**Índice da Aula: Evoluindo uma API de Produtos e Categorias**

* **Capítulo 1: A Fundação - Projeto, Modelos e DbContext**
  + Objetivo: Configurar o projeto Web API, criar os modelos Produto e Categoria, configurar o DbContext e gerar o banco de dados com o Entity Framework.
* **Capítulo 2: A Abordagem Direta (e seus Problemas)**
  + Objetivo: Criar um ProdutosController que usa o DbContext diretamente para criar um produto e listar produtos por categoria.
  + Ponto de Atenção: Identificar e explicar o erro de referência circular (JsonException) que ocorrerá ao listar categorias com seus produtos.
* **Capítulo 3: Introduzindo o Padrão Repository (IRepository)**
  + Objetivo: Abstrair o acesso a dados. Criaremos uma interface IRepository genérica e interfaces específicas (IProdutoRepository, ICategoriaRepository) e suas implementações, refatorando o Controller para não depender mais diretamente do DbContext.
* **Capítulo 4: Resolvendo o Problema de Serialização com DTOs (Data Transfer Objects)**
  + Objetivo: Criar objetos específicos para transferência de dados (ProdutoDTO, CategoriaDTO, CriarProdutoDTO). Com isso, resolvemos o problema da referência circular e passamos a expor apenas os dados necessários para o cliente da API.
* **Capítulo 5: Garantindo a Integridade com Unit of Work (UoW)**
  + Objetivo: Introduzir o padrão Unit of Work para garantir que operações que envolvem múltiplos repositórios (como cadastrar um produto e talvez atualizar a categoria ao mesmo tempo) sejam executadas em uma única transação, garantindo a consistência dos dados.

**Capítulo 1: A Fundação - Projeto, Modelos e DbContext**

**Objetivo:** Neste primeiro capítulo, vamos preparar todo o terreno. Criaremos a estrutura básica do nosso projeto, definiremos como são nossos dados (produtos e categorias) e configuraremos o Entity Framework para criar e gerenciar nosso banco de dados.

**Passo 1: Criando o Projeto Web API**

WebApi

Solução: "GerenciadorDeProdutos"

**Passo 2: Instalando os Pacotes do Entity Framework Core**

Precisamos de alguns pacotes (NuGet) para trabalhar com o EF Core e o SQL Server.

package Microsoft.EntityFrameworkCore

package Microsoft.EntityFrameworkCore.SqlServer

package Microsoft.EntityFrameworkCore.Tools

**Passo 3: Criando as Classes de Modelo (Entities)**

Vamos criar uma pasta chamada Models na raiz do projeto e adicionar nossas duas classes principais.

**Models/Categoria.cs**

|  |
| --- |
| // Models/Categoria.cs  namespace GerenciadorDeProdutos.Models;  public class Categoria  {  public int Id { get; set; }  public string Nome { get; set; }  // Propriedade de navegação: Uma categoria pode ter vários produtos.  // É ESSENCIAL para o EF Core entender a relação.  // ATENÇÃO: Essa linha será a causa do nosso futuro erro de serialização.  public ICollection<Produto> Produtos { get; set; }  } |

**Models/Produto.cs**

|  |
| --- |
| // Models/Produto.cs  using System.ComponentModel.DataAnnotations.Schema;  namespace GerenciadorDeProdutos.Models;  public class Produto  {  public int Id { get; set; }  public string Nome { get; set; }  public decimal Preco { get; set; }  // Chave estrangeira para a Categoria  public int CategoriaId { get; set; }  // Propriedade de navegação: Um produto pertence a uma categoria.  [ForeignKey("CategoriaId")]  public Categoria Categoria { get; set; }  } |

**Passo 4: Criando o Contexto do Banco de Dados (DbContext)**

O DbContext é a ponte entre nossos modelos C# e o banco de dados. Crie uma pasta chamada Data e adicione a classe abaixo.

**Data/AppDbContext.cs**

|  |
| --- |
| // Data/AppDbContext.cs  using GerenciadorDeProdutos.Models;  using Microsoft.EntityFrameworkCore;  namespace GerenciadorDeProdutos.Data;  public class AppDbContext : DbContext  {  public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)  {  }  // Mapeia o modelo Categoria para uma tabela "Categorias" no banco  public DbSet<Categoria> Categorias { get; set; }  // Mapeia o modelo Produto para uma tabela "Produtos" no banco  public DbSet<Produto> Produtos { get; set; }  } |

**Passo 5: Configurando a String de Conexão**

Abra o arquivo appsettings.json e adicione a string de conexão com o seu banco SQL Server.

|  |
| --- |
| ,  "ConnectionStrings": {  "DefaultConnection": "Server=SERVIDOR;Database=GrenciadorDB;Trusted\_Connection=True;TrustServerCertificate=True;MultipleActiveResultSets=true"  } |

**Atenção:** Substitua SEU\_SERVIDOR, SEU\_USUARIO e SUA\_SENHA pelos dados do seu ambiente SQL Server.

**Passo 6: Registrando o DbContext no Container de Injeção de Dependência**

Agora, precisamos "ensinar" nossa aplicação a criar e gerenciar o AppDbContext. Fazemos isso no arquivo Program.cs.

**Program.cs**

|  |
| --- |
| using GerenciadorDeProdutos.Data; // Adicione este using  using Microsoft.EntityFrameworkCore; // Adicione este using  var builder = WebApplication.CreateBuilder(args);  // Pega a string de conexão do appsettings.json  var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");  // Adiciona o AppDbContext ao container de serviços.  // Isso permite que ele seja injetado em outras partes da aplicação (como os controllers).  builder.Services.AddDbContext<AppDbContext>(options =>  options.UseSqlServer(connectionString));  builder.Services.AddControllers();  // ... resto do código (Swagger, etc)  var app = builder.Build();  // ... resto do código  app.Run(); |

**Passo 7: Criando o Banco de Dados com Migrations**

Com tudo configurado, vamos usar as ferramentas do EF Core para ler nossos modelos e gerar o script SQL que cria o banco e as tabelas.

