

◆ SECTION 4: MIDDLEWARE & REQUEST PIPELINE ◆

1 What is Middleware?

💡 Think of middleware like security checks at an airport ✈️

Before you board the plane:

- Security checks passport
- Another checks luggage
- Another checks ticket

Similarly, **every HTTP request passes through multiple middleware** before reaching your controller.

Each middleware:

- Can **inspect the request**
- Can **modify the response**
- Can **stop the request**

🔗 Middleware in ASP.NET Core is a **component in the HTTP request pipeline** that:

- Handles incoming requests
- Performs cross-cutting concerns
- Either passes control to the next middleware or short-circuits the pipeline

Each middleware:

- Receives `HttpContext`
- Executes **before and/or after** the next component

📌 Controllers are executed **only after all required middleware** run successfully.

📖 Q: Is middleware framework-level or application-level?

✓ Application-level, configured in `Program.cs`

📖 Q: Can middleware modify response?

✓ Yes, both request and response

2 What is the Request Processing Pipeline?

💡 Request pipeline is the **order** in which middleware runs.

Order matters:

- Wrong order → app breaks
- Correct order → app works

🎯 The ASP.NET Core pipeline is a **linear chain of middleware** executed in the order they are registered.

```
Request → Middleware 1 → Middleware 2 → ... → Controller  
Controller → Middleware ... → Response
```

Middleware can:

- Execute logic **before**
- Call `next()`
- Execute logic **after**

📌 This is called the **onion model**.

🔄 Q: Is pipeline **bidirectional**?

✓ Yes (request goes forward, response comes back)

3 Built-in Middleware (INTERVIEW MUST)

◆ A. Exception Handling Middleware

🧠 Catches errors so app doesn't crash.

🎯 Interview

- Handles unhandled exceptions
- Converts them to proper HTTP responses
- Prevents stack trace leaks

📌 Example:

```
• UseExceptionHandler  
• UseDeveloperExceptionPage
```

🔄 Q: Why not use try-catch everywhere?

✓ Centralized, cleaner, consistent error handling

◆ B. HTTPS Redirection Middleware

🧠 Forces HTTP → HTTPS.

🎯 Interview

- Improves security
- Prevents man-in-the-middle attacks

◆ C. Routing Middleware

🧠 Finds **which controller/action** to call.

🎯 Interview

- Matches URL to endpoint
- Stores routing metadata in `HttpContext`

◆ D. Authentication Middleware

🧠 Checks **who you are**.

🎯 Interview

- Validates credentials (JWT, cookies)
- Builds `ClaimsPrincipal`

◆ E. Authorization Middleware

🧠 Checks **what you are allowed to do**.

🎯 Interview

- Enforces policies and roles
- Runs **after authentication**

🚨 Q: Can authorization work without authentication?

❌ No; ✅ Authentication must run first

🔑 Order of Middleware (VERY IMPORTANT)

Correct Order (Interview Gold)

- `UseExceptionHandler`
- `UseHttpsRedirection`
- `UseRouting`
- `UseAuthentication`
- `UseAuthorization`
- `UseEndpoints`

🎯 Interview Explanation

- Routing must happen before auth
- Authentication before authorization
- Authorization before endpoint execution

🚨 Wrong order → 401 / 403 / routing issues

🔑 Q: What happens if `UseAuthorization` is before `UseAuthentication`?

✅ User is unauthenticated → authorization fails

5 Custom Middleware

🧠 Custom middleware is **your own security gate**.

🎯 Custom middleware is used for:

- Logging
- Correlation IDs
- Header validation
- Request/response auditing

It:

- Implements a constructor
- Accepts `RequestDelegate`
- Uses `Invoke` or `InvokeAsync`

📌 Executes for **every request** unless filtered.

💡 Q: When to use **middleware vs filters**?

- ✓ Middleware → cross-application
- ✓ Filters → MVC-specific

6 Middleware vs Filters (COMMON CONFUSION)

Aspect	Middleware	Filters
Scope	Global	MVC only
Executes	Before routing / after	Around action
Access to <code>ModelState</code>	✗	✓
Performance	Faster	Slightly slower

📌 “Middleware handles cross-cutting concerns at pipeline level, filters handle concerns at MVC execution level.”

7 Short-Circuiting the Pipeline

🧠 Middleware can stop request early.

🎯 Example scenarios:

- Invalid API key
- Blocked IP
- Maintenance mode

If middleware doesn't call `next()`, pipeline stops.

💡 Q: Is short-circuiting bad?

