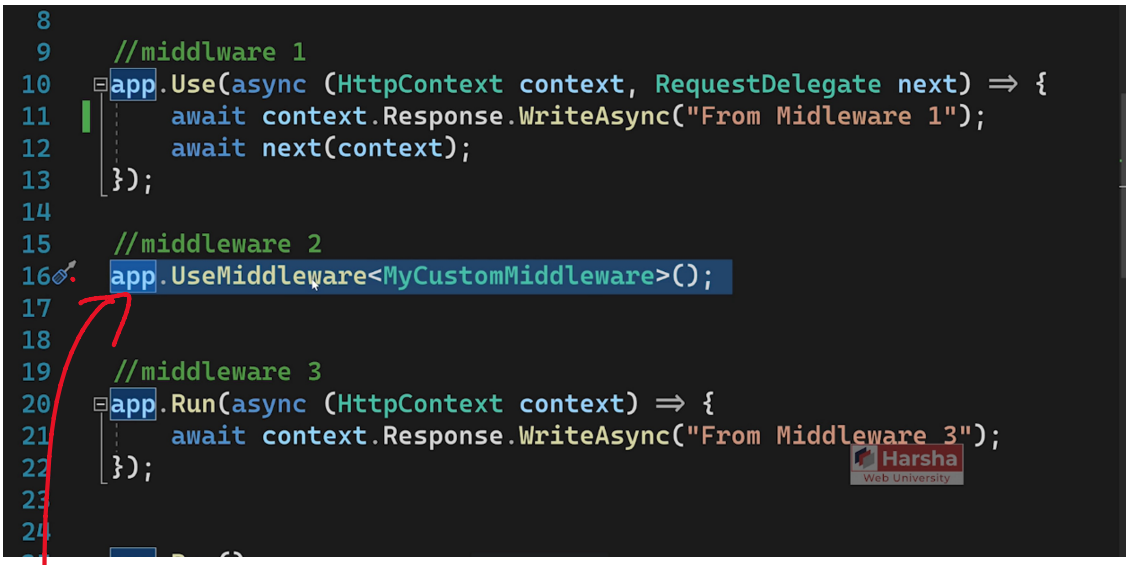
```
using MiddlewareExample.CustomMiddleware;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddTransient<MyCustomMiddleware>();
var app = builder.Build();

//middlware 1
app.Use(async (HttpContext context, RequestDelegate 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);
```