No terminal, na raiz do projeto, execute os seguintes comandos:

1. **Criar a migração:** Este comando cria os arquivos C# que representam as alterações no banco.

dotnet ef migrations add "Inicial"

ou

add-migration Initial

1. **Aplicar a migração:** Este comando executa a migração, criando de fato o banco de dados e as tabelas.

dotnet ef database update

ou

update-database

Se tudo correu bem, você agora tem um banco de dados com nome que sugeriu o appsettings e com as tabelas Categorias e Produtos prontas para uso.

**Resumo do Capítulo:** Neste momento, temos um projeto Web API funcional com toda a base de acesso a dados configurada. Nossas classes de modelo estão criadas, o DbContext está pronto e o banco de dados físico existe.

No **Capítulo 2**, vamos criar nossos primeiros endpoints e encontrar nosso primeiro (e muito didático) problema.

**Capítulo 2: A Abordagem Direta (e seus Problemas)**

**Objetivo:** Neste capítulo, vamos criar nossos primeiros endpoints (as URLs da nossa API). Faremos isso da maneira mais direta possível: injetando nosso AppDbContext diretamente nos Controllers. Essa abordagem é rápida e funcional para projetos pequenos, mas, como veremos, ela nos levará a um problema clássico e muito importante de se entender: a **referência circular**.

**Passo 1: Criando o Controller para Categorias**

Primeiro, precisamos de uma forma de criar categorias para depois associá-las aos produtos.

Crie um novo arquivo em Controllers/CategoriasController.cs.

**Controllers/CategoriasController.cs**

|  |
| --- |
| // Controllers/CategoriasController.cs  using GerenciadorDeProdutos.Data;  using GerenciadorDeProdutos.Models;  using Microsoft.AspNetCore.Mvc;  using Microsoft.EntityFrameworkCore;  namespace GerenciadorDeProdutos.Controllers;  [ApiController]  [Route("api/[controller]")] // Rota base: /api/categorias  public class CategoriasController : ControllerBase  {  // Declara uma variável privada para armazenar a instância do DbContext.  // O "readonly" garante que ela só pode ser atribuída no construtor.  private readonly AppDbContext \_context;  // Construtor do controller.  // A Injeção de Dependência do .NET vai "injetar" automaticamente  // uma instância do AppDbContext (que configuramos no Program.cs) aqui.  public CategoriasController(AppDbContext context)  {  \_context = context;  }  // Endpoint para criar uma nova categoria.  // POST /api/categorias  [HttpPost]  public async Task<IActionResult> CriarCategoria([FromBody] Categoria categoria)  {  // Adiciona a categoria recebida ao contexto do EF Core.  // Neste momento, a entidade está sendo "rastreada" (tracked).  await \_context.Categorias.AddAsync(categoria);  // Salva todas as mudanças rastreadas pelo contexto no banco de dados.  await \_context.SaveChangesAsync();  // Retorna um status 201 Created com a localização e o objeto criado.  return CreatedAtAction(nameof(BuscarCategoriaPorId), new { id = categoria.Id }, categoria);  }  // Endpoint para buscar uma categoria específica pelo seu ID, incluindo seus produtos.  // GET /api/categorias/{id}  [HttpGet("{id}")]  public async Task<IActionResult> BuscarCategoriaPorId(int id)  {  // Aqui está o ponto crucial que vai gerar o nosso erro. Usamos o .Include() para dizer ao EF Core: "Quando buscar a categoria, traga também a lista de produtos associada a ela".  var categoria = await \_context.Categorias  .Include(c => c.Produtos) // Traz os produtos relacionados  .FirstOrDefaultAsync(c => c.Id == id);  if (categoria == null)  {  return NotFound("Categoria não encontrada.");  }  // AQUI OCORRERÁ O ERRO!  // Estamos tentando retornar o objeto 'categoria' diretamente.  return Ok(categoria);  }  } |

**Passo 2: Criando o Controller para Produtos**

Agora, vamos criar o controller que nos permitirá cadastrar um produto dentro de uma categoria existente.

Crie um novo arquivo em Controllers/ProdutosController.cs.

**Controllers/ProdutosController.cs**

|  |
| --- |
| // Controllers/ProdutosController.cs  using GerenciadorDeProdutos.Data;  using GerenciadorDeProdutos.Models;  using Microsoft.AspNetCore.Mvc;  namespace GerenciadorDeProdutos.Controllers;  [ApiController]  [Route("api/[controller]")] // Rota base: /api/produtos  public class ProdutosController : ControllerBase  {  private readonly AppDbContext \_context;  // A mesma injeção de dependência que fizemos no CategoriasController.  public ProdutosController(AppDbContext context)  {  \_context = context;  }  // Endpoint para criar um novo produto.  // POST /api/produtos  [HttpPost]  public async Task<IActionResult> CriarProduto([FromBody] Produto produto)  {  // Verifica se a categoria informada no produto realmente existe.  var categoriaExistente = await \_context.Categorias.FindAsync(produto.CategoriaId);  if (categoriaExistente == null)  {  return BadRequest("A CategoriaId fornecida não existe.");  }  // Adiciona o produto ao contexto e salva no banco.  await \_context.Produtos.AddAsync(produto);  await \_context.SaveChangesAsync();  // Retorna um status 200 OK com o produto criado.  return Ok(produto);  }  } |

**Passo 3: Testando no Swagger e Identificando o Erro**

Agora vem a parte prática. Execute sua aplicação com o comando dotnet run e abra o navegador no endereço indicado (geralmente https://localhost:xxxx/swagger).

