

## INFORMAÇÕES

MÓDULO	AULA	INSTRUTOR
06 ASP.NET Core Web API - Avançado e Segurança Inicial	04 - ARQUITETURA EM CAMADAS, FUNDAMENTOS SOLID E EVOLUÇÃO COM CODE FIRST (MIGRATIONS)	Guilherme Paracatu

### 1. Engenharia de Software e Design de APIs Profissionais

#### 1. O CONCEITO: POR QUE DIVIDIR O PROJETO?

Antes de olharmos as pastas, precisamos entender o **Soc** - Separation of Concerns (Separação de Preocupações).

O que é **Soc**? É o princípio de design que prega que um sistema deve ser dividido em partes distintas, onde cada parte cuida de uma "preocupação" única do software.

#### Caso Real (O Restaurante):

Imagine um restaurante onde o Cozinheiro também atende as mesas, lava o chão e faz o marketing. Se ele ficar doente ou se o restaurante crescer, o sistema entra em colapso. Em uma arquitetura profissional, temos papéis definidos: o Garçom (API), o Cozinheiro (Domain) e o Estoquista (Infrastructure).

#### 2. DESBRAVANDO A ESTRUTURA E O SOLID

Vamos analisar a nossa **Solution** e entender o papel de cada projeto:

##### 1 - API (A Porta de Entrada)

É o projeto que "fala" com a internet via protocolo HTTP.

- **Controllers:** Recebem a requisição e devolvem a resposta.
- **SOLID (Letra S - SRP):** Aqui aplicamos o Single Responsibility Principle (Princípio da Responsabilidade Única). O Controller tem apenas uma responsabilidade: orquestrar a entrada e saída. Ele **não** faz cálculos de imposto e **não** salva no banco.

##### 2 - Application (O Orquestrador)

Serve de ponte entre a API e o Coração (Domain).

- **Dtos:** Objetos que moldam os dados para o usuário (evita expor segredos do banco).

- **Services:** Onde a mágica da regra de negócio acontece.
- **SOLID (Letra D - DIP):** Aplicamos o **Dependency Inversion Principle (Princípio da Inversão de Dependência)**. O serviço não depende de uma classe do banco, mas sim de uma **Interface** (um contrato).

### 📁 3 - Domain (O Coração)

- **Entities:** Classes principais (Produto, Fabricante).
- **Interfaces:** Aqui definimos os Contratos. O Domínio diz o que precisa ser feito, mas não como "Eu preciso de alguém que salve um Produto. Não me importa se você vai usar PostgreSQL, SQL Server ou um caderno. Eu só exijo que o método se chame Salvar(Produto p)".
- **SOLID (Letra D - DIP):** Aplicamos o Princípio da Inversão de Dependência. Em vez do Domínio depender do Banco de Dados, o Banco de Dados (**Infra**) é quem deve se adaptar às necessidades do Domínio através das Interfaces.
- **SOLID (Letra O - OCP):** O Domínio está fechado para modificações (não mudamos a regra de negócio se trocarmos o banco) e aberto para extensões (podemos criar um novo repositório de backup apenas implementando a interface existente).

### 📁 4 - Infrastructure (A Ferramenta)

Onde a tecnologia "suja as mãos" com o PostgreSQL.

- **Persistence:** Local do seu DbContext e configurações de tabelas.
- **Repositories:** Onde o código SQL/EF Core realmente é escrito.
- **SOLID (Letra L - LSP):** Aplicamos o **Liskov Substitution Principle (Princípio da Substituição de Liskov)**. Podemos trocar o repositório de PostgreSQL por um de SQL Server sem que o resto do sistema perceba a diferença, pois ambos respeitam o mesmo contrato.

### 📁 5 - CrossCutting (O Utilitário)

Contém funcionalidades técnicas que podem ser compartilhadas por qualquer camada, mas que não possuem regras de negócio.

- **Segurança e Criptografia:** Classes para hash de senhas ou utilitários de strings.
- **Independência:** Nesta nova estrutura, o CrossCutting é "puro", ou seja, ele não conhece as outras camadas, permitindo que todos o utilizem sem gerar erros de referência circular.

