

INFORMAÇÕES		
MÓDULO	AULA	INSTRUTOR
06 ASP.NET Core Web API – Avançado e Segurança Inicial	07 - Resiliência e Observabilidade: Middlewares, Serilog, Exception Handling	Guilherme Paracatu

1. O Conceito de Pipeline em Middlewares

1.1 O que é um Middleware?

Imagine a sua API como um restaurante. O **Middleware** é o funcionário que fica no corredor entre a mesa do cliente e a cozinha.

- **A função:** Ele pode interceptar o pedido (Request), verificar se o cliente está de terno (Autenticação), anotar o horário (Logging) ou até impedir que o pedido chegue à cozinha se algo estiver errado (Exception Handling).
- **O fluxo:** No .NET, os middlewares formam um **Pipeline**. A requisição passa por todos eles na ida e a resposta passa por todos eles na volta.

1.2 Como funciona?

Imagine a requisição do seu usuário como uma caixa em uma **esteira de produção** (o Pipeline).

- **O Fluxo Bidirecional:** O Pipeline no .NET não é uma rua de mão única. A requisição entra, passa por vários "postos de inspeção" (Middlewares) até chegar ao Controller. Depois que o Controller responde, a resposta **volta** pela mesma esteira, passando novamente pelos Middlewares antes de sair para o usuário.
- **A "Ponte" (_next):** Cada Middleware tem a responsabilidade de decidir: *"Eu vou processar algo e passar para o próximo da fila ou vou parar tudo aqui?"*. Quando chamamos `await _next(context)`, estamos passando a caixa para o próximo funcionário da esteira.
- **A Ordem Importa:** Se você colocar o Middleware de Log depois do Middleware de Autenticação, e a autenticação falhar, o log nem chegará a ser registrado. Por isso, a ordem no Program.cs é vital.

1.3 Serilog: Por que não usar apenas o Console?

O `Console.WriteLine` é volátil. Se o console fechar, a informação some. O **Serilog** é um logger estruturado. Ele transforma o log em dados.

- **Logging Estruturado:** Em vez de salvar "O usuário 10 comprou o item 5", ele salva um objeto JSON onde você pode pesquisar especificamente por `UsuarioId == 10`.
- **Sinks:** São os destinos. O Serilog pode enviar o mesmo log para o Console, para um arquivo `.txt`, para um banco de dados ou para ferramentas de monitoramento em tempo real.

No desenvolvimento básico, usamos `Console.WriteLine("Erro tal")`. No mundo real, isso é inútil para sistemas com milhares de acessos. O **Serilog** introduz o **Logging Estruturado**.

- **Texto vs. Dados:** Um log comum é apenas uma frase. O Serilog grava **objetos**. Em vez de salvar "O produto 5 foi criado", ele salva um registro onde o campo `ProdutoId` é igual a 5. Isso permite que você faça buscas em ferramentas de monitoramento como se estivesse consultando um banco de dados.
- **Sinks (Destinos):** O Serilog usa o conceito de "Sinks". Você configura um único código de log e ele "escoa" (sink) para vários lugares ao mesmo tempo: console, arquivo local, banco de dados ou serviços na nuvem (como Azure Application Insights).
- **Enrichers (Enriquecedores):** Você pode configurar o Serilog para "enriquecer" cada linha de log automaticamente com o nome da máquina, a versão do sistema ou o ID da requisição, sem precisar digitar isso toda vez.

1.4 Open Generics: O "Curinga" da Injeção de Dependência

Até agora, registramos um Handler para cada Command. Mas e se quisermos algo que monitore **todos** os comandos? Usamos **Open Generics** (`typeof(IHandler<>)`). É uma forma de dizer ao .NET: "Não importa qual seja o comando, se ele usar essa interface, aplique esta regra".

Este é um dos conceitos mais poderosos do C#.