**Vamos seguir este fluxo:**

1. **Criar uma Categoria:**
   * Vá até o endpoint POST /api/categorias.
   * Clique em "Try it out".
   * No corpo da requisição (Request body), insira:

JSON

{

"nome": "Eletrônicos"

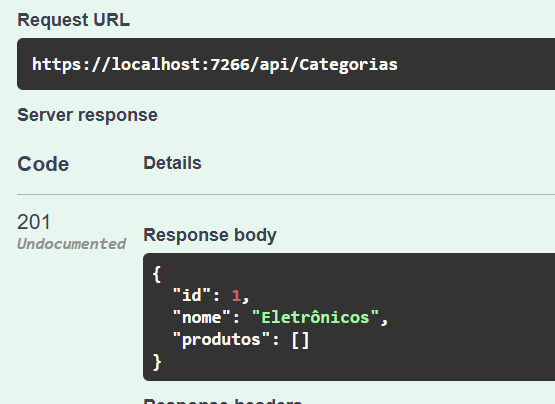
}

Vaie levantar erro 400 por causa da Nulabildiade não declarada dem categoria

Acrescentar ao código

|  |
| --- |
| public class Categoria  {  public int Id { get; set; }  public string Nome { get; set; }    // ATENÇÃO: Acrescentart a declaração da criação da lista.  public ICollection<Produto> Produtos { get; set; } = new List<Produto>();  } |

Execute novamente

O resultado será  


* + Clique em "Execute". Você deve receber uma resposta 201 Created com o corpo parecido com:

JSON

{

"id": 1,

"nome": "Eletrônicos",

"produtos": null

}

* + **Anote o id da categoria (neste caso, 1).**

1. **Criar um Produto:**
   * Vá até o endpoint POST /api/produtos.
   * Clique em "Try it out".
   * No corpo da requisição, insira os dados do produto, usando o categoriaId que você anotou:

JSON

{

"nome": "Notebook Gamer",

"preco": 7500.50,

"categoriaId": 1

}

Da mesma forma tornar Nulabel

|  |
| --- |
| using System.ComponentModel.DataAnnotations.Schema;  namespace GerenciadorDeProjetos.Models  {  public class Produto  {  public int Id { get; set; }  public string Nome { get; set; }  public decimal Preco { get; set; }  // Chave estrangeira para a Categoria  public int CategoriaId { get; set; }  // Propriedade de navegação: Um produto pertence a uma categoria.  [ForeignKey("CategoriaId")]  public Categoria? Categoria { get; set; } //declarar ?  }  } |

* + Clique em "Execute". Você deve receber uma resposta 200 OK.

1. **Provocar o Erro:**
   * Agora, vamos tentar buscar a categoria "Eletrônicos" e ver a lista de produtos dela.
   * Vá até o endpoint GET /api/categorias/{id}.
   * Clique em "Try it out".
   * No campo id, digite 1.
   * Clique em "Execute".

**O Resultado: O Erro de Referência Circular**

Em vez de uma resposta de sucesso, você verá um erro 500 Internal Server Error. Se olhar no console onde a aplicação está rodando, verá uma exceção parecida com esta:

System.Text.Json.JsonException: A possible object cycle was detected. This can either be due to a cycle or if the object depth is larger than the maximum allowed depth of 32. Consider using ReferenceHandler.Preserve on JsonSerializerOptions to support cycles.

at System.Text.Json.ThrowHelper.ThrowJsonException\_SerializerCycleDetected(Int32 maxDepth)

...

**Por que isso acontece?**

Isso é o ponto de aprendizado mais importante deste capítulo. Vamos entender o ciclo:

1. Você pediu para a API serializar (transformar em JSON) o objeto Categoria com Id = 1.
2. O serializador começa: {"id":1, "nome":"Eletrônicos", "produtos": [ ... ]}.
3. Ele entra na lista de produtos e começa a serializar o primeiro Produto ("Notebook Gamer").
4. O serializador do produto começa: {"id":1, "nome":"Notebook Gamer", "preco":7500.50, "categoriaId":1, "categoria": { ... }}.
5. Ele encontra a propriedade de navegação Categoria dentro do Produto e tenta serializá-la.
6. **PROBLEMA:** O objeto Categoria dentro do Produto é **exatamente o mesmo** objeto Categoria com que começamos no passo 1.
7. O serializador entra em um loop infinito: **Categoria -> Produtos[0] -> Categoria -> Produtos[0] -> ...**
8. Para evitar travar, o serializador detecta esse ciclo e lança a exceção que vimos.

SOLUÇÂO alterar o ADDCONTROLLERS

|  |
| --- |
| // \*\*\* SOLUÇÃO: Adicione esta configuração para resolver o erro 500 serialization\*\*\*  builder.Services.AddControllers().AddJsonOptions(options =>  {  // Ignora os ciclos de referência em vez de lançar uma exceção.  options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles;  }); |

**Resumo do Capítulo:** Criamos endpoints funcionais para cadastrar dados. No entanto, ao tentar retornar entidades que se relacionam entre si (Categoria que tem Produtos, e Produto que tem Categoria), expusemos uma falha fundamental da abordagem direta: misturar modelos de domínio (do EF Core) com modelos de API, o que causa o erro de serialização. Além disso, nosso Controller está fortemente acoplado ao DbContext, dificultando testes e manutenção.

Começaremos a resolver isso introduzindo o **Padrão Repository**, nosso primeiro passo para desacoplar o acesso a dados da nossa lógica de negócio.

**Capítulo 3: Introduzindo o Padrão Repository (IRepository)**

**Objetivo:** Nosso objetivo agora é remover a dependência direta do DbContext de dentro dos nossos Controllers. O Controller **não deve** saber ***como*** os dados são salvos ou buscados (se é com Entity Framework, Dapper, ou outro método). Ele só precisa pedir os dados.

Para isso, vamos criar uma camada de abstração chamada "Repositório".