## 6 - IoC (Inversion of Control - O Cérebro)

Este projeto é o Agregador Central da aplicação. Sua única função é registrar como as peças do sistema se encaixam.

- **Injeção de Dependência:** Centraliza todas as configurações do Entity Framework, Repositórios, Serviços e AutoMapper.
- **SOLID (Letra D - DIP):** Aqui aplicamos o **Dependency Inversion Principle** (Princípio da Inversão de Dependência). A API não precisa saber como o Banco de Dados funciona; ela apenas pede ao **IoC** para entregar uma implementação para as interfaces.
- **SOLID (Letra I - ISP):** Ao registrar nossos serviços, garantimos que cada classe dependa apenas das interfaces que realmente utiliza, promovendo um sistema modular e fácil de testar.

### 3. CENTRALIZANDO A INJEÇÃO DE DEPENDÊNCIA (IoC)

Para manter o Program.cs limpo (respeitando o **S** do SOLID), criamos uma classe na pasta **IoC** dentro do **CrossCutting**.

**Por que fazer isso?** Imagine uma oficina (API). Se todas as ferramentas (Repositórios e Serviços) estiverem jogadas no balcão de entrada (Program.cs), ninguém consegue trabalhar. A pasta IoC é o nosso **quadro de ferramentas organizado**, onde cada peça tem seu lugar.

### 4. RESUMO DOS PRINCÍPIOS SOLID

1. **S (SRP):** Uma classe deve ter apenas um motivo para mudar.
2. **O (OCP):** Você deve ser capaz de adicionar novos recursos sem mudar o código antigo.
3. **L (LSP):** Classes filhas (ou implementações) devem poder substituir suas bases sem erros.
4. **I (ISP):** É melhor ter várias interfaces específicas do que uma genérica que ninguém entende.
5. **D (DIP):** Dependa de abstrações (Interfaces), não de implementações (Classes concretas).

## Tarefa de Fixação: Módulo de Categorias

**Objetivo:** Implementar o CRUD completo para Categorias (ex: "Periféricos", "Hardware", "Software"), seguindo a mesma estrutura de camadas usada no Fabricante.

### 1. O que deve ser criado (Passo a Passo):

- **Camada Domain:**
  - Criar a entidade Categoria (Propriedades: Id e Nome).
- **Camada Application:**
  - Criar os DTOs: CategoriaReadDto, CategoriaCreateDto e CategoriaUpdateDto.
  - Criar a interface ICategoriaService.
  - Implementar o CategoriaService (usando o IMapper).
  - Configurar o mapeamento no MappingProfile.
- **Camada Infrastructure:**
  - Criar a interface ICategoriaRepository.
  - Implementar o CategoriaRepository.
- **Camada IoC:**
  - Registrar as novas dependências no DependencyInjection.cs.
- **Camada API:**
  - Criar o CategoriaController.

## Script de criação da tabela

### Script SQL

```
CREATE TABLE categorias (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL
);
```

## 💡 A Analogia da Cozinha (Service vs. Handler)

Vamos entender como os pedidos são preparados na evolução da arquitetura:

- 🍴 **O Service é o "Cozinheiro Geral"**: Ele sabe fazer tudo. Se você pedir a entrada, o prato principal ou a sobremesa, é ele quem faz. O FabricanteService tem os métodos Listar(), Criar(), Atualizar() e Deletar(). Ele é um arquivo que tende a crescer e acumular muitas responsabilidades.
- 🍷 **O Handler é o "Chef Especialista"**: Em vez de um cozinheiro geral, imagine que você tem um especialista só para "Grelhar Carne" e outro só para "Preparar Salada". Cada Handler cuida de **uma única ação**. Se você quer criar um fabricante, chama o CriarFabricanteHandler. Ele não sabe listar nem deletar; ele é mestre em apenas uma tarefa.