Harsha Web University

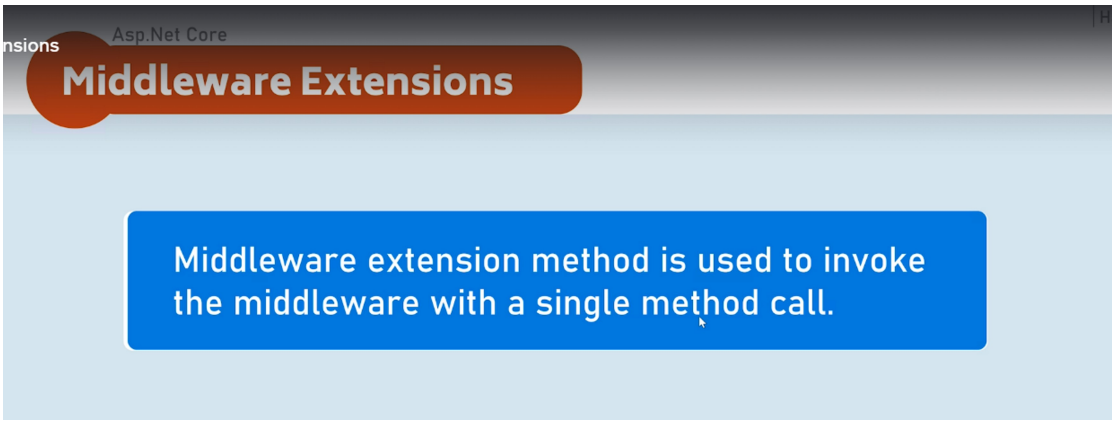
so we have added our custom middleware

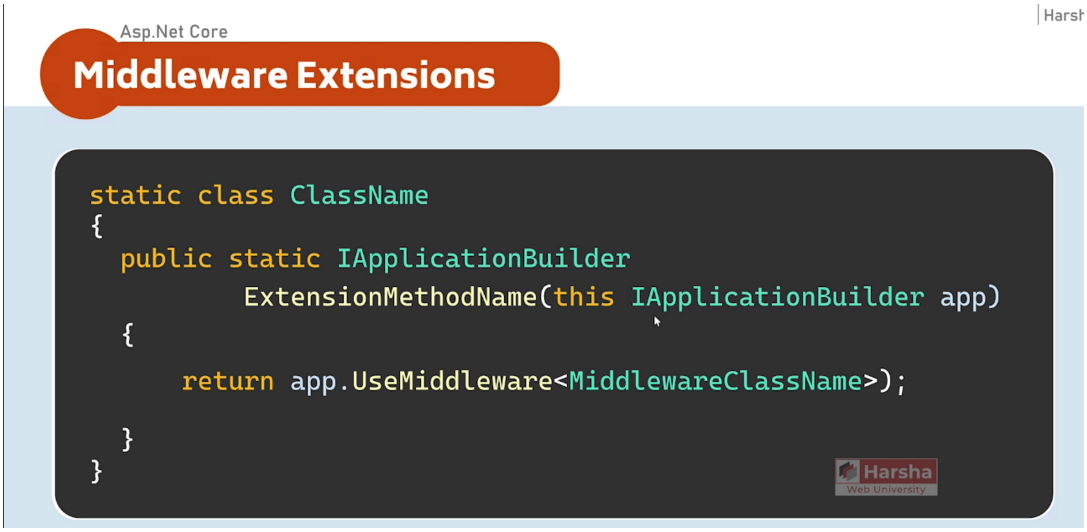
Asp.Net Core

# Custom Middleware Extensions



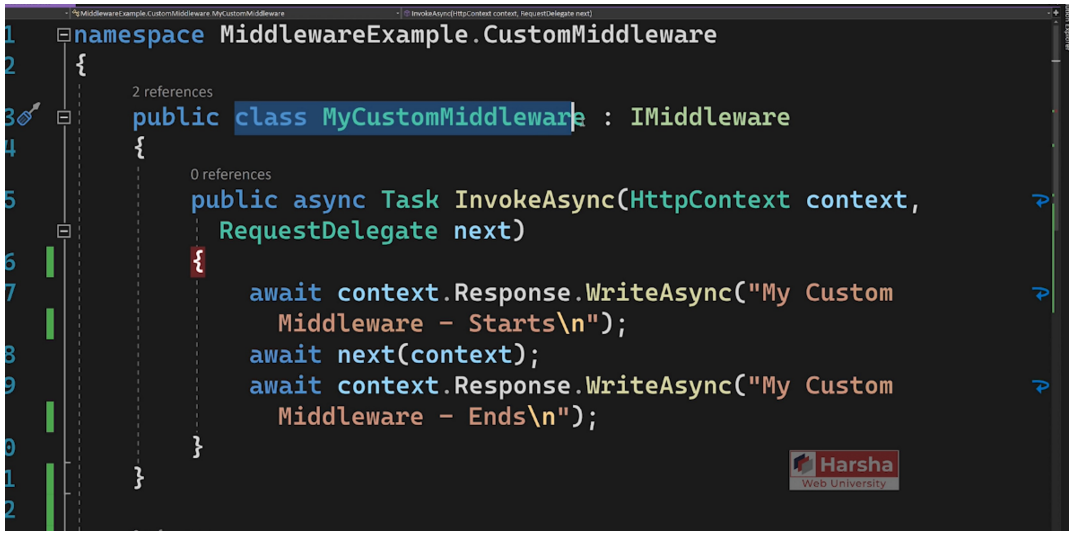

This particular statement can be simplified. If you can create an extension method and write the same statement, you can simply call that extension method by calling this app object.