- **O que é:** Normalmente, você registra uma interface específica para uma classe específica: `services.AddScoped<IHandler<CriarFabricanteCommand>, CriarFabricanteHandler>()`. Isso é um "Tipo Fechado".
- **O "Tipo Aberto":** Quando usamos `typeof(IHandler<>)`, deixamos os sinais de menor e maior **vazios**. Isso diz ao .NET: "Eu não sei qual Comando virá, mas para qualquer um que implemente `IHandler`, eu quero que você aplique esta regra".

- **Analogia do Adaptador Universal:** Imagine que você tem várias tomadas diferentes (Commands). Em vez de comprar um adaptador para cada uma, você cria um **Adaptador Universal (Open Generic)** que se molda a qualquer tomada que chegar.
- **No nosso caso:** Usamos isso para o Log. Não importa se é um comando de Fabricante, Produto ou Usuário; o `LogComandoGenericoHandler<T>` vai "abraçar" qualquer um deles automaticamente.

Exemplos no Dia-a-Dia do Desenvolvedor

- **Rastreamento de Erros (O "Onde quebrou?"):** Sabe aquele erro que só acontece no servidor de produção e nunca na sua máquina? Com o Serilog gravando em arquivo, você abre o log e vê exatamente os dados que o usuário enviou e qual linha do código falhou.
- **Auditoria e Segurança:** Imagine que um cliente alega que não deletou um registro. Com o **Log Genérico**, você tem uma prova técnica: "No dia X, às 10h, o comando `DeletarPedidoCommand` foi disparado pelo IP tal".
- **Experiência do Usuário (UX):** Em vez de o usuário ver aquela página branca feia de erro do navegador, o seu **Middleware de Exception** garante que ele receba uma mensagem educada em JSON, mantendo o padrão da sua API mesmo em momentos de crise.

2. Implementação Técnica

Preparação do Ambiente (Terminal do VS Code)

Antes de iniciar a codificação, precisamos instalar os pacotes do **Serilog** em nossos projetos. Abra o terminal na pasta raiz da solução e execute os seguintes comandos:

1. Instalação no projeto da API (ITB.API):

Este pacote contém a integração principal com o ASP.NET Core.

```
dotnet add ITB.API package Serilog.AspNetCore
```

2. Instalação no projeto de Infraestrutura/Configuração (ITB.IoC):

Precisamos destes pacotes para que a nossa classe de extensão reconheça os comandos do Serilog e consiga gravar em arquivos.

```
dotnet add ITB.IoC package Serilog.AspNetCore
```

```
dotnet add ITB.IoC package Serilog.Sinks.File
```

```
dotnet add ITB.IoC package Serilog.Settings.Configuration
```

A) Configuração de Infraestrutura e Log (ITB.IoC)

Centralizamos tudo aqui para manter o Program.cs minimalista e organizado.

Código C#

// Local: ITB.IoC/DependencyInjection.cs

using Microsoft.Extensions.Hosting; // Necessário para configurar o Host do Serilog
using Serilog; // Biblioteca de Logging Profissional

```
public static class DependencyInjection
{
```

```
    public static IServiceCollection AddInfrastructure(this IServiceCollection
services, IConfiguration configuration)
```

```
    {
```

```
        // 1. Configuração do Banco de Dados
```

```
        var connectionString = configuration.GetConnectionString("DefaultConnection");
```

```
        services.AddDbContext<AppDbContext>(options =>
```

```
options.UseNpgsql(connectionString));
```

```
        // 2. Registro de Repositórios e Serviços Tradicionais
```

```
        services.AddScoped<IFabricanteRepository, FabricanteRepository>();
```

```
        services.AddScoped<IFabricanteService, FabricanteService>();
```

```
        // 3. Arquitetura Mediator e CQRS
```

```
        services.AddScoped<IMessageBus, InMemoryBus>();
```

```
        // 3.1 Handlers Específicos (Um para cada ação)
```

```
        services.AddScoped<IHandler<CriarFabricanteCommand>, CriarFabricanteHandler>();
```

```
        // 3.2 Log Genérico (Open Generics) - O "Curinga"
```

```
        // Esta linha faz com que TODO comando dispare um log automaticamente
```

```
        services.AddScoped(typeof(IHandler<>), typeof(LogComandoGenericoHandler<>));
```

```
        // 4. AutoMapper
```

```
        services.AddAutoMapper(typeof(MappingProfile).Assembly);
```

```
        return services;
```

```
    }
```

```
// Extensão para configurar o Serilog sem poluir o Program.cs
```

```
public static void AddSerilogApi(this IHostBuilder host)
```

```
{
```

```
    host.UseSerilog((context, configuration) =>
```

```
    {
```

```
        configuration
```

```
            .WriteTo.Console() // Exibe no terminal do VS Code
```

```
            .WriteTo.File("logs/log-.txt", rollingInterval: RollingInterval.Day) //
```

