FMX Academy

INFORMAÇÕES		
MÓDULO	**AULA**	**INSTRUTOR**
05 ASP.NET Core Web API – Fundamentos	**04 - Dominando o Acesso a Dados com EF Core 8**	**Guilherme Paracatu**

1. A Ponte Entre Mundos

OBJETIVO: Dominar a abordagem **Database-First** com Entity Framework Core 8. Vamos aprofundar na teoria sobre ORMs, a história do EF Core, e, na prática, conectar nossa API a um banco de dados **PostgreSQL** pré-existente, usando **Data Annotations** para mapear manualmente nossas classes.

FERRAMENTAS: VS Code, Projeto LojaApi, PostgreSQL.

Essa aula vai conectar nosso código C# ao mundo real dos dados. Antes de codificarmos, é crucial entendermos a 'filosofia' e as ferramentas que nos permitem fazer essa mágica acontecer. Vamos construir uma base teórica sólida.

1.1. O Problema Fundamental: Dificuldade central e recorrente que surge ao tentar fazer dois mundos diferentes conversarem: o mundo da orientação a objetos (onde o C# vive) e o mundo dos bancos de dados relacionais (como SQL Server, PostgreSQL, etc.)

- **O Mundo dos Objetos (C#):** Nosso código pensa em **objetos**, herança e relacionamentos complexos (Cliente tem uma List<Pedido>).
- **O Mundo Relacional (Banco de Dados):** O banco de dados pensa em **tabelas**, **linhas** e **colunas**, com relacionamentos feitos por JOINs.

O **Conflito de Impedância (Impedance Mismatch)** é a dificuldade de encaixar o modelo de objetos no modelo relacional. Fazer essa "tradução" manualmente é repetitivo e propenso a erros.

1.2. A Solução: ORM (Object-Relational Mapper) Mapeia o banco de dados relacional

Um **ORM**, como o **Entity Framework Core**, é uma "ponte" que automatiza a tradução entre esses dois mundos.

Analogia do "Tradutor Universal":

> Pensem no EF Core como um **"tradutor universal"**. Nós escrevemos C# (usando LINQ), e o EF Core traduz para SQL. Quando o banco responde, ele traduz o resultado de volta para objetos C# que nosso código entende.

1.3. Aprofundando no DbContext: A Sessão de Trabalho do EF Core

Implementação de todos os métodos

O DbContext é a classe mais importante do EF Core, a nossa **"central de comando"** para o acesso a dados.

As 4 Responsabilidades Principais do DbContext:

1. **Gerenciamento da Conexão:** Abre e fecha a conexão com o banco de forma otimizada.
2. **Mapeamento de Entidades:** Lê suas classes (Cliente) e as mapeia para as tabelas do banco. O DbSet<Cliente> Clientes é a instrução que faz essa ligação.
3. **Tradução de Consultas:** Traduz suas consultas C# (LINQ) para o código SQL apropriado.
4. **Rastreamento de Alterações (Change Tracking):**
   - Quando você carrega um objeto, o DbContext tira uma "foto" do seu estado original.
   - Quando você altera uma propriedade em C#, ele compara o estado atual com a "foto" e detecta a mudança.
   - Ao chamar **SaveChanges()**, ele gera os comandos INSERT, UPDATE ou DELETE necessários em uma única transação, garantindo a consistência dos dados.

1.4. As Duas Grandes Abordagens: Code-First vs. Database-First

Característica	Code-First	Database-First (Nosso Foco Hoje) Mais utilizado
Fonte da Verdade	**O Código C#.** Suas classes ditam como o banco de dados deve ser.	**O Banco de Dados.** As tabelas existentes ditam como suas classes devem ser.
Fluxo de Trabalho	1. Você cria/modifica suas classes C#. 2. Usa **Migrations** para gerar e aplicar o SQL no banco.	1. O banco de dados já existe. 2. Você **cria ou gera** suas classes C# para espelhar o banco.
Quando Usar?	Ideal para **novos projetos**, onde se tem controle total sobre o modelo.	Ideal para se conectar a **bancos de dados legados** ou gerenciados por um time de DBAs.