## 📊 Comparação Técnica

Característica	Service (Arquitetura Atual)	Handler (Arquitetura com Bus Nativo)
Escopo	Geral. Uma classe para todas as ações.	Único. Uma classe para cada ação específica.
Complexidade	Tende a ficar gigante (500+ linhas).	Arquivos minúsculos e focados.
SOLID (Letra S)	Aplica o SRP na Entidade.	Aplica o SRP no nível máximo.
Dependência	O Controller injeta o Service inteiro.	O Controller injeta o Bus, que entrega a tarefa.
Biblioteca	Nativa do .NET.	<b>Nativa!</b> (Sem pacotes externos como MediatR).

## 💡 Entendendo o Padrão Mediator (O Mediador)

### O que é?

Imagine um **Aeroporto**. Se cada piloto tivesse que falar diretamente com todos os outros pilotos para decidir quem pousa agora, teríamos um caos total (e muitos acidentes). Em vez disso, todos os pilotos falam com um único ponto central: a **Torre de Comando**. A torre é o **Mediador**. Os pilotos não precisam se conhecer; eles só precisam conhecer a torre.

### Qual problema ele resolve?

O problema do **Acoplamento Excessivo**. Sem o Mediator, suas classes ficam "viciadas" umas nas outras. O ServiceA precisa do ServiceB, que precisa do ServiceC... Quando você tenta mudar um pedaço, o sistema inteiro reclama. O Mediator quebra esse vício.

## Service vs. Mediator: Quando usar cada um?

Não existe "melhor", existe a ferramenta certa para o momento do projeto.

### Use o Padrão Service (O Cozinheiro Geral)

- **Projetos Pequenos ou Iniciais:** Quando você tem poucas tabelas e poucas regras.
- **Rapidez:** É muito mais rápido criar um único ProdutoService com 4 métodos (CRUD) do que criar 4 Handlers diferentes.
- **Fluxos Simples:** Quando a ação é apenas "salvar no banco" e nada mais.
- **Dificuldade:** Baixa. Ótimo para quem está começando a entender camadas.

### Use o Padrão Mediator/Handlers (Os Especialistas)

- **Sistemas que Crescem:** Quando o seu Service passou de 300 linhas e você começa a ter medo de mexer nele.
- **Muitas Dependências:** Se para "Criar Produto" você precisa enviar E-mail, baixar Estoque, gerar Log e avisar o Financeiro. No Mediator, cada especialista cuida do seu, sem poluir o código do outro.
- **Times Maiores:** Quando vários desenvolvedores mexem na mesma entidade. Com Handlers, cada um trabalha em seu arquivo, eliminando conflitos de código (*Merge Conflicts*).
- **Dificuldade:** Média. Exige que o desenvolvedor já entenda bem de Interfaces e Injeção de Dependência.

## Comparação Visual

Situação	Service	Mediator
Ação simples (Ex: Listar cidades)	<input checked="" type="checkbox"/> Ideal	<input checked="" type="checkbox"/> Exagerado (Overkill)
Ação complexa (Ex: Fechar Pedido)	<input type="checkbox"/> Fica confuso	<input checked="" type="checkbox"/> Perfeito e Isolado
Manutenção rápida	<input checked="" type="checkbox"/> Muito fácil	<input type="checkbox"/> Exige abrir vários arquivos
Escalabilidade	<input checked="" type="checkbox"/> Difícil de manter	<input checked="" type="checkbox"/> Feito para crescer

## O "Resumo":

"Se o seu sistema é um **apartamento de 1 quarto**, você mesmo limpa tudo (Service). Se o seu sistema é um **Shopping Center**, você precisa de uma equipe de especialistas, onde cada um cuida da sua área e a administração centraliza os chamados (Mediator/Bus)."

### O "Pulo do Gato": Do Service para o Handler Nativo

Estamos aprendendo **Service/Repository** agora porque ele é o padrão mais comum. No entanto, conforme o sistema cresce, o **FabricanteService** começa a receber tantas dependências (Log, Email, Estoque) que fica difícil de manter.

