FMX Academy

INFORMAÇÕES		
MÓDULO	**AULA**	**INSTRUTOR**
05 ASP.NET Core Web API – Fundamentos	**05 - Aprofundando no EF Core: Relacionamentos, Performance e Validação**	**Guilherme Paracatu**

1. Problema de performance N+1

OBJETIVO: Dominar o mapeamento de **relacionamentos**, entender e resolver o **problema de performance N+1** com **Eager Loading**, e implementar o **padrão de validação moderno** do ASP.NET Core usando **Data Annotations** e entendendo a arquitetura de validação.

FERRAMENTAS: VS Code, Projeto LojaApi, PostgreSQL.

Parte 1: Aprofundamento Teórico - Dados Conectados, Eficientes e Seguros

Na última aula, conectamos nossa API a um banco de dados. Hoje, vamos aprender a fazer isso da forma certa: com **performance** e **segurança**. Vamos desvendar um dos maiores vilões de performance em ORMs, o 'problema N+1', modelar dados que se relacionam e aprender como o ASP.NET Core automatiza a validação para nós.

1.1. Modelando o Mundo Real: Relacionamentos

No mundo real, os dados não vivem isolados. Um Produto pertence a uma Categoria. O EF Core nos ajuda a modelar esses relacionamentos usando dois conceitos:

1. **Chave Estrangeira (Foreign Key):** Uma propriedade na classe "filha" (Produto) que armazena o ID da classe "pai" (Categoria). Ex: public int CategoriaId { get; set; }.
2. **Propriedades de Navegação (Navigation Properties):** As "pontes" que nos permitem navegar entre objetos em C#.
   - **Lado "Um" (Categoria):** Terá uma coleção dos filhos. Ex: public ICollection<Produto> Produtos { get; set; }.
   - **Lado "Muitos" (Produto):** Terá uma referência de volta para o pai. Ex: public Categoria Categoria { get; set; }.

1.2. O Problema Silencioso de Performance: N+1

O "Porquê" do Nome N+1 e o Vilão "Lazy Loading"

O nome 'N+1' descreve exatamente o que acontece. Imagine que você tem **N** itens principais e, para cada um, precisa buscar um dado relacionado. Isso resulta em **N** consultas adicionais, somadas à **1** consulta original. O causador desse comportamento,

por padrão em muitos ORMs, é uma funcionalidade chamada **Lazy Loading** (Carregamento Preguiçoso).

- **O que é Lazy Loading?** É quando o EF Core não carrega os dados relacionados imediatamente. Ele espera até que você tente acessá-los no código. Parece eficiente, mas é uma armadilha perigosa dentro de loops.

- **O Cenário:** Você quer listar **5 categorias** e, para cada uma, mostrar a lista de seus produtos.

- **A Abordagem Ineficiente (O Problema N+1 causado pelo Lazy Loading):**
  1. **Consulta (a consulta "1"):** O EF Core busca as 5 categorias.
     - SELECT * FROM "TB_CATEGORIAS";

  2. **N Consultas Adicionais (as consultas "+N"):** Dentro de um loop, ao tentar acessar categoria.Produtos pela primeira vez para cada uma das **N** categorias, o Lazy Loading é acionado e dispara uma nova consulta ao banco para buscar os produtos daquela categoria específica.
     - SELECT * FROM "TB_PRODUTOS" WHERE "id_categoria" = 1;
     - SELECT * FROM "TB_PRODUTOS" WHERE "id_categoria" = 2;
     - SELECT * FROM "TB_PRODUTOS" WHERE "id_categoria" = 3;
     - ... e assim por diante, para cada uma das 5 categorias.

  3. **Total de Consultas: 1 (categorias) + N (produtos por categoria) = 1 + 5 = 6 consultas!** Se fossem 100 categorias, seriam 101 consultas ao banco, deixando a aplicação extremamente lenta.

Exemplo de Código C# que Causa o Problema N+1 (com Lazy Loading habilitado):

```
// CÓDIGO INEFICIENTE - NÃO FAÇA ISSO!
public void ExibirProdutosPorCategoria()
{
    // 1ª CONSULTA: Busca todas as categorias
    var categorias = _context.Categorias.ToList();

    foreach (var categoria in categorias)
    {
        Console.WriteLine($"Categoria: {categoria.Nome}");

        // N CONSULTAS: O acesso a 'categoria.Produtos' dispara uma nova consulta ao banco para cada categoria!
        foreach (var produto in categoria.Produtos)
        {
            Console.WriteLine($" - Produto: {produto.Nome}");
        }
    }
}
```

1.3. A Solução: Carregamento Ansioso (Eager Loading) com .Include()

A solução é ser 'ansioso' e dizer ao EF Core de antemão: 'Ei, quando for buscar as categorias, já traga os produtos delas junto!'. Isso desabilita o Lazy Loading para esta consulta específica e usa o **Eager Loading**.

