

INFORMAÇÕES

MÓDULO	AULA	INSTRUTOR
06 ASP.NET Core Web API - Avançado e Segurança Inicial	08 - Resumo do módulo	Guilherme Paracatu

1. Fundamentação Teórica Avançada

Neste módulo, não apenas escrevemos código; nós desenhamos uma solução sustentável. Vamos dissecar os conceitos de engenharia aplicados.

1. Clean Architecture e o "Shared Kernel"

Imagine o seu projeto como uma cebola, com várias camadas uma dentro da outra. A regra de ouro é: quem está nas camadas de fora pode ver o que está dentro, mas quem está no centro não conhece nada do que acontece lá fora:

- **ITB.Domain (O Núcleo Puro):** Aqui vivem as regras de negócio que *não mudam* mesmo se mudarmos o banco de dados ou a API. Ex: A entidade Fabricante e suas validações.
- **ITB.Domain.Core (Shared Kernel):**
 - *O Problema:* Se Application precisa de interfaces de mensageria e Infrastructure também, criamos dependências circulares.
 - *A Solução:* O Shared Kernel contém contratos base (ICommand, IHandler, IMessageBus) que são estáveis e usados por todo o sistema. É a "linguagem comum" do projeto.
- **ITB.Application (Orquestração):** Não decide *como* salvar, apenas *o que* fazer. Coordena os fluxos (Recebe DTO -> Chama Repositório -> Retorna DTO).
- **ITB.Infrastructure (Adaptadores):** É o mundo sujo. Sabe falar SQL (EF Core), sabe escrever em disco (File Sink) e sabe instanciar classes (IoC).
- **ITB.API (Apresentação):** Apenas uma "casca" burra. Sua única função é receber HTTP e passar a batata quente para a Aplicação.

2. SOLID: A Alma do Projeto

Sem perceber, aplicamos os 5 princípios do SOLID. Veja onde eles estão no seu código:

- **S - Single Responsibility Principle (SRP):**
 - *Antes:* O Controller validava, mapeava e salvava.
 - *Agora:* O Controller só recebe HTTP. O Handler só executa a regra. O Repository só fala com o banco. Cada classe tem apenas **um motivo para mudar**.
- **O - Open/Closed Principle (OCP):**
 - *Conceito:* Aberto para extensão, fechado para modificação.
 - *Aplicação:* Nossa **Log Genérico** (LogComandoGenericoHandler). Se criarmos 50 novos comandos, não precisamos alterar uma linha de código do Log. O sistema aceita novos comportamentos sem mexer no código antigo.
- **L - Liskov Substitution Principle (LSP):**
 - *Aplicação:* O InMemoryBus implementa IMessageBus. Se amanhã quisermos trocar por um RabbitMQBus (Azure Service Bus), o resto do sistema continua funcionando sem saber a diferença, pois o contrato é respeitado.
- **I - Interface Segregation Principle (ISP):**
 - *Aplicação:* Criamos interfaces pequenas e específicas (IHandler<T>, ICommand). Não obrigamos uma classe a implementar métodos que ela não usa.
- **D - Dependency Inversion Principle (DIP):**
 - *O Mais Importante:* Módulos de alto nível (Application) não dependem de módulos de baixo nível (Infrastructure). Ambos dependem de **abstrações** (ITB.Domain.Core).
 - *Na Prática:* O Handler não conhece o AppDbContext; ele conhece apenas a interface IFabricanteRepository.

O SOLID parece um bicho de sete cabeças, mas no nosso projeto ele significa apenas:

- **Cada um no seu quadrado:** Uma classe só deve fazer uma coisa (ex: o Handler não deve saber salvar no banco, ele pede para o Repositório fazer isso).
- **Fácil de aumentar, difícil de estragar:** Conseguimos adicionar 10 novos comandos sem precisar mexer no código que já está pronto e funcionando (como o nosso Log automático).
- **Troca de peças:** Podemos trocar o banco de dados sem precisar reescrever a lógica de negócio, porque trabalhamos com "contratos" (Interfaces).

