FMX Academy

INFORMAÇÕES			
MÓDULO		**AULA**	**INSTRUTOR**
05	**ASP.NET Core Web API – Fundamentos**	**06 - EF Core e boas práticas: Fluent API, Relacionamentos 1:1 e DTOs**	**Guilherme Paracatu**

1. Aprofundamento Teórico - Arquitetura de Dados

OBJETIVO: Dominar o mapeamento de **relacionamentos 1:1**, aprofundar na configuração do EF Core com **Fluent API**, entender o padrão **DTO (Data Transfer Object)**, por que ele é crucial para APIs robustas, e como ele resolve o problema de **referência cíclica.**

FERRAMENTAS: VS Code, Projeto LojaApi, PostgreSQL.

Parte 1: Aprofundamento Teórico - Arquitetura de Dados Profissional (Aprox. 45 minutos)

Bem-vindos a uma aula fundamental. Hoje, vamos dar um passo além da simples conexão com o banco de dados e aprender as técnicas que separam uma API funcional de uma API profissional, segura e de fácil manutenção. Vamos falar sobre como mapear relacionamentos de forma avançada e por que **nunca** devemos expor nossas entidades de banco de dados diretamente.

1.1. O Grande Problema de Expor Entidades: A Referência Cíclica e Riscos de Segurança

Até agora, retornamos nossas entidades (Cliente, Produto, Categoria) diretamente nos endpoints. Isso é simples, mas é uma péssima prática que pode quebrar sua API e expor seus dados. Vamos entender os dois principais problemas.

Problema 1: A Referência Cíclica loop infinito

- **O Cenário:**
  - A classe Cliente terá uma propriedade de navegação public Endereco Endereco { get; set; }.
  - A classe Endereco terá uma propriedade de navegação de volta public Cliente Cliente { get; set; }.
- **O Problema:** Quando o ASP.NET Core tenta serializar isso para JSON:
  1. Ele começa a serializar o Cliente.
  2. Dentro do Cliente, ele encontra a propriedade Endereco e começa a serializá-la.

3. Dentro do Endereco, ele encontra a propriedade de volta Cliente e tenta serializar o Cliente novamente.
4. Isso cria um **loop infinito!**

Analogia do "Espelho Infinito":

É como colocar dois espelhos um de frente para o outro. A imagem se reflete infinitamente. O serializador JSON não consegue lidar com isso e lança uma exceção, quebrando sua API.

Problema 2: Riscos de Segurança e Contrato Frágil

- **Segurança:** Se sua entidade Usuario tivesse uma propriedade SenhaHash ou IsAdmin, e você a retornasse diretamente, estaria expondo dados críticos na sua API.
- **Contrato Frágil:** Se você mudar a entidade no banco (ex: renomear uma coluna), sua API quebra. A API fica "amarrada" à estrutura do banco.

1.2. A Solução Profissional: DTOs (Data Transfer Objects)

A solução correta e padrão do mercado é usar **DTOs**. Um DTO é uma classe simples cujo único propósito é **transferir dados** e definir o **contrato** da sua API.

Organiza oq vai ser usado das entities, é o único que pega dados delas

Analogia do "Formulário de Pedido" vs. "Manual Técnico":

Pense na sua entidade Produto como o **manual técnico completo** do carro. O DTO, como ProdutoDto, é o **folheto de vendas** que você entrega ao cliente. Ele só contém as informações que o cliente **precisa e pode ver**, formatadas de uma maneira bonita.

As 4 Grandes Vantagens de Usar DTOs:

1. **Resolve a Referência Cíclica:** Você projeta o DTO para conter apenas os dados necessários, quebrando o loop.
2. **Segurança:** Você controla exatamente quais dados são expostos.
3. **Contrato de API Estável:** Você pode mudar sua entidade de banco de dados sem quebrar os clientes da sua API.
4. **Flexibilidade:** Você pode criar DTOs específicos para cada caso de uso (ClienteResumoDto, ClienteDetalhadoDto).

1.3. Aprofundando no Mapeamento: Fluent API vs. Data Annotations

Para mapear nossas classes para o banco, temos duas técnicas: Data Annotations e Fluent API. As anotações são simples, mas a Fluent API é a abordagem mais poderosa e limpa.

Característica	Data Annotations ([Table], [Column])	Fluent API (OnModelCreating)
Localização	"Sujam" a classe de entidade com detalhes de persistência.	Centraliza toda a configuração de mapeamento no DbContext.
Poder	Limitada para configurações básicas.	Extremamente poderosa. Permite configurar índices, chaves compostas, relacionamentos complexos, etc.
Separação de Interesses	Mistura as regras do domínio (a entidade) com as regras de persistência.	Mantém a entidade "limpa" (POCO - Plain Old CLR Object), separando as responsabilidades.
Quando Usar?	Projetos simples, prototipagem rápida.	**Abordagem preferida** para projetos profissionais e arquiteturas limpas.