- **O Método .Include():** Usamos o método .Include() para instruir o EF Core a carregar os dados relacionados na mesma consulta inicial.
- **O Resultado:** O EF Core gera um único e eficiente comando SQL com JOIN, trazendo todos os dados necessários em **uma única consulta** ao banco.

Exemplo de Código C# com a Solução:

```
// CÓDIGO EFICIENTE - FAÇA ISSO!
public List<Categoria> ObterCategoriasComProdutos()
{
    // UMA ÚNICA CONSULTA: Graças ao .Include(), o EF Core gera um JOIN.
    return _context.Categorias
        .Include(c => c.Produtos) // "Inclua a propriedade de navegação Produtos"
        .ToList();
}
```

1.4. A Arquitetura da Validação: Onde e Por Que?

Um dos tópicos mais importantes em arquitetura de software é decidir **onde** colocar a lógica de validação. A resposta correta é: **em camadas, com responsabilidades diferentes.**

A Arquitetura Correta da Validação

1. **Validação de Modelo/Entidade (Camada de Entrada):**
    - **O quê:** Valida o **formato**, a **estrutura** e as **regras básicas** dos dados de entrada. Ex: "O e-mail tem um formato válido?", "O nome tem mais de 3 caracteres?".
    - **Como:** Usando **Data Annotations** ([Required], [StringLength], etc.) nas propriedades das nossas classes de Entidade (ou DTOs).
    - **Quem executa:** O **ASP.NET Core**, automaticamente, graças ao atributo [ApiController]. Ele rejeita a requisição com um 400 Bad Request antes mesmo de chegar ao seu código.
2. **Validação de Negócio (Camada de Serviço):**
    - **O quê:** Valida regras que dependem do estado atual do sistema ou de outras entidades (requer acesso ao banco de dados). Ex: "Este e-mail já está cadastrado no sistema?", "Um cliente VIP pode ter um desconto maior que 20%?".
    - **Como:** Com lógica if dentro dos métodos do serviço.

Conclusão: A validação de formato fica nas **Data Annotations** (executada pelo framework), e a validação de negócio fica na **camada de Serviço**. O Controller permanece limpo.

2. Mão na Massa - Implementando Relacionamentos e Validação

Passo 1: O Cenário - Evoluindo o Banco de Dados

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:

```sql
-- Script para criar as tabelas com relacionamentos
CREATE TABLE "TB_CATEGORIAS" (
    "id_categoria" SERIAL PRIMARY KEY,
    "nome_categoria" VARCHAR(100) NOT NULL UNIQUE
);
CREATE TABLE "TB_PRODUTOS" (
    "id_produto" SERIAL PRIMARY KEY,
    "nome_produto" VARCHAR(150) NOT NULL,
    "preco_produto" DECIMAL(10, 2) NOT NULL,
    "estoque_produto" INT NOT NULL,
    "id_categoria" INT NOT NULL,
    CONSTRAINT "fk_categoria" FOREIGN KEY ("id_categoria") REFERENCES
"TB_CATEGORIAS"("id_categoria")
);
INSERT INTO "TB_CATEGORIAS" ("nome_categoria") VALUES ('Eletrônicos'), ('Livros'),
('Vestuário');
INSERT INTO "TB_PRODUTOS" ("nome_produto", "preco_produto", "estoque_produto",
"id_categoria") VALUES
('Smartphone XYZ', 1999.99, 50, 1),
('Notebook Gamer ABC', 7500.00, 15, 1),
('O Senhor dos Anéis', 120.50, 100, 2),
('Camiseta Básica', 49.90, 200, 3);
```

Passo 2: Mapeamento e Validação das Entidades

Vamos adicionar as Data Annotations de mapeamento e validação de formato diretamente em nossas entidades.