- ✓ No, it improves performance and security when used correctly

Performance Impact of Middleware

Interview Explanation

- Every middleware adds processing
- Heavy logic impacts latency
- Ordering affects performance

Best practices:

- Keep middleware lightweight
- Avoid unnecessary middleware
- Place frequently-used middleware early

ONE-LINE INTERVIEW ANSWER (MEMORIZE)

“Middleware in ASP.NET Core forms a request pipeline where each component processes the HTTP request and response in order, enabling cross-cutting concerns like routing, authentication, authorization, and error handling.”

HANDS-ON 4: MIDDLEWARE & REQUEST PIPELINE

STEP 0: Project Setup

```
dotnet new webapi -n MiddlewareDemo
```


Structure:

```
MiddlewareDemo
|— Controllers
|— Program.cs
```

STEP 1: Understand Default Pipeline (Before Writing Code)

Open `Program.cs` (default template):

```
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
```

 Even default templates use middleware. Controllers are executed only after middleware.

STEP 2: Add Routing & Authentication Correctly

Update pipeline:

```
app.UseExceptionHandler("/error");
app.UseHttpsRedirection();

app.UseRouting();


app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();
```

 Interview must-know:

- `UseRouting` → selects endpoint
- `UseAuthentication` → builds user identity
- `UseAuthorization` → enforces access rules

STEP 3: Create a Custom Logging Middleware

 Create folder: `Middleware`

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

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

    public async Task InvokeAsync(HttpContext context)
    {
        Console.WriteLine($"Incoming Request: {context.Request.Method}
{context.Request.Path}");

        await _next(context); // Pass control

        Console.WriteLine($"Outgoing Response:
{context.Response.StatusCode}");
    }
}
```

Interview Explanation

- `RequestDelegate` → next middleware
- `InvokeAsync` → executed per request
- Code before `_next` → request
- Code after `_next` → response

STEP 4: Register Custom Middleware

In `Program.cs`:

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

 Interview line:

“Middleware registration order defines execution order.”

STEP 5: Test Middleware Execution Order

Pipeline now:

```
ExceptionHandler
HTTPS
Routing
Custom Logging
Authentication
Authorization
Controller
```

Call any endpoint

```
Incoming Request: GET /weatherforecast
Outgoing Response: 200
```

📌 Interview clarity: **Logging middleware wraps controller execution**

STEP 6: Short-Circuiting Middleware (IMPORTANT)

Create API Key Middleware

```
public class ApiKeyMiddleware
{
    private readonly RequestDelegate _next;
    private const string API_KEY = "x-api-key";
    public ApiKeyMiddleware(RequestDelegate next)
    {
        _next = next;
    }
    public async Task InvokeAsync(HttpContext context)
    {
        if (!context.Request.Headers.ContainsKey(API_KEY))
        {
            context.Response.StatusCode = StatusCodes.Status401Unauthorized;
            await context.Response.WriteAsync("API Key missing");
            return; // SHORT-CIRCUIT
        }
        await _next(context);
    }
}
```

Register it BEFORE controllers: `app.UseMiddleware<ApiKeyMiddleware>();`

📌 Middleware can block unauthorized requests before they reach controllers.

STEP 7: Middleware vs Filters (Hands-on Comparison)

Add Action Filter

```
public class SampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        Console.WriteLine("Action executing");
    }
    public void OnActionExecuted(ActionExecutedContext context)
    {
        Console.WriteLine("Action executed");
    }
}
```

Register:

```
builder.Services.AddControllers(options =>
{
    options.Filters.Add<SampleActionFilter>();
});
```

📌 Interview comparison:

- **Middleware runs before routing**
- **Filters run after routing**

STEP 8: Exception Handling Middleware

Create error controller

```
[ApiController]
[Route("error")]
public class ErrorController : ControllerBase
{
    public IActionResult HandleError()
    {
        return Problem("Something went wrong");
    }
}
```

Trigger exception in controller

```
throw new Exception("Test exception");
```

📌 Interview insight: **Centralized exception handling avoids leaking stack traces.**

STEP 9: Visualize the Pipeline (INTERVIEW FAVORITE)

Draw this mentally:

```
Request
↓
Exception Middleware
↓
HTTPS
↓
Routing
↓
Custom Middleware
↓
Authentication
↓
Authorization
↓
Controller
↓
Authorization
↓
Authentication
↓
Custom Middleware
↓
Response
```

 Interview tip: **Saying “onion model” gives bonus points.**

STEP 10: Common Interview Traps (WITH ANSWERS)

- ? Can middleware access `ModelState`?
- ✗ No; ✓ Filters can
- ? Can middleware access route data?
- ✓ Yes, after `UseRouting`
- ? Should heavy DB calls be in middleware?
- ✗ No, hurts performance

INTERVIEW SUMMARY (MEMORIZE)

“ASP.NET Core middleware forms a request pipeline where each component processes the request and response in order. Custom middleware is used for cross-cutting concerns like logging, security, and auditing, while ordering is critical to correct behavior.”