Middleware in ASP.NET Core

Complete Guide to the Request Pipeline

What is Middleware?

Middleware is software that sits between the web server and your application logic. Each piece of middleware:

- Receives an HTTP request
- Performs some operation (logging, authentication, etc.)
- Either responds immediately OR passes the request to the next middleware
- Can also modify the response on the way back

Think of middleware as a chain of components that process every HTTP request and response.

Middleware in Express vs .NET

Express.js	ASP.NET Core

`app.use((req, res, next) => {`
`  console.log('Request');`
`  next();`
`});`
	`app.Use(async (context, next) =>`
	`{`
	`    // Before`
	`    await next();`
	`    // After`
	`});`

Key difference: .NET middleware can execute code BOTH before and after calling next(), allowing you to wrap the entire request/response cycle.

The Middleware Pipeline

Middleware components are executed in the order they're added to the pipeline:

```
Request → M1 → M2 → M3 → Endpoint → M3 → M2 → M1 → Response
```

Visual representation:

```
┌─────────────────────────────────────────────┐
│  Request                                    │
├─────────────────────────────────────────────┤
│  Middleware 1  (Logging)                    │
│      ↓
│  Middleware 2  (Authentication)             │
│      ↓
│  Middleware 3  (CORS)                    │
│      ↓
│  Endpoint      (Your API handler)        │
│      ↓
│  [Response travels back up through all]  │
├──────────────────────────────────────────┤
│  Response                                │
└──────────────────────────────────────────┘
```

How to Use Middleware

1. Using Built-in Middleware

ASP.NET Core provides many built-in middleware via app.UseXxx() methods:

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

app.UseHttpsRedirection();  // Redirects HTTP → HTTPS
app.UseCors();              // CORS policy
app.UseAuthentication();    // Authentication
app.UseAuthorization();     // Authorization
app.UseStaticFiles();       // Serve static files

app.MapControllers();       // Endpoint routing
```

2. Inline Middleware with app.Use()

Quick middleware for simple tasks:

```
app.Use(async (context, next) =>
{
    // BEFORE the next middleware
    Console.WriteLine($"Request: {context.Request.Path}");

    await next(); // Call the next middleware

    // AFTER the next middleware (on the way back)
    Console.WriteLine($"Response: {context.Response.StatusCode}");
});
```

3. Terminal Middleware with app.Run()

Terminal middleware never calls next() - it ends the pipeline:

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Pipeline ends here");
    // No next() - nothing else runs after this
});
```

⚠️ **Warning:** app.Run() should always be the LAST thing in your pipeline.

4. Conditional Middleware with app.Map()

Branch the pipeline based on the request path:

```
    app.Map("/api", apiApp =>
    {
        // This middleware only runs for /api/* paths
        apiApp.Use(async (context, next) =>
        {
            Console.WriteLine("API middleware");
            await next();
        });
    });
```

5. Custom Middleware Class

For reusable, testable middleware, create a class:

```
    public class RequestLoggingMiddleware
    {
        private readonly RequestDelegate _next;
        private readonly ILogger _logger;

        // Constructor: DI injects next and any services
        public RequestLoggingMiddleware(
            RequestDelegate next,
            ILogger<RequestLoggingMiddleware> logger)
        {
            _next = next;
            _logger = logger;
        }

        // InvokeAsync: the method that runs on each request
        public async Task InvokeAsync(HttpContext context)
        {
            _logger.LogInformation($"Request: {context.Request.Path}");

            await _next(context);
            _logger.LogInformation($"Response:
    {context.Response.StatusCode}");
        }
    }
```

Use it in Program.cs:

```
app.UseMiddleware<RequestLoggingMiddleware>();
```

Middleware Parameters Explained

You're correct! Middleware has TWO key parameters, and the Host provides both:

Parameter 1: HttpContext (context)

The current HTTP request/response:

```
context.Request.Path            // /api/users
context.Request.Method          // GET, POST, etc.
context.Request.Headers         // All headers
context.Request.Query           // Query string
context.Request.Body            // Request body stream

context.Response.StatusCode     // 200, 404, etc.
context.Response.Headers        // Response headers
context.Response.Body           // Response body stream

context.User                    // Authenticated user
context.RequestServices         // DI container
```

Parameter 2: RequestDelegate (next)

A delegate (function pointer) to the next middleware in the pipeline:

```
await next(context);  // OR simply: await next();
```

What happens when you call next():

- Passes control to the next middleware
- Waits (await) for all subsequent middleware to finish
- Returns control back to your middleware
- You can then execute code AFTER the response is generated

If you DON'T call next():

- The pipeline stops
- Your middleware must handle the response
- Nothing else in the pipeline runs

How the Host Injects These Parameters

For inline middleware (app.Use):

```
app.Use(async (context, next) => { /* ... */ });
```

The Host automatically provides both parameters.