Nossa Abordagem Hoje: Vamos simular o cenário **Database-First**. O banco já existe, e nossa tarefa é criar o código C# que se "adapta" a ele, usando **Data Annotations** para o mapeamento manual.

2. Mão na Massa - Conectando a um Banco Existente

Passo 1: O Cenário - O Banco de Dados Legado

Vamos imaginar que o DBA nos entregou o script da tabela de clientes, com nomes específicos que precisamos respeitar.

1. Use uma ferramenta (DBeaver, pgAdmin) para se conectar ao seu PostgreSQL.
2. Crie um novo banco de dados chamado LojaDb.
3. Execute o seguinte script SQL para criar a tabela TB_CLIENTES.

```sql
-- Script do "DBA"
CREATE TABLE "TB_CLIENTES" (
    "id_cliente" SERIAL PRIMARY KEY,
    "nome_cliente" VARCHAR(150) NOT NULL,
    "email_cliente" VARCHAR(150) NOT NULL UNIQUE,
    "ativo" BOOLEAN NOT NULL DEFAULT true,
    "data_cadastro" TIMESTAMP WITH TIME ZONE NOT NULL
);

-- Inserindo alguns dados iniciais
INSERT INTO "TB_CLIENTES" ("nome_cliente", "email_cliente", "ativo", "data_cadastro")
VALUES
('ALICE SILVA', 'alice@mail.com', true, NOW()),
('BRUNO COSTA', 'bruno@mail.com', true, NOW()),
('CARLOS SANTOS', 'carlos@mail.com', false, NOW());
```

Passo 2: Instalar os Pacotes NuGet Necessários

No terminal, na raiz do projeto LojaApi, execute:

```
dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet add package Microsoft.EntityFrameworkCore.Design
```

Passo 3: Configurar a Connection String

Em appsettings.Development.json adicione:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Host=localhost;Database=LojaDb;Username=postgres;Password=SUA_SENHA_DO_POSTGRES"
  }
}
```

Passo 4: Mapeamento Manual da Entidade com Data Annotations

Agora, vamos usar as **Data Annotations**. Elas são 'etiquetas' que colocamos em nossas classes para dizer ao EF Core como elas devem ser mapeadas para o banco, resolvendo o problema de nomes diferentes.

1. Abra o arquivo Entities/Cliente.cs e adicione as anotações.

```
// Entities/Cliente.cs
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace LojaApi.Entities
{
    // [Table] mapeia esta classe para a tabela "TB_CLIENTES".
    [Table("TB_CLIENTES")]
    public class Cliente
    {
        // [Key] indica que esta é a chave primária.
        [Key]
        // [Column] mapeia esta propriedade para a coluna "id_cliente".
        [Column("id_cliente")]
        public int Id { get; set; }

        [Column("nome_cliente")]
        [Required] // Garante que o campo é obrigatório (NOT NULL).
        [MaxLength(150)] // Mapeia para VARCHAR(150).
        public string Nome { get; set; } = string.Empty;

        [Column("email_cliente")]
        [Required]
        [MaxLength(150)]
        public string Email { get; set; } = string.Empty;

        [Column("ativo")]
        public bool Ativo { get; set; }

        [Column("data_cadastro")]
        public DateTime DataCadastro { get; set; }
    }
}
```

Passo 5: Criar o DbContext e o ClienteDBRepository

Data/LojaContext.cs

```
// Data/LojaContext.cs
using LojaApi.Entities;
using Microsoft.EntityFrameworkCore;
```

```
namespace LojaApi.Data
{
    public class LojaContext : DbContext
    {
        public LojaContext(DbContextOptions<LojaContext> options) : base(options)
        {
        }

        // O EF Core lerá as Data Annotations da classe Cliente para saber como
interagir
        // com a tabela "TB_CLIENTES" e suas colunas.
        public DbSet<Cliente> Clientes { get; set; }
    }
}
```

Um para cada tabela

Repositories/ClienteDBRepository.cs