Entities/Categoria.cs

```csharp
// Entities/Categoria.cs
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;
```

```
namespace LojaApi.Entities
{
    [Table("TB_CATEGORIAS")]
    public class Categoria
    {
        [Key]
        [Column("id_categoria")]
        public int Id { get; set; }

        [Column("nome_categoria")]
        [Required(ErrorMessage = "O nome da categoria é obrigatório.")]
        [StringLength(100, MinimumLength = 3, ErrorMessage = "O nome da categoria deve
ter entre 3 e 100 caracteres.")]
        public string Nome { get; set; } = string.Empty;

        [JsonIgnore]
        public ICollection<Produto> Produtos { get; set; } = new List<Produto>();
    }
}
```

Entities/Produto.cs

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

namespace LojaApi.Entities
{
    [Table("TB_PRODUTOS")]
    public class Produto
    {
        [Key]
        [Column("id_produto")]
        public int Id { get; set; }

        [Column("nome_produto")]
        [Required(ErrorMessage = "O nome do produto é obrigatório.")]
        [StringLength(150, MinimumLength = 3, ErrorMessage = "O nome deve ter entre 3 e
150 caracteres.")]
        public string Nome { get; set; } = string.Empty;

        [Column("preco_produto", TypeName = "decimal(10, 2)")]
        [Required(ErrorMessage = "O preço é obrigatório.")]
        [Range(0.01, 100000.00, ErrorMessage = "O preço deve ser entre 0.01 e
100,000.00.")]
        public decimal Preco { get; set; }

        [Column("estoque_produto")]
        [Range(0, 9999, ErrorMessage = "O estoque deve ser entre 0 e 9999.")]
```

```
        public int Estoque { get; set; }

        [Column("id_categoria")]
        [Required(ErrorMessage = "O ID da categoria é obrigatório.")]
        public int CategoriaId { get; set; }

        [ForeignKey("CategoriaId")]
        public Categoria? Categoria { get; set; }
    }
}
```

Passo 3: Atualizar o DbContext, Repositórios e Serviços (com Validação de Negócio)

Data/LojaContext.cs

```
// Data/LojaContext.cs
// Adicione os novos DbSets
public DbSet<Categoria> Categorias { get; set; }
public DbSet<Produto> Produtos { get; set; }
```

Repositories/ProdutoDBRepository.cs (com .Include())

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

namespace LojaApi.Repositories
{
    public class ProdutoDBRepository : IProdutoRepository
    {
        private readonly LojaContext _context;
        public ProdutoDBRepository(LojaContext context) { _context = context; }

        public List<Produto> ObterTodos()
        {
            // Eager Loading: Traz os dados da Categoria junto com os Produtos.
            return _context.Produtos.Include(p => p.Categoria).ToList();
        }

        public Produto? ObterPorId(int id)
        {
            return _context.Produtos.Include(p => p.Categoria).FirstOrDefault(p => p.Id
```

```
== id);
        }

        public Produto Adicionar(Produto novoProduto)
        {
            _context.Produtos.Add(novoProduto);
            _context.SaveChanges();
            return novoProduto;
        }
    }
}
```

Services/ProdutoService.cs (com Validação de Negócio)

```
// Services/ProdutoService.cs
using LojaApi.Entities;
using LojaApi.Repositories.Interfaces;

namespace LojaApi.Services
{
    public class ProdutoService : IProdutoService
    {
        private readonly IProdutoRepository _produtoRepository;
        private readonly ICategoriaRepository _categoriaRepository; // Dependência para
validar a categoria

        public ProdutoService(IProdutoRepository produtoRepository,
ICategoriaRepository categoriaRepository)
        {
            _produtoRepository = produtoRepository;
            _categoriaRepository = categoriaRepository;
        }

        public List<Produto> ObterTodos()
        {
            return _produtoRepository.ObterTodos().Where(p => p.Estoque > 0).ToList();
        }

        public Produto? ObterPorId(int id)
        {
            return _produtoRepository.ObterPorId(id);
        }

        public Produto Adicionar(Produto novoProduto)
        {
            // Validação de Negócio: Regras que precisam de acesso a dados.
            var categoria = _categoriaRepository.ObterPorId(novoProduto.CategoriaId);
            if (categoria == null)
            {
```

```
                // Em um projeto real, lançaríamos uma exceção customizada que vamos
ver mais à frente no curso.
                // Por simplicidade, podemos retornar null ou uma mensagem.
                throw new Exception("A categoria informada não existe.");
            }

            if (categoria.Nome.Equals("Eletrônicos",
StringComparison.OrdinalIgnoreCase) && novoProduto.Preco < 50.00m)
            {
                throw new Exception("Produtos da categoria 'Eletrônicos' devem custar
no mínimo R$ 50,00.");
            }

            return _produtoRepository.Adicionar(novoProduto);
        }
    }
}
```

Crie também as interfaces e implementações para a Categoria, seguindo o mesmo padrão.