**Por que fazer isso?**

1. **Separação de Responsabilidades:** O Controller cuida do HTTP, o Repositório cuida do acesso a dados. Código mais limpo.
2. **Testabilidade:** Fica muito mais fácil testar o Controller de forma isolada, pois podemos "simular" (fazer um Mock) um repositório em vez de precisar de um banco de dados real para os testes.
3. **Manutenção:** Se um dia quisermos trocar o SQL Server por outro banco, só precisaremos mudar a implementação do repositório, e os controllers continuarão intactos.

**Passo 1: Criando as Interfaces do Repositório**

Uma interface é como um contrato. Ela define *o que* um repositório deve ser capaz de fazer, mas não *como* ele fará.

Crie uma nova pasta Repositories e dentro dela uma subpasta Interfaces.

**Repositories/Interfaces/ICategoriaRepository.cs**

|  |
| --- |
| // Repositories/Interfaces/ICategoriaRepository.cs  using GerenciadorDeProdutos.Models;  namespace GerenciadorDeProdutos.Repositories.Interfaces;  // Contrato para o repositório de Categoria  public interface ICategoriaRepository  {  // Define um método para buscar uma categoria pelo ID, incluindo seus produtos  Task<Categoria> GetByIdWithProductsAsync(int id);  // Define um método para buscar uma categoria pelo ID (sem produtos)  Task<Categoria?> FindByIdAsync(int id);  // Define um método para adicionar uma nova categoria  Task AddAsync(Categoria categoria);  } |

**Repositories/Interfaces/IProdutoRepository.cs**

|  |
| --- |
| // Repositories/Interfaces/IProdutoRepository.cs  using GerenciadorDeProdutos.Models;  namespace GerenciadorDeProdutos.Repositories.Interfaces;  // Contrato para o repositório de Produto  public interface IProdutoRepository  {  // Define um método para adicionar um novo produto  Task AddAsync(Produto produto);  } |

**Passo 2: Criando as Implementações Concretas do Repositório**

Agora vamos criar as classes que cumprem os contratos (interfaces) que acabamos de definir. É aqui que a lógica do Entity Framework vai morar.

Na pasta Repositories, crie os seguintes arquivos:

**Repositories/CategoriaRepository.cs**

|  |
| --- |
| // Repositories/CategoriaRepository.cs  using GerenciadorDeProdutos.Data;  using GerenciadorDeProdutos.Models;  using GerenciadorDeProdutos.Repositories.Interfaces;  using Microsoft.EntityFrameworkCore;  namespace GerenciadorDeProdutos.Repositories;  // Implementação concreta do repositório de Categoria  public class CategoriaRepository : ICategoriaRepository  {  private readonly AppDbContext \_context;  public CategoriaRepository(AppDbContext context)  {  \_context = context;  }  public async Task<Categoria> GetByIdWithProductsAsync(int id)  {  // A mesma lógica que estava no controller, agora está aqui.  return await \_context.Categorias  .Include(c => c.Produtos)  .FirstOrDefaultAsync(c => c.Id == id);  }  public async Task<Categoria?> FindByIdAsync(int id)  {  return await \_context.Categorias.FindAsync(id);  }  public async Task AddAsync(Categoria categoria)  {  await \_context.Categorias.AddAsync(categoria);  // ATENÇÃO: Note que não estamos chamando o SaveChangesAsync() aqui.  // Isso será responsabilidade de outra classe (Unit of Work), para garantir  // que múltiplas operações sejam salvas em uma única transação.  }  } |

**Repositories/ProdutoRepository.cs**

|  |
| --- |
| // Repositories/ProdutoRepository.cs  using GerenciadorDeProdutos.Data;  using GerenciadorDeProdutos.Models;  using GerenciadorDeProdutos.Repositories.Interfaces;  namespace GerenciadorDeProdutos.Repositories;  // Implementação concreta do repositório de Produto  public class ProdutoRepository : IProdutoRepository  {  private readonly AppDbContext \_context;  public ProdutoRepository(AppDbContext context)  {  \_context = context;  }  public async Task AddAsync(Produto produto)  {  await \_context.Produtos.AddAsync(produto);  // Novamente, sem SaveChangesAsync() aqui.  }  } |

**Passo 3: Registrando os Repositórios na Injeção de Dependência**

Temos que dizer à nossa aplicação que, sempre que alguém pedir um ICategoriaRepository, ela deve entregar uma instância de CategoriaRepository. Fazemos isso no Program.cs.

**Program.cs**

|  |
| --- |
| using GerenciadorDeProdutos.Data;  using Microsoft.EntityFrameworkCore;  // SAI  // (Nenhum código sai, apenas adicionamos novas linhas)  // ENTRA  using GerenciadorDeProdutos.Repositories;  using GerenciadorDeProdutos.Repositories.Interfaces;  var builder = WebApplication.CreateBuilder(args);  var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");  builder.Services.AddDbContext<AppDbContext>(options =>  options.UseSqlServer(connectionString));  // --- INÍCIO DAS NOVAS LINHAS ---  // AddScoped significa que uma nova instância dos repositórios será criada  // para cada requisição HTTP, e a mesma instância será usada durante  // todo o ciclo de vida dessa requisição.  builder.Services.AddScoped<ICategoriaRepository, CategoriaRepository>();  builder.Services.AddScoped<IProdutoRepository, ProdutoRepository>();  // --- FIM DAS NOVAS LINHAS ---  // ... resto do código |

**Passo 4: Refatorando os Controllers para Usar os Repositórios**

Este é o grande momento! Vamos alterar os controllers para que eles não conheçam mais o AppDbContext. Eles só conhecerão as interfaces dos repositórios.