```
// Repositories/ClienteDBRepository.cs
using LojaApi.Data;
using LojaApi.Entities;
using LojaApi.Repositories.Interfaces;
using Microsoft.EntityFrameworkCore;

namespace LojaApi.Repositories
{
    public class ClienteDBRepository : IClienteRepository
    {
        private readonly LojaContext _context;

        public ClienteDBRepository(LojaContext context)
        {
            _context = context;
        }

        public List<Cliente> ObterTodos()
        {
            return _context.Clientes.ToList();
        }

        public Cliente? ObterPorId(int id)
        {
            return _context.Clientes.FirstOrDefault(c => c.Id == id);
        }

        public Cliente Adicionar(Cliente novoCliente)
        {
            // A regra de negócio da data de cadastro fica aqui, pois é uma
responsabilidade de persistência.
            novoCliente.DataCadastro = DateTime.UtcNow;
            _context.Clientes.Add(novoCliente);
            _context.SaveChanges();
            return novoCliente;
```

```
        }

        public Cliente? Atualizar(int id, Cliente clienteAtualizado)
        {
            // O serviço já carregou e alterou a entidade. O repositório apenas
persiste.
            _context.Clientes.Update(clienteAtualizado);
            _context.SaveChanges();
            return clienteAtualizado;
        }

        public bool Remover(int id)
        {
            var clienteParaDeletar = ObterPorId(id);
            if (clienteParaDeletar == null) return false;

            _context.Clientes.Remove(clienteParaDeletar);
            _context.SaveChanges();
            return true;
        }
    }
}
```

Passo 6: Configurar a Injeção de Dependência (Program.cs)

Vamos conectar todas as peças no Program.cs.

```
// Program.cs
using LojaApi.Data;
using LojaApi.Repositories;
using LojaApi.Repositories.Interfaces;
using LojaApi.Services;
using LojaApi.Services.Interfaces;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

// >>>>> CONFIGURAÇÃO DO ENTITY FRAMEWORK CORE E DI <<<<<

// 1. Configuração do DbContext com PostgreSQL
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<LojaContext>(options =>
    options.UseNpgsql(connectionString));

// 2. Registro do Serviço
builder.Services.AddScoped<IClienteService, ClienteService>();

// 3. Registro do Repositório
builder.Services.AddScoped<IClienteRepository, ClienteDBRepository>();
```

```
// Configuração do Swagger
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// ... (restante do código)
```

3. Consultando Dados com LINQ

Agora que estamos conectados, como fazemos as consultas? Com LINQ! O EF Core traduz nosso C# para SQL.

Método LINQ	O Que Faz	Equivalente SQL (Aproximado)
`.ToList()`	Executa a consulta e retorna todos os resultados em uma lista.	`SELECT * FROM ...`
`.FirstOrDefault(c => ...)`	Retorna o primeiro elemento que satisfaz a condição, ou null.	`SELECT ... FROM ... WHERE ... LIMIT 1`
`.Where(c => ...)`	Filtra a sequência de elementos com base em uma condição.	`WHERE ...`
`.OrderBy(c => c.Nome)` esse é asc desc tem que colocar	Ordena os elementos em ordem crescente.	`ORDER BY nome_cliente ASC`
`.Select(c => ...)`	Projeta cada elemento em uma nova forma (ótimo para otimizar).	`SELECT nome_cliente, email_cliente FROM ...`
`.Any(c => ...)`	Retorna true se algum elemento satisfaz a condição (eficiente).	`EXISTS (SELECT 1 FROM ... WHERE ...)`

Parte 4: Teste e Verificação

Um ponto muito importante: nesta aula, **NÃO usamos migrations**. Por quê? Porque na abordagem Database-First, o banco de dados é a fonte da verdade. Nós não mandamos no banco, nós o obedecemos. Nosso código C# se adapta a ele.

1. **Rode a API:** dotnet run.
2. **Acesse o Swagger** e teste os *Endpoints*.

O que esperar agora:

- **GET /api/clientes:** A lista virá com os **3 clientes** que inserimos manualmente com o script SQL!
- **POST, PUT, DELETE:** Todas as operações agora interagem diretamente com a tabela TB_CLIENTES.
- **Verificação:** Use uma ferramenta como **DBeaver** ou **pgAdmin** para verificar que as alterações feitas via API estão sendo refletidas na sua tabela.

Na próxima aula: Vamos aprofundar no EF Core, explorando **relacionamentos** entre entidades (um-para-muitos, como Produtos e Clientes) e como o EF Core lida com chaves estrangeiras.