As entities ficam dependendo da extensão:
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Sche

Conclusão: Vamos refatorar nosso projeto para usar **Fluent API**, centralizando todo o mapeamento no OnModelCreating e deixando nossas entidades puras.

2. Mão na Massa - Implementando Relacionamentos e DTOs

Passo 1: O Cenário - O Banco de Dados Existente

1. Use uma ferramenta (DBeaver, pgAdmin) para se conectar ao seu PostgreSQL.
2. Execute o seguinte script SQL no seu banco LojaDb para criar a tabela de endereços.

```
-- Relacionamento 1:1 com Clientes
CREATE TABLE "TB_ENDERECOS" (
    "id_cliente" INT PRIMARY KEY, -- Mesma chave primária de Clientes
    "rua" VARCHAR(200) NOT NULL,
    "cidade" VARCHAR(100) NOT NULL,
    "estado" VARCHAR(50) NOT NULL,
    "cep" VARCHAR(10) NOT NULL,
    CONSTRAINT "fk_cliente_endereco" FOREIGN KEY ("id_cliente") REFERENCES
"TB_CLIENTES"("id_cliente")
);
```

Passo 2: Mapeamento "Limpo" das Entidades (Sem Data Annotations)

Vamos refatorar nossas entidades para remover as anotações de mapeamento. Elas agora serão classes "puras".

Entities/Endereco.cs

```
// Entities/Endereco.cs
namespace LojaApi.Entities
{
    // Classe pura, sem anotações de mapeamento.
    public class Endereco
    {
        public int Id { get; set; } // Por convenção, o EF Core entende que esta é a PK
        public string Rua { get; set; } = string.Empty;
        public string Cidade { get; set; } = string.Empty;
        public string Estado { get; set; } = string.Empty;
        public string Cep { get; set; } = string.Empty;
        public Cliente? Cliente { get; set; }
    }
}
```

Entities/Cliente.cs

```
// Entities/Cliente.cs
namespace LojaApi.Entities
{
    // Classe pura, sem anotações de mapeamento.
    public class Cliente
    {
        public int Id { get; set; }
        public string Nome { get; set; } = string.Empty;
        public string Email { get; set; } = string.Empty;
        public bool Ativo { get; set; }
        public DateTime DataCadastro { get; set; }

        // Propriedade de Navegação para o relacionamento 1:1
        public Endereco? Endereco { get; set; }
    }
}
```

Passo 3: Criar os DTOs para um Contrato de API Limpo

Crie a pasta DTOs e os seguintes arquivos.

DTOs/EnderecoDto.cs

```
// DTOs/EnderecoDto.cs
using System.ComponentModel.DataAnnotations;
namespace LojaApi.DTOs
{
    public class EnderecoDto
    {
        [Required]
        public string Rua { get; set; } = string.Empty;
        [Required]
        public string Cidade { get; set; } = string.Empty;
        [Required]
        public string Estado { get; set; } = string.Empty;
        [Required]
        public string Cep { get; set; } = string.Empty;
    }
}
```

DTOs/CriarClienteDto.cs

```
// DTOs/CriarClienteDto.cs
using System.ComponentModel.DataAnnotations;
namespace LojaApi.DTOs
{
    public class CriarClienteDto
    {
        [Required]
        public string Nome { get; set; } = string.Empty;
        [Required]
        [EmailAddress]
        public string Email { get; set; } = string.Empty;
        public EnderecoDto? Endereco { get; set; }
    }
}
```

DTOs/ClienteDetalhadoDto.cs

```
// DTOs/ClienteDetalhadoDto.cs
namespace LojaApi.DTOs
{
    public class ClienteDetalhadoDto
    {
        public int Id { get; set; }
        public string Nome { get; set; } = string.Empty;
        public string Email { get; set; } = string.Empty;
        public bool Ativo { get; set; }
        public EnderecoDto? Endereco { get; set; }
    }
}
```

Passo 4: Centralizando o Mapeamento no DbContext com Fluent API

Agora, vamos usar o OnModelCreating para configurar todo o mapeamento.

Data/LojaContext.cs (com Fluent API)

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

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

        public DbSet<Cliente> Clientes { get; set; }
        public DbSet<Endereco> Enderecos { get; set; }
        // ... outros DbSets ...