**Controllers/CategoriasController.cs**

|  |
| --- |
| // Controllers/CategoriasController.cs  // SAI  // using GerenciadorDeProdutos.Data;  // using Microsoft.EntityFrameworkCore;  // ENTRA  using GerenciadorDeProdutos.Repositories.Interfaces;  using GerenciadorDeProdutos.Models;  using Microsoft.AspNetCore.Mvc;  namespace GerenciadorDeProdutos.Controllers;  [ApiController]  [Route("api/[controller]")]  public class CategoriasController : ControllerBase  {  // SAI  // private readonly AppDbContext \_context;  // public CategoriasController(AppDbContext context)  // {  // \_context = context;  // }  // ENTRA  private readonly ICategoriaRepository \_categoriaRepository;  // Agora o controller depende da ABSTRAÇÃO (interface) e não da implementação.  public CategoriasController(ICategoriaRepository categoriaRepository)  {  \_categoriaRepository = categoriaRepository;  }  [HttpPost]  public async Task<IActionResult> CriarCategoria([FromBody] Categoria categoria)  {  // SAI  // await \_context.Categorias.AddAsync(categoria);  // await \_context.SaveChangesAsync();  // ENTRA  // A lógica de adicionar foi para o repositório.  await \_categoriaRepository.AddAsync(categoria);  // POR ENQUANTO, vamos ter que adicionar um SaveChanges aqui, pois ainda não  // temos o Unit of Work. Isso será refatorado no capítulo 5.  // \*\*Este é um passo intermediário para fins didáticos.\*\*  // Idealmente, o controller não deveria fazer isso.  // Para que isso funcione, precisaremos de um ajuste temporário...  // Vamos adiar a implementação do Save e introduzir o Unit of Work já!  // Fica mais limpo. Vamos pular para o UoW como parte deste capítulo.  // \*\*\* CORREÇÃO DE CURSO PARA UMA SOLUÇÃO MELHOR \*\*\*  // A explicação acima sobre o SaveChanges mostra a necessidade do Unit of Work.  // Em vez de uma solução temporária, vamos implementar uma versão simples dele agora.  // O conceito será aprofundado no capítulo 5.  // A implementação correta será mostrada após a introdução do UoW abaixo.  return CreatedAtAction(nameof(BuscarCategoriaPorId), new { id = categoria.Id }, categoria);  }  [HttpGet("{id}")]  public async Task<IActionResult> BuscarCategoriaPorId(int id)  {  // SAI  // var categoria = await \_context.Categorias  // .Include(c => c.Produtos)  // .FirstOrDefaultAsync(c => c.Id == id);  // ENTRA  var categoria = await \_categoriaRepository.GetByIdWithProductsAsync(id);  if (categoria == null)  {  return NotFound("Categoria não encontrada.");  }  return Ok(categoria);  }  } |

**Pausa para Reflexão:** Como vimos acima, surgiu um problema: quem chama o SaveChangesAsync()? Se cada repositório chamar, perdemos a capacidade de fazer várias operações em uma única transação. A resposta correta é o padrão **Unit of Work**. Para não deixar o código "quebrado", vamos introduzir uma versão simples dele agora, e aprofundar no Capítulo 5.

**Passo 5 (Ajuste): Introduzindo um Unit of Work Simples**

O Unit of Work (UoW) será uma classe que agrupa nossos repositórios e centraliza a chamada ao SaveChangesAsync, que chamaremos de CommitAsync.

**Crie as interfaces e classes a seguir:**

**Repositories/Interfaces/IUnitOfWork.cs**

C#

// Repositories/Interfaces/IUnitOfWork.cs

namespace GerenciadorDeProdutos.Repositories.Interfaces;

public interface IUnitOfWork : IDisposable

{

// Expõe os repositórios que ele gerencia

ICategoriaRepository CategoriaRepository { get; }

IProdutoRepository ProdutoRepository { get; }

// Centraliza o método para salvar as alterações no banco

Task<int> CommitAsync();

}

**Repositories/UnitOfWork.cs**

C#

// Repositories/UnitOfWork.cs

using GerenciadorDeProdutos.Data;

using GerenciadorDeProdutos.Repositories.Interfaces;

namespace GerenciadorDeProdutos.Repositories;

public class UnitOfWork : IUnitOfWork

{

private readonly AppDbContext \_context;

public ICategoriaRepository CategoriaRepository { get; private set; }

public IProdutoRepository ProdutoRepository { get; private set; }

public UnitOfWork(AppDbContext context)

{

\_context = context;

// Instancia os repositórios, passando o contexto para eles

CategoriaRepository = new CategoriaRepository(\_context);

ProdutoRepository = new ProdutoRepository(\_context);

}

public async Task<int> CommitAsync()

{

// O único lugar que chama o SaveChangesAsync

return await \_context.SaveChangesAsync();

}

public void Dispose()

{

\_context.Dispose();

}

}

**Agora, registre o UoW no Program.cs:**

C#

// Program.cs - dentro da configuração de serviços

// SAI

// builder.Services.AddScoped<ICategoriaRepository, CategoriaRepository>();

// builder.Services.AddScoped<IProdutoRepository, ProdutoRepository>();

// ENTRA

// O UnitOfWork agora é responsável por fornecer os repositórios.

// Nós o registramos e ele cuidará do resto.

builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();

**Passo 6: Refatoração Final dos Controllers com Unit of Work**

Agora sim podemos limpar os controllers de forma definitiva.