You can also create custom middleware without inheriting from the IMiddleware interface. A plain class can be converted into middleware by implementing the necessary logic.


namespace MiddlewareExample.CustomMiddleware
{
2 references
public class MyCustomMiddleware : IMiddleware
{
0 references
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");
}
}
Harsha
Web University


Asp.Net Core
Harsha
Conventional Middleware
class MiddlewareClassName
{
private readonly RequestDelegate _next;
public MiddlewareClassName(RequestDelegate next)
{
_next = next;
}
public async Task InvokeAsync(HttpContext context)
{
//before logic
await _next(context);
//after logic
}
});
Harsha
Web University


ocalhost:5154
localhost:5154?firstname=john&lastname=resig
localhost:5154/?firstname=john&lastname=resig
localhost:5154?firstname=john&lastname=resig - Google Search
From Midleware 1

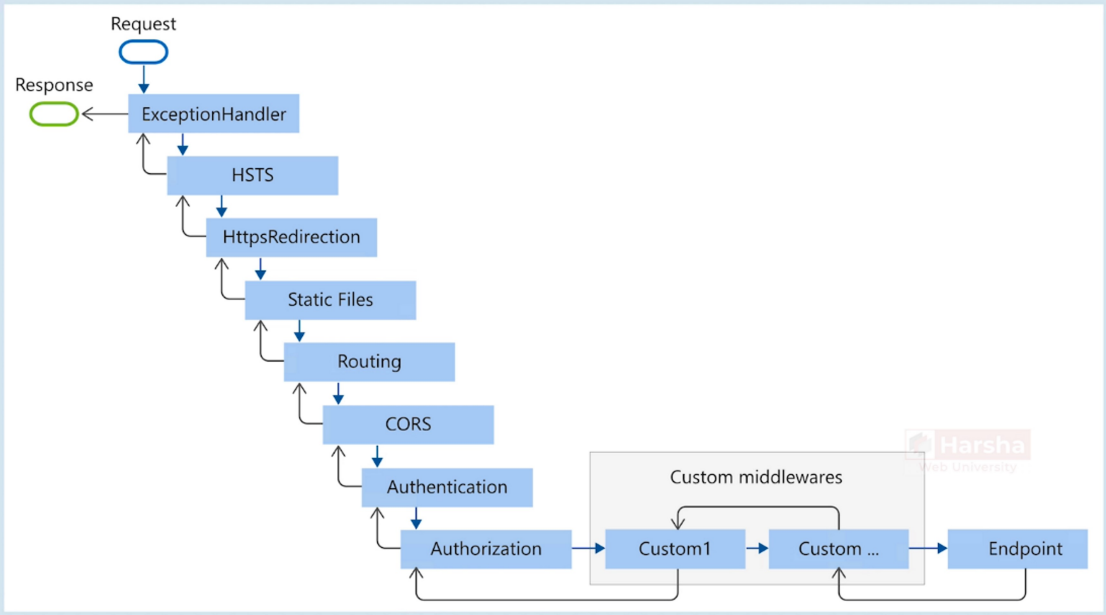
Asp.Net Core

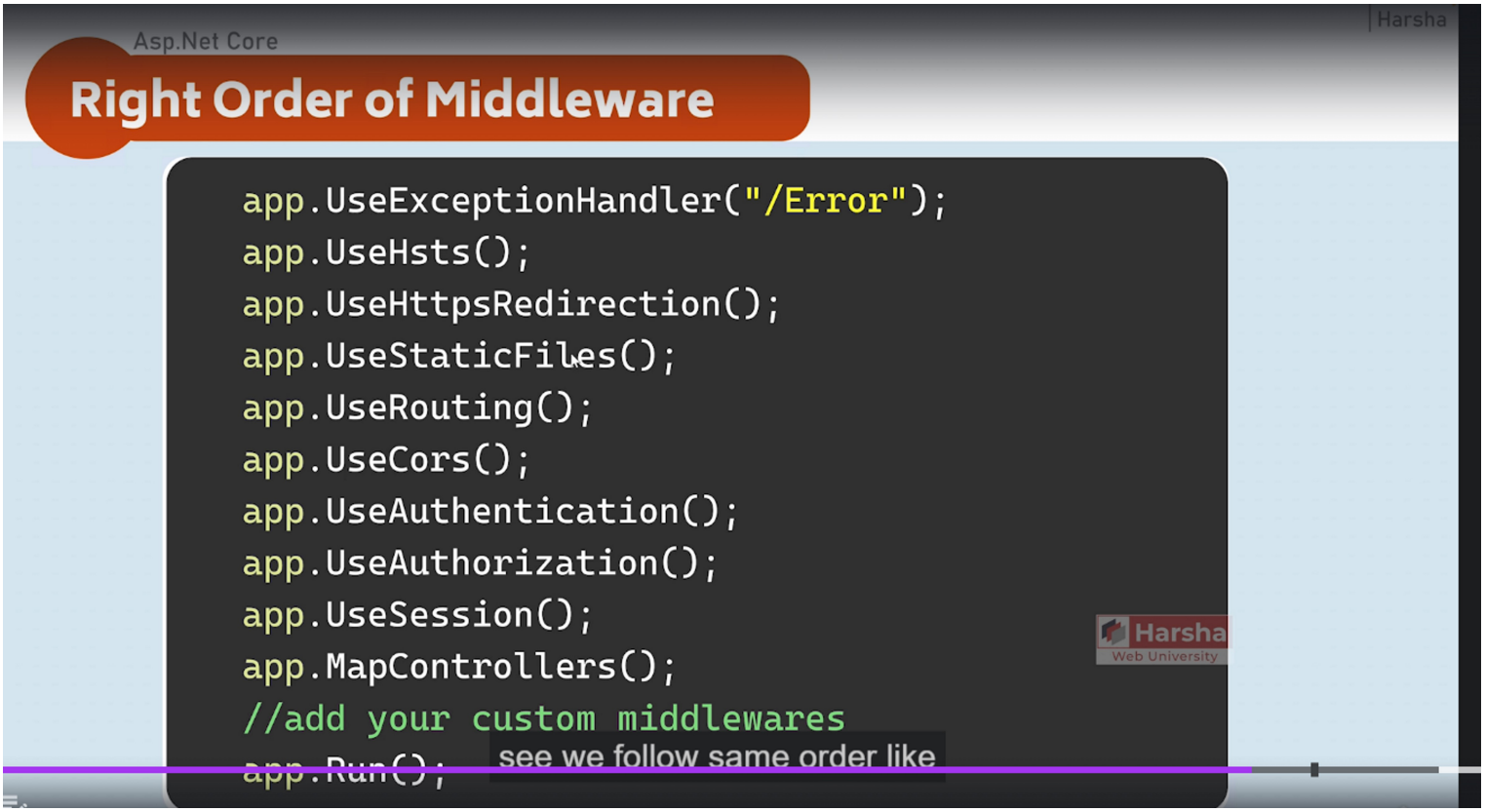
# The Right Order of Middleware



Asp.Net Core

## Right Order of Middleware






Here's the trimmed version:

ASP.NET Core provides many predefined middleware functions, and Microsoft recommends using them in a specific order to ensure proper functionality.

1. **Exception Handler Middleware**: Use in the development environment for handling errors.
2. **HSTS Middleware**: Enforces HTTPS by redirecting HTTP requests.
3. **Static Files Middleware**: Serves static files like `.html` or `.css`.
4. **Routing Middleware**: Handles request routing.
5. **CORS Middleware** (optional): Configures cross-origin resource sharing.
6. **Authentication Middleware**: Enables authentication.
7. **Authorization Middleware**: Controls access based on permissions.
8. **Custom Middleware**: Processes cookies, query strings, or other request data.
9. **Controller Endpoints**: Executes controller logic.

The **recommended order** ensures proper handling of features like HTTPS, routing, and authentication. Misordering middleware (e.g., placing `MapControllers` above HSTS) may lead to unintended behavior, such as failing to enforce HTTPS. Use this order in `Startup.cs` to avoid issues. Concepts like CORS, authentication, and routing will be covered later.

Asp.Net Core

# UseWhen