3. Padrão Mediator (Manual e Nativo)

Decidimos não usar a biblioteca *MediatR* para ter controle total e evitar custos/licenciamento. Implementamos o padrão **Mediator** puro.

- **O Conceito de "Tráfego Aéreo":** Em vez de vários aviões (Controllers) conversarem diretamente com as pistas de pouso (Repositories), eles falam com a Torre de Controle (Bus).
- **Desacoplamento Temporal:** Quem envia a mensagem não precisa saber *quem* vai processar, nem *quando*.
- **Como funciona a nossa implementação manual?**
 - Usamos o `IServiceProvider` do .NET.
 - O `InMemoryBus` recebe um objeto genérico `T`.
 - Ele pergunta ao container de Injeção de Dependência: "*Quem sabe lidar com `IHandler<T>`?*"
 - O .NET devolve a classe concreta (ex: `CriarFabricanteHandler`) e o Bus invoca o método `Handle`. É metaprogramação pura e eficiente.

4. Entity Framework Core & Migrations

Adotamos a abordagem **Code-First** (O código manda no banco).

- **O que é uma Migration?** É um sistema de versionamento para o esquema do banco de dados (como um "Git" para o SQL).
- **Snapshot vs. Model:** Quando você roda `add migration`, o EF Core compara o seu código C# atual (`DbSet<Fabricante>`) com o "Snapshot" (o estado da última migration). A diferença entre eles gera o script de mudança.
- **Idempotência:** Migrations bem feitas permitem que você destrua o banco e o recrie do zero (database update) em qualquer ambiente (Dev, Teste, Prod) garantindo que a estrutura seja idêntica.
- **O Perigo Oculto:** Vimos na prática (logs de erro) que se o tamanho da string no C# (`MaxLength`) não bater com o banco (varchar), ocorrem exceções. Por isso, a Migration é a verdade absoluta da infraestrutura.

5. Pipeline & Middlewares (A Esteira Industrial)

O ASP.NET Core processa requisições como uma esteira industrial.

1. **Request Chega:** Passa pelo `ExceptionMiddleware` -> `Https` -> `Auth` -> `Controller`.
2. **Processamento:** O Controller chama o Bus, que chama o Handler.

3. Response Volta: Passa de volta pelo Auth -> Https -> ExceptionMiddleware.

- **Por que o ExceptionMiddleware é especial?**

- Ele usa um bloco try/catch gigante que envolve toda a execução dos middlewares seguintes (_next).
- Se qualquer peça da engrenagem falhar (banco travou, erro de lógica), o erro "sobe" a pilha de chamada e cai no colo dele.
- Ele transforma o erro técnico (Stack Trace feio) em um protocolo HTTP padronizado (JSON 500), garantindo que a API nunca "morra" silenciosamente.

6. Observabilidade Estruturada (Serilog)

Sair do Console.WriteLine para o **Serilog** é a diferença entre um projeto amador e profissional.

- **Logs Estruturados:** Em vez de concatenar strings ("Erro no id " + id), usamos templates de mensagem ("Erro no id {Id}", id).
 - *Vantagem:* Isso salva os dados como propriedades JSON pesquisáveis, não como texto plano.
- **Enriquecimento (Enrichment):** O log ganha "superpoderes" automáticos, como saber o nome da máquina, a Thread ID e o RequestId, sem você precisar escrever isso.
- **Signal-to-Noise Ratio (Relação Sinal-Ruído):** Um bom log não é aquele que mostra tudo, é aquele que mostra o que importa. Por isso configuramos o Override no appsettings para calar a boca do Entity Framework (Fatal), permitindo que nossos logs de negócio (Information) brilhem no console.

2. Guia Prático de Implementação (Passo a Passo)

Preparação do Ambiente (Terminal do VS Code)

Criar do zero hoje, esta seria a sequência exata.