**Controllers/CategoriasController.cs (Versão Final do Capítulo)**

C#

// Controllers/CategoriasController.cs

using GerenciadorDeProdutos.Repositories.Interfaces;

using GerenciadorDeProdutos.Models;

using Microsoft.AspNetCore.Mvc;

namespace GerenciadorDeProdutos.Controllers;

[ApiController]

[Route("api/[controller]")]

public class CategoriasController : ControllerBase

{

// SAI

// private readonly ICategoriaRepository \_categoriaRepository;

// public CategoriasController(ICategoriaRepository categoriaRepository)

// {

// \_categoriaRepository = categoriaRepository;

// }

// ENTRA

private readonly IUnitOfWork \_uow;

public CategoriasController(IUnitOfWork uow)

{

\_uow = uow;

}

[HttpPost]

public async Task<IActionResult> CriarCategoria([FromBody] Categoria categoria)

{

// Usa o repositório de categoria exposto pelo Unit of Work

await \_uow.CategoriaRepository.AddAsync(categoria);

// "Commita" a transação

await \_uow.CommitAsync();

return CreatedAtAction(nameof(BuscarCategoriaPorId), new { id = categoria.Id }, categoria);

}

[HttpGet("{id}")]

public async Task<IActionResult> BuscarCategoriaPorId(int id)

{

var categoria = await \_uow.CategoriaRepository.GetByIdWithProductsAsync(id);

if (categoria == null)

{

return NotFound("Categoria não encontrada.");

}

// O erro de referência circular AINDA existe aqui!

return Ok(categoria);

}

}

**Controllers/ProdutosController.cs (Versão Final do Capítulo)**

C#

// Controllers/ProdutosController.cs

using GerenciadorDeProdutos.Repositories.Interfaces; // ENTRA

using GerenciadorDeProdutos.Models;

using Microsoft.AspNetCore.Mvc;

namespace GerenciadorDeProdutos.Controllers;

[ApiController]

[Route("api/[controller]")]

public class ProdutosController : ControllerBase

{

// SAI

// private readonly AppDbContext \_context;

// public ProdutosController(AppDbContext context)

// {

// \_context = context;

// }

// ENTRA

private readonly IUnitOfWork \_uow;

public ProdutosController(IUnitOfWork uow)

{

\_uow = uow;

}

[HttpPost]

public async Task<IActionResult> CriarProduto([FromBody] Produto produto)

{

// SAI

// var categoriaExistente = await \_context.Categorias.FindAsync(produto.CategoriaId);

// ENTRA

var categoriaExistente = await \_uow.CategoriaRepository.FindByIdAsync(produto.CategoriaId);

if (categoriaExistente == null)

{

return BadRequest("A CategoriaId fornecida não existe.");

}

// SAI

// await \_context.Produtos.AddAsync(produto);

// await \_context.SaveChangesAsync();

// ENTRA

await \_uow.ProdutoRepository.AddAsync(produto);

await \_uow.CommitAsync();

return Ok(produto);

}

}

**Resumo do Capítulo:** Ufa! Foi um capítulo denso, mas veja o que conquistamos:

* A lógica de acesso a dados foi movida do Controller para classes de Repositório.
* Os Controllers agora dependem de interfaces (IUnitOfWork), não de implementações concretas (AppDbContext), o que é um pilar da boa arquitetura de software.
* Centralizamos o ato de salvar no banco (CommitAsync), garantindo a integridade das nossas operações.

**Capítulo 4**, onde os DTOs entram em cena para resolver nosso problema?

Agora que entendemos e corrigimos este ponto, estamos prontos para ir para o **Capítulo 4** e finalmente resolver o erro de referência circular usando DTOs. Vamos lá?

// Models/Categoria.cs

namespace GerenciadorDeProdutos.Models;

public class Categoria

{

    public int Id { get; set; }

    public string Nome { get; set; }

    // Propriedade de navegação: Uma categoria pode ter vários produtos.

    // É ESSENCIAL para o EF Core entender a relação.

    // ATENÇÃO: Essa linha será a causa do nosso futuro erro de serialização.

    public ICollection<Produto> Produtos { get; set; }

}

Mostrar raciocínio

Correto. Você postou novamente o modelo Categoria.cs, e a linha public ICollection<Produto> Produtos { get; set; } é exatamente a causa do nosso problema de referência circular.

Agora, vamos resolver isso de uma vez por todas, da maneira correta, usando DTOs.

**Capítulo 4: Resolvendo o Problema de Serialização com DTOs (Data Transfer Objects)**

**Objetivo:** Vamos criar classes especiais, chamadas DTOs, que servirão como "moldes" para os dados que entram e saem da nossa API. Eles nos permitem controlar exatamente quais informações são expostas, resolvendo o problema da referência circular e tornando nossa API mais segura e robusta.

**Por que usar DTOs?**

1. **Resolver Referências Circulares:** Nosso DTO de Categoria terá uma lista de DTOs de Produto, e o DTO de Produto **não** terá uma referência de volta para a Categoria, quebrando o ciclo.
2. **Segurança (Evitar Over-Posting):** Ao criar um produto, o cliente só precisa enviar Nome, Preco e CategoriaId. Usar um DTO de criação garante que ele não possa enviar outros dados maliciosos (como tentar setar o Id do produto, por exemplo).
3. **Contrato de API Estável:** Nossos modelos de banco de dados (Produto, Categoria) podem mudar internamente, mas desde que consigamos mapeá-los para os DTOs, o "contrato" com os clientes da API permanece o mesmo, sem quebrar as aplicações deles.

**Passo 1: Criando os DTOs**

Crie uma nova pasta na raiz do projeto chamada DTOs. Dentro dela, adicione as seguintes classes:

**DTOs/CriarCategoriaDto.cs**

C#

// DTOs/CriarCategoriaDto.cs

using System.ComponentModel.DataAnnotations;

namespace GerenciadorDeProdutos.DTOs;

// DTO usado para receber os dados para a criação de uma nova categoria.

public class CriarCategoriaDto

{

[Required(ErrorMessage = "O nome da categoria é obrigatório.")]

[StringLength(100, ErrorMessage = "O nome não pode ter mais de 100 caracteres.")]

public string Nome { get; set; }

}

**DTOs/CriarProdutoDto.cs**

|  |
| --- |
| // DTO para receber os dados de criação de um novo produto.  public class CriarProdutoDto  {  [Required(ErrorMessage = "O nome do produto é obrigatório.")]  [StringLength(120)]  public string Nome { get; set; }  [Required]  [Range(0.01, 99999.99, ErrorMessage = "O preço deve ser maior que zero.")]  public decimal Preco { get; set; }  [Required]  public int CategoriaId { get; set; }  } |

**DTOs/ProdutoDto.cs**

C#

// DTOs/ProdutoDto.cs

namespace GerenciadorDeProdutos.DTOs;

// DTO para EXIBIR os dados de um produto.