Antigamente, usávamos uma biblioteca chamada **MediatR** para resolver isso. **Hoje, evoluímos:** criamos nosso próprio sistema de mensagens (**InMemoryBus**). Isso significa:

1. **Zero Overhead:** O sistema não fica "pesado" com bibliotecas de terceiros.
2. **Performance:** O envio da tarefa é direto e ultra-rápido.
3. **Controle Total:** Nós somos donos do "motor" que entrega as mensagens.

### O Spoiler: A Evolução do Código

"Hoje, construímos o **Canivete Suíço (Service)**. Nas próximas aulas, vamos aprender a criar **Ferramentas de Precisão (Handlers)** usando nosso próprio **Bus de Dados**."

#### 1. Como estamos fazendo hoje (O Service)

Tudo dentro de uma única classe. Se precisarmos de 10 ações, teremos 10 métodos e uma lista enorme de dependências no topo.

#### 2. Como faremos no futuro (O Handler com Bus Nativo)

Nós "fatiamos" o Service. Criamos um **Command** (o pedido) e um **Handler** (quem executa).

- **Command:** CriarFabricanteCommand (apenas os dados).
- **Handler:** CriarFabricanteHandler (apenas a lógica de criação).

### O que é o InMemoryBus (Nosso Mediador Nativo)?

O **InMemoryBus** é o "garçom" do nosso sistema. Ele recebe um pedido do Controller e, usando a **Injeção de Dependência** que configuramos no nosso projeto **IoC**, ele descobre quem é o especialista (Handler) que sabe resolver aquele pedido.

#### Por que remover o MediatR e usar o Bus Nativo?

1. **Simplicidade:** Não precisamos de "etiquetas" de bibliotecas externas (IRequest, INotification). Usamos nossas próprias interfaces ( ICommand, IEvent).
2. **Manutenibilidade:** Menos pacotes NuGet significa menos atualizações e menos riscos de segurança.
3. **Transparência:** Você consegue ver exatamente como a mensagem sai do Controller e chega no Handler, sem "mágica" escondida dentro de DLLs de terceiros.

### 💡 Analogia Final: (O Garçom Particular vs. Agência de Entregas)

"Usar o MediatR é como contratar uma Grande Agência de Entregas: funciona bem, mas você tem que seguir as regras deles, usar as caixas deles e pagar a taxa de serviço.

Usar o nosso InMemoryBus Nativo é como ter um Garçom Particular: ele já conhece sua cozinha (IoC), fala a sua língua, não cobra taxas extras e entrega o pedido instantaneamente na mão do Chef."

### 📋 Resumo:

- **O Padrão:** Continua sendo o **Mediator** (um intermediário).
- **A Mudança:** Tiramos a "muleta" da biblioteca externa para usar o poder nativo do .NET.
- **O Ganho:** Código mais limpo, mais rápido e 100% controlado por nós.

### 📊 A Comparação Direta

Ponto de Vista	Service (Como estamos fazendo agora)	Mediator (Como faremos no futuro)
Onde está a lógica?	Em métodos dentro de uma classe grande.	Em classes pequenas e separadas (Handlers).
Quem o Controller chama?	O FabricanteService diretamente.	O InMemoryBus (A Central).
E se o sistema crescer?	O Service vira um "arquivo monstro" impossível de ler.	Você apenas cria novos arquivos pequenos sem mexer nos antigos.
Dependências	O Service carrega tudo nas costas (Repo, Log, Email...).	O Handler só carrega o que ele realmente vai usar.

## 2. GUIA DE IMPLEMENTAÇÃO: ENTITY FRAMEWORK CORE MIGRATIONS (VS CODE)

### 1. O CONCEITO: DO DATABASE FIRST AO CODE FIRST

Até agora, trabalhamos no modo **Database First** (O Banco manda no código). A partir de hoje, adotamos o **Code First** (O Código manda no Banco).

**Por que mudar?**

1. **S do SOLID (SRP):** A estrutura de dados agora é definida no **Domain** (Coração), que é o dono das regras. O banco torna-se apenas um reflexo do código.