For class-based middleware:

```
// Constructor: Host injects 'next' + any DI services
public MyMiddleware(RequestDelegate next, ILogger logger)

// InvokeAsync: Host injects 'context' + scoped services
public async Task InvokeAsync(HttpContext context, IScopedService svc)
```

Key insight: *Constructor gets singleton/scoped services; InvokeAsync gets context + scoped services for THAT request.*

Important Built-in Middleware

Middleware	Purpose	Usage
UseHttpsRedirection	Redirects HTTP to HTTPS	`app.UseHttpsRedirection();`
UseCors	Handles Cross-Origin requests	`app.UseCors("policy");`
UseAuthentication	Authenticates users (JWT, Cookies)	`app.UseAuthentication();`
UseAuthorization	Checks permissions/roles	`app.UseAuthorization();`
UseStaticFiles	Serves static files (CSS, JS, images)	`app.UseStaticFiles();`
UseRouting	Matches requests to endpoints	`app.UseRouting();`
UseEndpoints	Executes matched endpoint	`app.MapControllers();`
UseExceptionHandler	Global exception handling	`app.UseExceptionHandler();`
UseResponseCompression	Compresses responses (gzip, brotli)	`app.UseResponseCompression();`

⚠️ Middleware Order MATTERS!

The order you add middleware determines the order they execute.

Recommended Order

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

// 1. Exception handling (catch everything)
app.UseExceptionHandler("/error");

// 2. HTTPS redirection
app.UseHttpsRedirection();

// 3. Static files (no need for auth)
app.UseStaticFiles();

// 4. Routing (determines which endpoint)
app.UseRouting();

// 5. CORS (after routing, before auth)
app.UseCors();

// 6. Authentication (who are you?)
app.UseAuthentication();

// 7. Authorization (are you allowed?)
app.UseAuthorization();

// 8. Custom middleware
app.UseMiddleware<RequestLoggingMiddleware>();

// 9. Endpoints (LAST - execute the handler)
app.MapControllers();
```

Why this order?

- **Exception handler first:** Catches errors from ALL other middleware
- **Static files before routing:** No need to authenticate for CSS/JS
- **CORS after routing:** Needs to know which endpoint matched
- **Authentication before authorization:** Must identify user before checking permissions
- **Endpoints last:** Only run after all checks pass

Practical Middleware Scenarios

Scenario 1: Request Timing

Measure how long each request takes:

```
app.Use(async (context, next) =>
{
    var watch = System.Diagnostics.Stopwatch.StartNew();

    await next();

    watch.Stop();
    Console.WriteLine($"Request took {watch.ElapsedMilliseconds}ms");
});
```

Scenario 2: API Key Authentication

```
app.Use(async (context, next) =>
{
    var apiKey = context.Request.Headers["X-API-
Key"].FirstOrDefault();

    if (string.IsNullOrEmpty(apiKey) || !IsValidApiKey(apiKey))
    {
        context.Response.StatusCode = 401;
        await context.Response.WriteAsync("Unauthorized");
        return; // Stop pipeline
    }

    await next();
});
```

Scenario 3: Request ID for Tracing

```
app.Use(async (context, next) =>
{
    var requestId = Guid.NewGuid().ToString();
    context.Items["RequestId"] = requestId;

    await next();

    context.Response.Headers["X-Request-ID"] = requestId;
});
```

Scenario 4: Global Error Handling

```
app.Use(async (context, next) =>
{
    try
    {
        await next();
    }
    catch (Exception ex)
    {
        context.Response.StatusCode = 500;
        await context.Response.WriteAsJsonAsync(new
        {
            error = ex.Message,
            timestamp = DateTime.UtcNow
        });
    }
});
```

Scenario 5: Rate Limiting

```
public class RateLimitingMiddleware
{
    private readonly RequestDelegate _next;
    private static Dictionary<string, DateTime> _requests = new();

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

    public async Task InvokeAsync(HttpContext context)
    {
        var ip = context.Connection.RemoteIpAddress?.ToString();

        if (_requests.TryGetValue(ip, out var lastRequest))
        {
            if ((DateTime.UtcNow - lastRequest).TotalSeconds < 1)
            {
                context.Response.StatusCode = 429; // Too Many
Requests
                return;
            }
        }

        _requests[ip] = DateTime.UtcNow;
        await _next(context);
    }
}
```

Advanced Middleware Concepts

Accessing Scoped Services in Middleware

You can inject scoped services into InvokeAsync:

```
    public class MyMiddleware
    {
        private readonly RequestDelegate _next;

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

        // Inject scoped services here!
        public async Task InvokeAsync(
            HttpContext context,
            DbContext db,              // Scoped
            ILogger<MyMiddleware> logger) // Singleton
        {
            // Use db and logger
            await _next(context);
        }
    }
```

Short-Circuiting the Pipeline

Sometimes you want to stop the pipeline early:

```
    app.Use(async (context, next) =>
    {
        if (context.Request.Path == "/health")
        {
            context.Response.StatusCode = 200;
            await context.Response.WriteAsync("OK");
            return; // Don't call next()
        }

        await next();
    });
```

Conditional Middleware with UseWhen()

```
    app.UseWhen(
        context => context.Request.Path.StartsWithSegments("/api"),
        appBuilder =>
        {
            // Only runs for /api/* requests
            appBuilder.Use(async (context, next) =>
            {
                Console.WriteLine("API request");
                await next();
            });
        });
```

Summary: Key Takeaways

1. **Middleware = chain of components** processing every HTTP request
2. **Two parameters:** HttpContext (request/response) and RequestDelegate (next)
3. **Host injects both:** You don't create them manually
4. **Order matters:** Exception handling → Static files → Routing → Auth → Endpoints
5. **Three ways to add:** Built-in (UseXxx), inline (Use), class-based (UseMiddleware)
6. **Can execute before AND after:** Code before next() and after next()
7. **Short-circuit when needed:** Don't call next() to stop the pipeline

Now you understand the middleware pipeline!