// Note que ele não tem a propriedade "Categoria", apenas propriedades simples.

public class ProdutoDto

{

public int Id { get; set; }

public string Nome { get; set; }

public decimal Preco { get; set; }

}

**DTOs/CategoriaComProdutosDto.cs**

C#

// DTOs/CategoriaComProdutosDto.cs

namespace GerenciadorDeProdutos.DTOs;

// DTO para exibir uma categoria com sua lista de produtos.

// ESTA É A CLASSE QUE RESOLVE O PROBLEMA!

public class CategoriaComProdutosDto

{

public int Id { get; set; }

public string Nome { get; set; }

// A lista não é de "Produto", e sim de "ProdutoDto".

// Como "ProdutoDto" não tem uma referência de volta para Categoria, o ciclo é quebrado.

public ICollection<ProdutoDto> Produtos { get; set; }

}

**Passo 2: Refatorando os Controllers para Usar DTOs**

Agora, vamos alterar os controllers para que eles recebam e retornem apenas DTOs. A conversão entre DTO e Entidade (Modelo do Banco) acontecerá dentro dos métodos do controller.

Nota: Em projetos maiores, essa lógica de mapeamento geralmente fica em uma camada de "Serviços", mas para fins didáticos, faremos no controller.

**Controllers/CategoriasController.cs**

// Controllers/CategoriasController.cs

using GerenciadorDeProdutos.Repositories.Interfaces;

using GerenciadorDeProdutos.Models;

using Microsoft.AspNetCore.Mvc;

// ENTRA

using GerenciadorDeProdutos.DTOs;

namespace GerenciadorDeProdutos.Controllers;

[ApiController]

[Route("api/[controller]")]

public class CategoriasController : ControllerBase

{

private readonly IUnitOfWork \_uow;

public CategoriasController(IUnitOfWork uow)

{

\_uow = uow;

}

|  |
| --- |
| // POST /api/categorias  [HttpPost]  // SAI  // public async Task<IActionResult> CriarCategoria([FromBody] Categoria categoria)  // ENTRA  public async Task<IActionResult> CriarCategoria([FromBody] CriarCategoriaDto categoriaDto)  {  // Mapeamento do DTO para a Entidade  var categoria = new Categoria  {  Nome = categoriaDto.Nome  };  await \_uow.CategoriaRepository.AddAsync(categoria);  await \_uow.CommitAsync();  // Mapeamento da Entidade para um DTO de retorno (opcional, mas boa prática)  var categoriaResultDto = new CategoriaComProdutosDto  {  Id = categoria.Id,  Nome = categoria.Nome  };  return CreatedAtAction(nameof(BuscarCategoriaPorId), new { id = categoria.Id }, categoriaResultDto);  } |

## 📁 Estrutura Geral

Este código está dentro de um controller da API, e segue uma arquitetura com:

* **Camada Controller** → Responsável por receber e responder requisições HTTP.
* **DTOs (Data Transfer Objects)** → Objetos usados para transportar dados entre as camadas (evita exposição direta das entidades).
* **Unit of Work** → Coordena múltiplos repositórios e gerencia o SaveChanges() de forma unificada.

## 🔍 Análise Linha a Linha

### Declaração do Namespace e Controller

csharp

CopiarEditar

namespace GerenciadorDeProdutos.Controllers;

* Define o **namespace**, que agrupa classes relacionadas. Aqui indica que esta classe pertence à camada de controllers do sistema GerenciadorDeProdutos.

csharp

CopiarEditar

[ApiController]

[Route("api/[controller]")]

public class CategoriasController : ControllerBase

* [ApiController]: Habilita comportamentos automáticos úteis, como validação de ModelState.
* [Route("api/[controller]")]: Define a rota como api/categorias, usando o nome da classe (CategoriasController).
* ControllerBase: Classe base para APIs (sem suporte a Views como Controller do MVC).

### Injeção de Dependência

csharp

CopiarEditar

private readonly IUnitOfWork \_uow;

public CategoriasController(IUnitOfWork uow)

{

\_uow = uow;

}

* Injeção de **IUnitOfWork**, que fornece acesso aos repositórios e gerencia transações.
* Promove **injeção de dependência** e **princípio de responsabilidade única**.

## 🎯 Ação: CriarCategoria

csharp

CopiarEditar

[HttpPost]

public async Task<IActionResult> CriarCategoria([FromBody] CriarCategoriaDto categoriaDto)

* Define uma ação que será chamada com um POST /api/categorias.
* O corpo da requisição será um JSON que será convertido em um objeto CriarCategoriaDto.

### 🚀 O que é um DTO?

* **DTO (Data Transfer Object)** é uma **classe de transporte de dados**, usada para:
  + **Evitar exposição de entidades diretamente** (proteção e encapsulamento).
  + **Evitar vazamento de dados sensíveis ou desnecessários.**
  + **Evitar problemas com referências circulares** durante a serialização.
  + **Permitir validações específicas** com DataAnnotations no DTO.

csharp

CopiarEditar

// Exemplo típico de DTO de entrada:

public class CriarCategoriaDto

{

public string Nome { get; set; } = string.Empty;

}

### Conversão do DTO para Entidade

csharp

CopiarEditar

var categoria = new Categoria

{

Nome = categoriaDto.Nome

};

* Aqui ocorre o **mapeamento manual** de um DTO (entrada) para a entidade de domínio (Categoria).
* Evita que o client envie mais campos do que o permitido.