Arquivo diário

```
            .ReadFrom.Configuration(context.Configuration); // Lê ajustes do
```

appsettings.json

```

    });
}
}

```

B) O Log Genérico (ITB.Application)

Este código roda para qualquer comando, graças ao registro que fizemos acima.

Código C#

// Local: ITB.Application.Handlers/LogComandoGenericoHandler.cs

```

public class LogComandoGenericoHandler<T> : IHandler<T> where T : ICommand
{
    private readonly ILogger<LogComandoGenericoHandler<T>> _logger;

    public LogComandoGenericoHandler(ILogger<LogComandoGenericoHandler<T>> logger)
    {
        _logger = logger; // Injeção do Logger configurado (Serilog)
    }

    public async Task Handle(T comando)
    {
        // {@Data} serializa o objeto inteiro como um JSON no Log (muito útil!)
        _logger.LogInformation("Executando Comando: {CommandName} | Dados Enviados:
{@Data}",
            typeof(T).Name,
            comando);

        await Task.CompletedTask;
    }
}

```

C) Middleware de Exception Global (ITB.API)

A "rede de segurança" da nossa aplicação.

Código C#

// Local: ITB.API/Middleware/ExceptionMiddleware.cs

```

public class ExceptionMiddleware
{
    private readonly RequestDelegate _next; // Próximo passo no pipeline
    private readonly ILogger<ExceptionMiddleware> _logger;

    public ExceptionMiddleware(RequestDelegate next, ILogger<ExceptionMiddleware>
logger)
    {
        _next = next;
        _logger = logger;
    }
}

```

```

public async Task InvokeAsync(HttpContext context)
{
    try {
        await _next(context); // Tenta seguir com a requisição
    }
    catch (Exception ex) {
        _logger.LogError(ex, "ERRO NÃO TRATADO DETECTADO!"); // Registra no Serilog
        await HandleExceptionAsync(context, ex); // Responde ao usuário
    }
}

private static Task HandleExceptionAsync(HttpContext context, Exception exception)
{
    context.Response.ContentType = "application/json";
    context.Response.StatusCode = 500;

    var response = new {
        Error = "Ops! Algo deu errado internamente.",
        Message = exception.Message,
        Timestamp = DateTime.Now
    };

    // Use o JsonSerializer para converter o objeto em uma string JSON real
    return context.Response.WriteAsync(JsonSerializer.Serialize(response));
}
}

```

D) O "Novo" Program.cs (Limpo e Profissional)

Com as extensões que criamos, o arquivo principal da sua API fica assim:

Código C#

// Local: ITB.API/program.cs

```

var builder = WebApplication.CreateBuilder(args);

// 1. Configura o Serilog (via nossa extensão no IoC)
builder.Host.AddSerilogApi();

// 2. Configura a Infraestrutura (Banco, Repos, Handlers, Bus)
builder.Services.AddInfrastructure(builder.Configuration);

builder.Services.AddControllers();

var app = builder.Build();

// 3. Ativa a nossa Rede de Segurança (Middleware)
app.UseMiddleware<ExceptionMiddleware>();

app.UseAuthorization();

```

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

E) Configuração dos Logs

Serve para o log não ficar "poluído" com mensagens internas do próprio Windows/Entity Framework.