1. Preparação da Solução e Projetos (Terminal)

Terminal VSCode:

```
# 1. Criar a Solução
```

```
dotnet new sln -n ITB
```

```
# 2. Criar os Projetos
```

```
dotnet new classlib -n ITB.Domain.Core
```

```
dotnet new classlib -n ITB.Domain
```

```
dotnet new classlib -n ITB.Application
```

```
dotnet new classlib -n ITB.Infrastructure
```

```
dotnet new classlib -n ITB.IoC
```

```
dotnet new webapi -n ITB.API
```

```
# 3. Adicionar à solução
dotnet sln add ITB.Domain.Core/ITB.Domain.Core.csproj
dotnet sln add ITB.Domain/ITB.Domain.csproj
dotnet sln add ITB.Application/ITB.Application.csproj
dotnet sln add ITB.Infrastructure/ITB.Infrastructure.csproj
dotnet sln add ITB.IoC/ITB.IoC.csproj
dotnet sln add ITB.API/ITB.API.csproj
```

2. Amarração das Referências (Quem enxerga quem)

Aqui está o ajuste crucial baseado na sua imagem: **Todo mundo precisa ver o Domain.Core.**

Terminal VSCode:

```
# --- DOMAIN CORE (Não depende de ninguém) ---
```

```
# --- DOMAIN (Vê o Core) ---
```

```
dotnet add ITB.Domain reference ITB.Domain.Core
```

```
# --- APPLICATION (Vê Domain e Core) ---
```

```
dotnet add ITB.Application reference ITB.Domain
```

```
dotnet add ITB.Application reference ITB.Domain.Core
```

```
# --- INFRASTRUCTURE (Vê Domain, Application e Core) ---
```

```
dotnet add ITB.Infrastructure reference ITB.Domain
```

```
dotnet add ITB.Infrastructure reference ITB.Application
```

```
dotnet add ITB.Infrastructure reference ITB.Domain.Core
```

```
# --- IOC (O Maestro - Vê todos) ---
```

```
dotnet add ITB.IoC reference ITB.Domain
```

```
dotnet add ITB.IoC reference ITB.Domain.Core
```

```
dotnet add ITB.IoC reference ITB.Application
```

```
dotnet add ITB.IoC reference ITB.Infrastructure
```

```
# --- API (Vê Application e IoC) ---
```

```
dotnet add ITB.API reference ITB.Application
```

```
dotnet add ITB.API reference ITB.IoC
```

3. Implementação do Mediator (Baseado na Imagem)

Local: ITB.Domain.Core/Messages/Interfaces

As interfaces "puras" ficam aqui para que qualquer camada possa criar um comando ou handler.

CÓDIGO C#:

```
// ICommand.cs
public interface ICommand { }
```

```
// IHandler.cs
public interface IHandler<T> where T : ICommand
```

```

{
    Task Handle(T command);
}

// IMessageBus.cs
public interface IMessageBus
{
    Task EnviarComando<T>(T comando) where T : ICommand;
}

```

Local: ITB.Infrastructure/Bus/InMemoryBus.cs

A implementação concreta de como o bus funciona fica na infraestrutura.

CÓDIGO C#:

```

using ITB.Domain.Core.Messages.Interfaces; // Using correto!

public class InMemoryBus : IMessageBus
{
    private readonly IServiceProvider _serviceProvider;

    public InMemoryBus(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    public async Task EnviarComando<T>(T comando) where T : ICommand
    {
        // Busca quem sabe resolver esse comando específico
        var handlers = _serviceProvider.GetServices<IHandler<T>>();

        foreach (var handler in handlers)
        {
            await handler.Handle(comando);
        }
    }
}

```

4. Infraestrutura de Banco (EF Core)

Comandos de Instalação:

Terminal VSCode:

```

dotnet add ITB.Infrastructure package Microsoft.EntityFrameworkCore
dotnet add ITB.Infrastructure package Npgsql.EntityFrameworkCore.PostgreSQL
dotnet add ITB.Infrastructure package Microsoft.EntityFrameworkCore.Tools

```

Comandos de Migrations:

Terminal VSCode:

Criar

```
dotnet ef migrations add AjusteTabelas --project ITB.Infrastructure --startup-project ITB.API
```

Aplicar

```
dotnet ef database update --project ITB.Infrastructure --startup-project ITB.API
```

5. Observabilidade (Serilog)

Comandos de Instalação:

Terminal VSCode:

Na API

```
dotnet add ITB.API package Serilog.AspNetCore  
dotnet add ITB.API package Serilog.Sinks.Console  
dotnet add ITB.API package Serilog.Sinks.File
```

No IoC (Para configuração)

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

Configuração Limpa (appsettings.Development.json):

JSON:

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

6. Configuração Final (IoC e Middleware)

Local: ITB.IoC/DependencyInjection.cs

CÓDIGO C#:

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

```

    {
        // 1. Banco de Dados
        var connectionString = configuration.GetConnectionString("DefaultConnection");
        services.AddDbContext<AppDbContext>(options =>
            options.UseNpgsql(connectionString));

        // 2. Mediator (Bus e Handlers)
        services.AddScoped<IMessageBus, InMemoryBus>();
        services.AddScoped<IHandler<CriarFabricanteCommand>, CriarFabricanteHandler>();

        // 3. Log Genérico (Open Generics)
        services.AddScoped(typeof(IHandler<>), typeof(LogComandoGenericoHandler<>));

        return services;
    }

    public static void AddSerilogApi(this IHostBuilder host)
    {
        host.UseSerilog((context, config) =>
        {
            config
                .ReadFrom.Configuration(context.Configuration)
                .WriteTo.Console()
                .WriteTo.File("Logs/Log-.txt", rollingInterval: RollingInterval.Day);
        });
    }
}

```

Local: ITB.API/Program.cs

CÓDIGO C#:

```

var builder = WebApplication.CreateBuilder(args);

builder.Host.AddSerilog();
builder.Services.AddInfrastructure(builder.Configuration);

var app = builder.Build();

// ORDEM DO PIPELINE (CRUCIAL):
app.UseMiddleware<ExceptionMiddleware>(); // Captura erros globais
app.UseHttpsRedirection();
app.MapControllers();
app.Run();

```

Checklist Final

- Estrutura:** O projeto ITB.Domain.Core existe e contém as interfaces IMessageBus, ICommand e IHandler?
- Logs:** Ao rodar a API, o console está limpo (sem spam do EF Core)?
- Funcionamento:** Ao fazer um POST no Swagger, você vê o log [INF] Pipeline Mediator...?
- Resiliência:** Ao forçar um erro (string gigante), o log mostra [ERR] FALHA CRÍTICA com o motivo real do banco?

O Guia de Sobrevivência .NET

Para te ajudar na jornada, preparamos este guia rápido para solucionar problemas e lembrar dos comandos essenciais.

1. Top 5 Erros Comuns (Troubleshooting)

Se o terminal ficar vermelho, não entre em pânico. Verifique esta lista:

Erro 1: Unable to resolve service for type 'ITB.Domain...'.

- O que significa:** O sistema de Injeção de Dependência (DI) não sabe criar a classe que você pediu no construtor.
- Causa Provável:** Você criou um novo Repository ou Handler, mas esqueceu de registrá-lo no arquivo DependencyInjection.cs na camada IoC.
- Solução:** Vá em ITB.IoC e adicione services.AddScoped<ISeuServiço, SuaImplementação>();.

Erro 2: 22001: value too long for type character varying(X)

- O que significa:** Você tentou salvar um texto maior do que o banco de dados permite.
- Causa Provável:** No C#, a string é infinita, mas no Banco (Postgres), a coluna foi criada com limite (ex: VARCHAR(100)).
- Solução:**
 - Verifique o tamanho do dado enviado no Swagger.
 - Se precisar aumentar o limite, altere a configuração na entidade (FluentAPI ou DataAnnotation).
 - Importante:** Crie uma nova Migration e atualize o banco para aplicar a mudança.

Erro 3: Npgsql.PostgresException: 28P01: password authentication failed

- O que significa:** A API não conseguiu se conectar ao PostgreSQL.
- Causa Provável:**
 - A senha no appsettings.Development.json está errada.
 - O container do Docker não está rodando.
- Solução:** Verifique se o Docker está verde (Running) e confira a ConnectionString.

Erro 4: PendingModelChangesWarning: The model backing the context has changed

- O que significa:** O Entity Framework percebeu que você mudou uma classe de Entidade, mas o banco de dados ainda está na versão antiga.