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            // --- MAPEAMENTO DA ENTIDADE CLIENTE ---
            modelBuilder.Entity<Cliente>(entity =>
            {
                // Mapeia para a tabela TB_CLIENTES
                entity.ToTable("TB_CLIENTES");

                // Define a chave primária e o nome da coluna
                entity.HasKey(c => c.Id).HasName("id_cliente_pk"); // Nome opcional
```

```
para a constraint
                entity.Property(c => c.Id).HasColumnName("id_cliente");

                // Mapeia as outras propriedades para suas colunas
                entity.Property(c =>
c.Nome).HasColumnName("nome_cliente").HasMaxLength(150).IsRequired();
                entity.Property(c =>
c.Email).HasColumnName("email_cliente").HasMaxLength(150).IsRequired();
                entity.Property(c => c.Ativo).HasColumnName("ativo");
                entity.Property(c => c.DataCadastro).HasColumnName("data_cadastro");
            });

            // --- MAPEAMENTO DA ENTIDADE ENDERECO ---
            modelBuilder.Entity<Endereco>(entity =>
            {
                entity.ToTable("TB_ENDERECOS");
                entity.HasKey(e => e.Id);
                entity.Property(e => e.Id).HasColumnName("id_cliente");
                entity.Property(e =>
e.Rua).HasColumnName("rua").HasMaxLength(200).IsRequired();
                entity.Property(e =>
e.Cidade).HasColumnName("cidade").HasMaxLength(100).IsRequired();
                entity.Property(e =>
e.Estado).HasColumnName("estado").HasMaxLength(50).IsRequired();
                entity.Property(e =>
e.Cep).HasColumnName("cep").HasMaxLength(10).IsRequired();
            });

            // --- CONFIGURAÇÃO DO RELACIONAMENTO 1:1 ---
            modelBuilder.Entity<Cliente>()
                .HasOne(c => c.Endereco)      // Um Cliente tem um Endereço
                .WithOne(e => e.Cliente)      // Um Endereço tem um Cliente
                .HasForeignKey<Endereco>(e => e.Id); // A chave estrangeira está em
Endereco.Id
        }
    }
}
```

Passo 5: Refatorar o Service e o Controller para usar DTOs

Services/ClienteService.cs

```
// Em Services/IClienteService.cs, adicione:
Cliente Adicionar(CriarClienteDto clienteDto);
ClienteDetalhadoDto? ObterDetalhesPorId(int id);

// Em Services/ClienteService.cs
public Cliente Adicionar(CriarClienteDto clienteDto)
```

```
{
    var novoCliente = new Cliente
    {
        Nome = clienteDto.Nome.ToUpper(),
        Email = clienteDto.Email,
        Ativo = true,
        DataCadastro = DateTime.UtcNow,
        Endereco = clienteDto.Endereco != null ? new Endereco
        {
            Rua = clienteDto.Endereco.Rua,
            Cidade = clienteDto.Endereco.Cidade,
            Estado = clienteDto.Endereco.Estado,
            Cep = clienteDto.Endereco.Cep
        } : null
    };
    return _clienteRepository.Adicionar(novoCliente);
}

public ClienteDetalhadoDto? ObterDetalhesPorId(int id)
{
    var cliente = _clienteRepository.ObterPorId(id);
    if (cliente == null) return null;

    return new ClienteDetalhadoDto
    {
        Id = cliente.Id,
        Nome = cliente.Nome,
        Email = cliente.Email,
        Ativo = cliente.Ativo,
        Endereco = cliente.Endereco != null ? new EnderecoDto
        {
            Rua = cliente.Endereco.Rua,
            Cidade = cliente.Endereco.Cidade,
            Estado = cliente.Endereco.Estado,
            Cep = cliente.Endereco.Cep
        } : null
    };
}
```

Controllers/ClientesController.cs

```
// Em Controllers/ClientesController.cs

// GET api/Clientes/{id}
[HttpGet("{id}")]
public ActionResult<ClienteDetalhadoDto> GetById(int id)
{
    var clienteDto = _clienteService.ObterDetalhesPorId(id);
    if (clienteDto == null) return NotFound();
```

```
    return Ok(clienteDto);
}

// POST api/Clientes
[HttpPost]
public ActionResult<ClienteDetalhadoDto> Add(CriarClienteDto clienteDto)
{
    var clienteCriado = _clienteService.Adicionar(clienteDto);
    var dtoRetorno = _clienteService.ObterDetalhesPorId(clienteCriado.Id);
    return CreatedAtAction(nameof(GetById), new { id = clienteCriado.Id }, dtoRetorno);
}
```

Parte 4: Teste e Verificação

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

O que testar:

- **POST /api/clientes:** Envie um JSON com um objeto endereco aninhado.
  JSON
  { "nome": "Fernanda Lima", "email": "fernanda@mail.com", "endereco": { "rua": "Av. Principal", "cidade": "São Paulo", "estado": "SP", "cep": "01000-000" } }

- **GET /api/clientes/{id}:** Verifique se o cliente criado acima é retornado com seu endereço, mas sem a propriedade de navegação de volta para Cliente, evitando a referência cíclica.