2. **Versionamento:** As Migrations são o "Git do Banco de Dados". Cada alteração gera um arquivo que conta a história da evolução do sistema.
3. **Consistência:** Garante que todos os desenvolvedores do time tenham exatamente a mesma estrutura de tabelas apenas rodando um comando.

## 2. PREPARANDO O TERRENO (CONFIGURAÇÃO)

Como utilizamos uma **Arquitetura em Camadas com IoC**, precisamos garantir que as ferramentas do Entity Framework encontrem nossa Connection String.

### 🛠️ Instalação das Ferramentas (Terminal na Raiz):

```
# 1. Instala a ferramenta de linha de comando globalmente  
dotnet tool install --global dotnet-ef  
  
# 2. Instala o pacote de Design na Infraestrutura (onde está o DbContext)  
dotnet add ITB.Infrastructure package Microsoft.EntityFrameworkCore.Design  
  
# 3. Instala o pacote de Design na API (Startup Project)  
dotnet add ITB.API package Microsoft.EntityFrameworkCore.Design
```

## 3. O CORAÇÃO DO SISTEMA: O DBCONTEXT

Para o Migrations funcionar com **Injeção de Dependência (IoC)**, seu LojaDbContext precisa de um construtor específico que receba as configurações da nossa camada de **CrossCutting**.

### Código C#

```
// Local: ITB.Infrastructure/Persistence/LojaDbContext.cs  
public class LojaDbContext : DbContext  
{  
    public LojaDbContext(DbContextOptions<LojaDbContext> options) :  
    base(options) { }  
  
    public DbSet<Fabricante> Fabricantes { get; set; }  
  
    protected override void OnModelCreating(ModelBuilder modelBuilder)  
    {
```

```
// Aplica todas as configurações de nomes de tabela e colunas (Fluent API)
modelBuilder.ApplyConfigurationsFromAssembly(typeof(LojaDbContext).Assembly);
}
```

## O que muda no código?

A maior mudança não é no conteúdo, mas na **organização**. No seu código anterior, todas as tabelas estavam "amontoadas" dentro do método OnModelCreating. Em um projeto real com 50 tabelas, esse arquivo ficaria impossível de ler.

**A forma moderna (usando Reflection):** Substituímos o mapeamento manual de cada entidade por uma única linha que busca as configurações em arquivos separados.

## Explicando a "Mágica": ApplyConfigurationsFromAssembly

*Como o DbContext sabe onde estão os arquivos de Map sem que eu precise registrar um por um?", a resposta está nessa linha:*

```
modelBuilder.ApplyConfigurationsFromAssembly(typeof(LojaDbContext).Assembly);
```

### Tradução:

1. **typeof(LojaDbContext)**: Pega o tipo da nossa classe de banco.
2. **.Assembly**: O "Assembly" é como se fosse o "pacote" ou o arquivo .dll onde todo o nosso projeto de Infraestrutura está empacotado.
3. **ApplyConfigurationsFromAssembly(...)**: Este comando diz ao Entity Framework o seguinte:

*"Ei, EF! Entre neste pacote (Assembly), vasculhe todas as classes que você encontrar e veja quais delas implementam o contrato IEntityTypeConfiguration. Quando achar, execute as regras que estão dentro delas automaticamente."*

### Por que isso é bom?

- **SOLID (Letra O - OCP)**: Você nunca mais precisa abrir o arquivo LojaDbContext.cs para configurar tabelas. O projeto está **fechado** para modificação naquela parte, mas **aberto** para extensão: basta criar um novo

arquivo .cs de Map e o sistema o reconhecerá sozinho na próxima vez que rodar.

- **Organização:** Evita que o OnModelCreating vire um "arquivo linguiça" com 2.000 linhas de código.

### Por que usar arquivos separados (Map)?

Agora, criamos uma pasta chamada Mappings (ou Configuration) dentro da Infraestrutura. Cada tabela ganha seu próprio arquivo (ex: ProdutoMap.cs).

