In a real-world ASP.NET Core Web API, you can use middleware to handle **logging** and **error handling** globally. This ensures all requests are logged and errors are caught and processed in one central place, rather than duplicating this logic in multiple controllers.

---

1. Logging Middleware

The purpose of a logging middleware is to log details about incoming requests and outgoing responses. For example:

- Log request information (HTTP method, URL, headers).
- Log response details (status code, response time).

Implementation: Logging Middleware

Create a custom middleware for logging:

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

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

        public async Task InvokeAsync(HttpContext context)
        {
        // Log the incoming request
        Console.WriteLine($"Request: {context.Request.Method} {context.Request.Path}");

        // Call the next middleware in the pipeline
        await _next(context);

        // Log the outgoing response
        Console.WriteLine($"Response: {context.Response.StatusCode}");
        }
}
```

Register the Middleware

Add the middleware in the `Program.cs` file:

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

// Add Logging Middleware
app.UseMiddleware<LoggingMiddleware>();

app.MapControllers();
app.Run();
```

Output Example

For a request to `GET /api/products`:

```
Request: GET /api/products
Response: 200
```

2. Error Handling Middleware

The purpose of error handling middleware is to catch unhandled exceptions, log them, and return a user-friendly error message.

Implementation: Error Handling Middleware

Create a custom middleware for error handling:

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

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

        public async Task InvokeAsync(HttpContext context)
        {
        try
        {
        await _next(context); // Pass to the next middleware
        }
        catch (Exception ex)
        {
        // Log the exception
        Console.WriteLine($"An error occurred: {ex.Message}");

        // Return a generic error response
        context.Response.StatusCode = 500; // Internal Server Error
        context.Response.ContentType = "application/json";

        var errorResponse = new
        {
                StatusCode = 500,
                Message = "An unexpected error occurred. Please try again later."
        };

        await context.Response.WriteAsJsonAsync(errorResponse);
        }
        }
}
```

Register the Middleware

Add it in `Program.cs`:

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

// Add Error Handling Middleware
app.UseMiddleware<ErrorHandlingMiddleware>();

app.MapControllers();
app.Run();
```

Output Example

For a request that causes an exception in the application:

```
An error occurred: Object reference not set to an instance of an object.
```

Response sent to the client:

```
{
	"StatusCode": 500,
	"Message": "An unexpected error occurred. Please try again later."
}
```

3. Combining Logging and Error Handling

You can combine logging and error handling middleware in the pipeline to handle both tasks:

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

// Add Logging Middleware
app.UseMiddleware<LoggingMiddleware>();

// Add Error Handling Middleware
app.UseMiddleware<ErrorHandlingMiddleware>();

app.MapControllers();
app.Run();
```

How It Works

1. **Logging Middleware** logs the request and passes it to the next middleware.
2. **Error Handling Middleware** catches any exceptions thrown during the request processing.
3. If no exceptions occur, the response is logged by the **Logging Middleware**.