- **Solução:** Você esqueceu de criar a Migration! Rode dotnet ef migrations add e depois dotnet ef database update.
- **Erro 5: O Log não aparece no Console (Apenas "Building...")**
- **O que significa:** O Log está sendo filtrado ou a aplicação travou antes de iniciar.
 - **Solução:**
 1. Verifique o appsettings.Development.json. O Default deve ser Information.
 2. Verifique se o Serilog foi configurado no Program.cs (builder.Host.UseSerilog(...)).
 3. Se estiver travado no VS Code, use o comando Developer: Reload Window.

↳ 2. Comandos Essenciais

Salve esta lista. Em Arquitetura Limpa, os comandos são longos porque precisamos apontar os projetos certos.

⌚ Entity Framework Core (Migrations)

Sempre execute estes comandos na pasta raiz da Solução (onde está o arquivo .sln).

Ação	Comando	Explicação
Criar Migration	dotnet ef migrations add NomeDaMudanca --project ITB.Infrastructure --startup-project ITB.API	Tira uma "foto" das suas entidades e cria o script de mudança na pasta Migrations.
Atualizar Banco	dotnet ef database update --project ITB.Infrastructure --startup-project ITB.API	Aplica as mudanças pendentes no banco de dados real.
Remover Última	dotnet ef migrations remove --project ITB.Infrastructure --startup-project ITB.API	Desfaz a criação da <i>última</i> migration (só funciona se não tiver aplicado no banco ainda).
Gerar Script SQL	dotnet ef migrations script --project ITB.Infrastructure --startup-project ITB.API	Gera o SQL puro para você rodar manualmente (útil para Produção).

Dica: O parâmetro --project diz onde estão as Migrations (Infra). O --startup-project diz quem tem a ConnectionString (API).

⌚ Como voltar atrás se eu JÁ fiz o database update?

Se você aplicou a migration no banco e percebeu que o nome da tabela ou o tamanho da coluna está errado, você precisa seguir **exatamente** estes dois passos:

Passo 1: Voltar o estado do Banco de Dados Você precisa dizer ao EF para voltar o banco para a versão anterior. Digite o nome da migration que estava **antes** da que você quer apagar:

Terminal VSCode:

```
dotnet ef database update NomeDaMigrationAnterior --project ITB.Infrastructure --startup-project ITB.API
```

O banco de dados agora "esqueceu" a última mudança, mas os arquivos de código ainda existem no seu projeto.

Passo 2: Remover os arquivos de código Agora que o banco está em segurança, você pode deletar os arquivos da migration "errada":

Terminal VSCode:

```
dotnet ef migrations remove --project ITB.Infrastructure --startup-project ITB.API
```

⚠️ Atenção: Ao voltar uma migration (database update), se você tiver deletado uma coluna ou tabela, os dados que estavam nela **serão perdidos**. Migrations são poderosas, use com atenção!

💻 .NET CLI (Terminal)

Ação	Comando
Rodar a API	dotnet run --project ITB.API/ITB.API.csproj
Limpar Cache	dotnet clean (Resolve erros fantasma de build)
Adicionar Pacote	dotnet add ITB.NomeDoProjeto package NomeDoPacote
Adicionar Referência	dotnet add ITB.API reference ITB.Application (Liga um projeto ao outro)

⌨️ 3. Atalhos do VS Code

Atalho (Windows/Linux)	O que faz	Por que usar?
Ctrl + . (Ponto)	Quick Fix	Resolve imports (using), implementa interfaces e corrige erros de digitação. É o atalho mais importante.
Ctrl + Shift + P	Command Palette	A barra de busca universal. Use para Reload Window, Clean, etc.
Shift + Alt + F	Format Document	Indent o código bagunçado automaticamente.
F12	Go to Definition	Navega para a classe/método que está sendo chamado.
Ctrl + P	Go to File	Digite o nome do arquivo para abri-lo sem usar o mouse na árvore lateral.
Ctrl + ' (Aspas)	Toggle Terminal	Abre e fecha o terminal integrado rapidamente.