Passo 4: Configurar a Injeção de Dependência e o ProdutosController

Program.cs

```
// Program.cs
// ...
builder.Services.AddScoped<IProdutoService, ProdutoService>();
builder.Services.AddScoped<IProdutoRepository, ProdutoDBRepository>();
builder.Services.AddScoped<ICategoriaService, CategoriaService>();
builder.Services.AddScoped<ICategoriaRepository, CategoriaDBRepository>();
// ...
```

Controllers/ProdutosController.cs (com tratamento de exceção de negócio)

```
// Controllers/ProdutosController.cs
using LojaApi.Entities;
using LojaApi.Services.Interfaces; // Use a interface
using Microsoft.AspNetCore.Mvc;

namespace LojaApi.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ProdutosController : ControllerBase
    {
        private readonly IProdutoService _produtoService;
        public ProdutosController(IProdutoService produtoService) { _produtoService =
produtoService; }
```

```
        [HttpGet]
        public ActionResult<List<Produto>> GetAll()
        {
            return Ok(_produtoService.ObterTodos());
        }

        [HttpGet("{id}")]
        public ActionResult<Produto> GetById(int id)
        {
            var produto = _produtoService.ObterPorId(id);
            if (produto == null) return NotFound();
            return Ok(produto);
        }

        [HttpPost]
        public ActionResult<Produto> Add(Produto novoProduto)
        {
            // A validação de formato é automática!
            // Agora, tratamos os erros de negócio que podem vir do serviço.
            try
            {
                var produtoAdicionado = _produtoService.Adicionar(novoProduto);
                return CreatedAtAction(nameof(GetById), new { id = produtoAdicionado.Id
}, produtoAdicionado);
            }
            catch (Exception ex)
            {
                // Retorna 400 Bad Request com a mensagem de erro de negócio.
                return BadRequest(ex.Message);
            }
        }
    }
}
```

Parte 5: Tabela de Referência - Data Annotations de Validação Aprofundada

Anotação	O Que Valida	Exemplo de Uso
[Required]	Garante que a propriedade não seja nula. Para strings, não pode ser nula ou vazia.	[Required(ErrorMessage = "O nome é obrigatório.")]
[StringLength]	Define um comprimento máximo e, opcionalmente, mínimo para uma string.	[StringLength(100, MinimumLength = 3)]

[Range]	Garante que um valor numérico esteja dentro de um intervalo.	[Range(18, 99, ErrorMessage = "A idade deve ser entre 18 e 99.")]
[EmailAddress]	Valida se a string tem um formato de e-mail válido. **Não é uma validação perfeita**; ele verifica a estrutura básica (algo@dominio.com), mas não a existência do domínio.	[EmailAddress(ErrorMessage = "Formato de e-mail inválido.")]
[Phone]	Valida se a string tem um formato de telefone. **Não é específico de uma região.** Ele aceita vários formatos internacionais. **Não valida o formato brasileiro ((xx) xxxxx-xxxx) por padrão.** Para isso, usaríamos [RegularExpression].	[Phone]
[Compare]	Compara o valor da propriedade com o de outra. Ideal para confirmação de senha.	public string Senha { get; set; } [Compare("Senha", ErrorMessage = "As senhas não coincidem.")] public string ConfirmarSenha { get; set; }
[RegularExpression]	Valida a string contra uma expressão regular (regex), permitindo validações complexas e customizadas.	[RegularExpression(@"^\d{5}-\d{3}$", ErrorMessage = "CEP inválido.")]

Parte 6: Teste e Verificação

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

O que esperar agora:

- **GET /api/produtos:** A resposta será uma lista de produtos, com o objeto categoria aninhado.
- **POST /api/produtos:**
  - Tente enviar um JSON **sem o campo nome**. Deve retornar 400 Bad Request.
  - Tente enviar um JSON com um preco de 40.00 e categoriaId de 1 (Eletrônicos). Deve retornar 400 Bad Request com a mensagem da nossa **validação de negócio**.
  - Envie um JSON válido e observe o produto sendo criado com sucesso (201 Created).

Na próxima aula: APIs robustas com DTOs e Middlewares

Vamos aprender a utilizar Data Transfer Objects (DTOs) para validar e modelar os dados de nossas APIs de forma segura e eficiente. Além disso, veremos como criar um tratamento de exceções global e centralizado com Middlewares, tornando nosso código mais limpo e de fácil manutenção.