Código C#

```
// Local: ITB.API/appsettings.json
```

```
{  
  "Serilog": {  
    "MinimumLevel": {  
      "Default": "Information",  
      "Override": {  
        "Microsoft": "Warning",  
        "Microsoft.AspNetCore": "Warning",  
        "System": "Warning",  
        "Microsoft.EntityFrameworkCore": "Fatal"  
      }  
    }  
  }  
}
```

```
// Local: ITB.API/appsettings.Development.json
```

```
{  
  "ConnectionStrings": {  
    "DefaultConnection":  
"Host=localhost;Port=5432;Database=itb_mig;Username=postgres;Password=admin"  
  },  
  "Serilog": {  
    "MinimumLevel": {  
      "Default": "Information",  
      "Override": {  
        "Microsoft": "Warning",  
        "Microsoft.AspNetCore": "Warning",  
        "System": "Warning",  
        "Microsoft.EntityFrameworkCore": "Fatal"  
      }  
    }  
  },  
  "JwtSettings": {  
    "SecretKey": "chave-secreta-muito-segura-e-longa-123456",  
    "Issuer": "ITB.API",  
    "Audience": "ITB.Client"  
  }  
}
```

Exercício Prático de Fixação

1. **Forçar Erro:** Vá em qualquer Handler e adicione a linha :
`throw new Exception("Falha Crítica de Teste");`
2. **Verificar Resposta:** Rode a API e veja se o JSON de erro aparece formatado no Swagger.
3. **Verificar Arquivo:** Abra a pasta /logs no seu projeto e verifique se o Serilog gravou o erro e os dados do comando enviado.

Glossário de Erros Comuns: O "Kit de Sobrevivência" do Desenvolvedor

Muitas vezes, o código está "certo", mas a aplicação não se comporta como esperado. Aqui estão os tropeços mais frequentes e como sair deles:

1. Logs e Serilog

Sintoma	Causa Provável	Solução
A pasta /logs não é criada.	Falta de permissão de escrita ou caminho relativo inválido.	Verifique se o VS Code tem permissão na pasta. Tente rodar o projeto como Administrador uma vez ou verifique se o caminho no <code>WriteToFile</code> está correto.
O console mostra logs, mas o arquivo .txt está vazio.	O Serilog não foi "fechado" corretamente ou o arquivo está travado por outro processo.	Certifique-se de que o <code>builder.Host.AddSerilogApi()</code> está no início do <code>Program.cs</code> . Reinicie a aplicação.
Logs internos do EF Core poluindo o arquivo.	Nível de log padrão muito baixo (Information ou Debug).	No <code>appsettings.json</code> , adicione o override: <code>"Microsoft.EntityFrameworkCore": "Warning"</code> .

2. Middlewares e Pipeline

Sintoma	Causa Provável	Solução
A exceção estoura no console, mas o JSON não volta	O Middleware de Exception foi registrado depois do <code>app.MapControllers()</code> .	O <code>app.UseMiddleware<ExceptionHandler>()</code> deve ser uma das primeiras linhas após o <code>builder.Build()</code> .

no Swagger.		
Erro 500 sem mensagem nenhuma.	Erro de sintaxe dentro do próprio <code>ExceptionHandlerMiddleware</code> .	Verifique se você usou o <code>JsonSerializer.Serialize</code> e se o <code>context.Response.ContentType</code> está como <code>application/json</code> .
O log genérico não registra o erro.	O <code>try/catch</code> do <code>Middleware</code> capturou o erro antes do log genérico ser disparado.	Isso é normal! O <code>Middleware</code> é a última linha de defesa. Se o erro chegou nele, o log deve ser feito pelo próprio <code>Middleware</code> usando <code>_logger.LogError</code> .