- **SOLID (Letra S - SRP):** No seu código anterior, o DbContext tinha duas responsabilidades: Gerenciar a conexão e configurar as tabelas. Agora, o DbContext apenas gerencia a conexão, e cada classe Map cuida de sua própria tabela.
- **SOLID (Letra O - OCP):** Se precisarmos adicionar uma tabela nova, não "abrimos" o DbContext para escrever mais linhas. Apenas criamos um novo arquivo de mapeamento. O DbContext está **fechado** para modificação, mas o sistema está **aberto** para extensão.

### 3. O detalhe dos Relacionamentos (Cuidado!)

No código anterior, você tinha:

```
.HasConstraintName("produtos_fabricanteid_fkey")
```

O que muda:

- **No Database First:** Você precisava desse nome exato porque o banco já tinha criado a restrição com esse nome.
- **No Code First (Migrations):** Você pode remover essa linha! O Entity Framework vai criar um nome padrão automático para a chave estrangeira (FK). Só usamos o HasConstraintName se tivermos uma exigência muito específica de nomenclatura do DBA.

### 4. Resumo

**O que tínhamos (Database First):** Um DbContext que servia de "dicionário" para traduzir o que já existia no PostgreSQL para o C#.

**O que temos agora (Code First):** Um DbContext que é o **Líder da Equipe**.

1. Ele limpa o OnModelCreating para ficar mais legível.

2. Ele usa o **Migrations** para ler as classes e decidir quais comandos SQL (CREATE TABLE, ALTER TABLE) ele deve enviar para o banco.
3. Ele automatiza a criação de Índices e Chaves Estrangeiras, sem precisarmos decorar os nomes técnicos que o PostgreSQL usa internamente.

#### **4. EXEMPLO PRÁTICO: ENTIDADE FABRICANTE**

Iniciaremos pelo **Fabricante**, pois ele é a base para os produtos.

##### **Passo 1: Criar a Entidade (Domain)**

###### **Código C#**

// Local: ITB.Domain/Entities/Fabricante.cs

```
public class Fabricante {
    public int Id { get; set; }
    public string Nome { get; set; } = string.Empty;
}
```

##### **Passo 2: Mapeamento Independente (O Padrão Map)**

Em vez de poluir o DbContext, criamos uma classe de configuração. Isso aplica o **SRP (Responsabilidade Única)**: o DbContext gerencia a sessão, e o Map gerencia o desenho da tabela.

###### **Criando o arquivo FabricanteMap.cs**

Crie uma pasta chamada **Mappings** (ou **Configurations**) dentro de **ITB.Infrastructure/Persistence**.

###### **Código C#**

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using ITB.Domain.Entities;

public class FabricanteMap : IEntityTypeConfiguration<Fabricante>
{
    public void Configure(EntityTypeBuilder<Fabricante> builder)
    {
```

```

// 1. Nome da Tabela
builder.ToTable("fabricantes");

// 2. Chave Primária
builder.HasKey(f => f.Id);

// 3. Configuração de Colunas e Tipos (O que você precisa saber)
builder.Property(f => f.Id)
    .HasColumnName("id")
    .ValueGeneratedOnAdd(); // Garante que o banco gere o ID
(Serial/Identity)

builder.Property(f => f.Nome)
    .HasColumnName("nome")
    .HasColumnType("varchar(100)") // Força o uso de varchar em vez
de text
    .IsRequired(); // NOT NULL
}
}

```

## 2. Dicionário de Configurações para Migrations

Quando estamos no **Code First**, precisamos ser específicos para o banco não criar colunas "pesadas" demais.

- **HasColumnType("varchar(100)")**: No PostgreSQL, se você não definir, ele pode criar como text. O varchar(N) limita o tamanho e ajuda na performance.
- **IsRequired()**: Define que a coluna é **NOT NULL**. Se você não colocar, o EF assume que pode ser nula (dependendo da sua classe C#).
- **HasMaxLength(N)**: Define o limite de caracteres (também gera o varchar).
- **HasDefaultValue(valor)**: Define um valor padrão caso nada seja enviado.

## Passo 3: Criar a Migração (Terminal)

Este comando lê suas classes e gera um arquivo C# com as instruções para o PostgreSQL.

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

## 5. ENTENDENDO O ARQUIVO DE MIGRAÇÃO: UP() E DOWN()

Ao rodar o comando acima, uma pasta **Migrations** surgirá. Dentro dela, você verá dois métodos:

- **Método Up():** Contém os comandos para **subir** as alterações (ex: `CreateTable`). É executado quando avançamos com o banco.
- **Método Down():** Contém os comandos para **reverter** a alteração (ex: `DropTable`). É a nossa "saída de emergência" caso algo dê errado.

## 6. APLICANDO NO BANCO DE DADOS (DATABASE UPDATE)

Para que a tabela realmente apareça no PostgreSQL (pgAdmin), execute:  
`dotnet ef database update --project ITB.Infrastructure --startup-project ITB.API`

**Atenção:** Como vocês usavam Database First, o banco antigo pode conflitar.

**Sugestão:** Apague (Drop) o banco de dados no pgAdmin e deixe o EF Core criar tudo do zero com o comando acima.

### ⚠ GUIA DE ERROS COMUNS

Erro	Causa Provável	Solução
dotnet-ef não reconhecido	Ferramenta não instalada	Rodar <code>dotnet tool install --global dotnet-ef</code>
Startup project doesn't reference Design	Falta o pacote Design na API	Rodar <code>dotnet add ITB.API package Microsoft.EntityFrameworkCore.Design</code>
Erro de Conexão Npgsql	Docker/Postgres desligado	Inicie o serviço do banco de dados

## Passo 4: O Fluxo da Segunda Migration (Adicionando Coluna)

Digamos que o cliente pediu para salvar o **CNPJ** do fabricante. Veja como é simples evoluir o banco sem perder os dados.

### 1: Alterar a Entidade (Domain)

Adicione a propriedade na sua classe Fabricante:

**Código C#**

```
public string CNPJ { get; set; } = string.Empty;
```

## 2: Atualizar o Mapeamento (Infrastructure)

Adicione a regra no FabricanteMap.cs:

### Código C#

```
builder.Property(f => f.CNPJ)
    .HasColumnName("cnpj")
    .HasColumnType("varchar(14)");
```

## 3: Gerar e Aplicar a Evolução (Terminal)

Agora você não está mais criando o banco, está **evoluindo** ele.

```
# 1. Cria o arquivo com a diferença (Alter Table)
dotnet ef migrations add AdicionarCnpjFabricante --project ITB.Infrastructure -
--startup-project ITB.API
```

# 2. Executa a mudança no banco

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

## 💡 Resumo Técnico

1. **Onde configurar o tamanho?** No arquivo de Map, usando `.HasMaxLength()` ou `.HasColumnType()`.
2. **Como forçar varchar?** Usando `.HasColumnType("varchar(N)")`.
3. **Posso rodar migrations várias vezes?** Sim! Você pode ter centenas de migrations. O Entity Framework guarda uma tabela chamada `_EFMigrationsHistory` no seu banco para saber quais ele já rodou e quais ainda faltam rodar.

## 💡 Por que isso é importante no SOLID?

Isso é o **OCP (Princípio Aberto/Fechado)** em sua forma mais pura:

- O banco está **Fechado** para alterações manuais e perigosas (ninguém vai lá dar um `DROP COLUMN` por engano).
- O banco está **Aberto** para extensões através das Migrations, que mantêm o histórico de tudo o que foi feito.

### 3. TAREFA DE CASA: DESAFIO CODE FIRST

Agora que criamos o Fabricante juntos, sua missão é implementar a tabela de Produtos seguindo o fluxo Code First:

1. Crie/Ajuste a Entidade Produto no projeto Domain.
2. Lembre-se da relação: Um Produto tem um Fabricante (FabricanteId).
3. Crie uma nova Migration chamada AdicionarTabelaProduto.
4. Atualize o banco de dados usando o comando database update.
5. DICA SOLID: Verifique se os nomes das colunas estão em minúsculo na sua configuração de Fluent API dentro da Infraestrutura!

*"No Code First, o seu código é a Lei. Respeite o Domínio e o Banco de Dados será apenas uma consequência."*