### Salvar no banco de dados

csharp

CopiarEditar

await \_uow.CategoriaRepository.AddAsync(categoria);

await \_uow.CommitAsync();

* Adiciona a entidade ao banco por meio do repositório.
* CommitAsync() salva as mudanças, coordenado pelo **UnitOfWork**.

### Mapeamento de retorno: DTO de saída

csharp

CopiarEditar

var categoriaResultDto = new CategoriaComProdutosDto

{

Id = categoria.Id,

Nome = categoria.Nome

};

* Cria um DTO específico para **resposta (output)**.
* **Evita retornar a entidade**, que pode conter dados indesejados (ex: navegação EF, timestamps, tracking).

csharp

CopiarEditar

return CreatedAtAction(nameof(BuscarCategoriaPorId), new { id = categoria.Id }, categoriaResultDto);

* Retorna um HTTP 201 com cabeçalho Location apontando para GET /api/categorias/{id}.
* Boa prática RESTful.

|  |
| --- |
| // Endpoint para buscar uma categoria específica pelo seu ID, incluindo seus produtos.  // GET /api/categorias/{id}  [HttpGet("{id}")]  public async Task<IActionResult> BuscarCategoriaPorId(int id)  {  var categoria = await \_uow.CategoriaRepository.GetByIdWithProductsAsync(id);  if (categoria == null)  {  return NotFound("Categoria não encontrada.");  }  // SAI  // return Ok(categoria);  // ENTRA  // Mapeamento manual da entidade Categoria para o nosso DTO.  var categoriaDto = new CategoriaComProdutosDto  {  Id = categoria.Id,  Nome = categoria.Nome,  // Usamos Linq para transformar cada Produto da lista em um ProdutoDto.  Produtos = categoria.Produtos.Select(p => new ProdutoDto  {  Id = p.Id,  Nome = p.Nome,  Preco = p.Preco  }).ToList()  };  // Retornamos o DTO, que é um objeto limpo e sem referências circulares.  return Ok(categoriaDto);  } |

**🔍 Ação: BuscarCategoriaPorId**

csharp

CopiarEditar

[HttpGet("{id}")]

public async Task<IActionResult> BuscarCategoriaPorId(int id)

* Endpoint GET /api/categorias/{id}.
* Recebe id como parâmetro de rota.

**Busca com relacionamento**

csharp

CopiarEditar

var categoria = await \_uow.CategoriaRepository.GetByIdWithProductsAsync(id);

* Busca a entidade Categoria com seus produtos associados.
* Método do repositório especializado CategoriaRepository.

**Retorno com DTO composto**

csharp

CopiarEditar

var categoriaDto = new CategoriaComProdutosDto

{

Id = categoria.Id,

Nome = categoria.Nome,

Produtos = categoria.Produtos.Select(p => new ProdutoDto

{

Id = p.Id,

Nome = p.Nome,

Preco = p.Preco

}).ToList()

};

* Mapeia manualmente a categoria e seus produtos para um DTO de saída.
* Evita problemas de **serialização circular** com EF e permite retornar apenas os dados relevantes.

**🧾 DTOs envolvidos**

**📥 CriarCategoriaDto (entrada)**

csharp

CopiarEditar

public class CriarCategoriaDto

{

public string Nome { get; set; } = string.Empty;

}

* Usado para **criação**. Não expõe ID nem produtos.

**📤 CategoriaComProdutosDto (resposta)**

csharp

CopiarEditar

public class CategoriaComProdutosDto

{

public int Id { get; set; }

public string Nome { get; set; } = string.Empty;

public List<ProdutoDto> Produtos { get; set; } = new();

}

* Inclui produtos relacionados.
* Ideal para **detalhamento** de uma categoria.

**📤 ProdutoDto (resposta)**

csharp

CopiarEditar

public class ProdutoDto

{

public int Id { get; set; }

public string Nome { get; set; } = string.Empty;

public decimal Preco { get; set; }

}

* Retorna somente os dados essenciais do produto.
* Exclui CategoriaId, timestamps ou qualquer propriedade de navegação.

**✅ Vantagens do uso de DTOs**

| **Vantagem** | **Explicação** |
| --- | --- |
| 🧼 Separação de camadas | A entidade não se mistura com os dados trafegados pela API. |
| 🔒 Segurança | Evita vazamento de dados sensíveis (como senhas, tokens, timestamps). |
| 📉 Performance | Pode reduzir o tamanho do payload JSON. |
| 🧪 Testabilidade | Facilita mocks e testes com dados previsíveis. |
| 🔄 Manutenção | DTOs podem evoluir sem quebrar regras de negócio da entidade. |

**📌 Resumo**

Este controller está muito bem estruturado. O uso de DTOs aqui segue essas boas práticas:

* **Entrada (request)** com CriarCategoriaDto para proteger a entidade.
* **Saída (response)** com CategoriaComProdutosDto para entregar apenas os dados úteis.
* **Mapeamento manual**, mas claro e didático.
* Uso de UnitOfWork e repositórios especializados.

**Controllers/ProdutosController.cs**

C#

// Controllers/ProdutosController.cs

using GerenciadorDeProdutos.Repositories.Interfaces;

using GerenciadorDeProdutos.Models;

using Microsoft.AspNetCore.Mvc;

// ENTRA

using GerenciadorDeProdutos.DTOs;

namespace GerenciadorDeProdutos.Controllers;

[ApiController]

[Route("api/[controller]")]

public class ProdutosController : ControllerBase

{

private readonly IUnitOfWork \_uow;

public ProdutosController(IUnitOfWork uow)

{

\_uow = uow;

}

[HttpPost]

// SAI

// public async Task<IActionResult> CriarProduto([FromBody] Produto produto)

// ENTRA

public async Task<IActionResult> CriarProduto([FromBody] CriarProdutoDto produtoDto)

{

// SAI

// var categoriaExistente = await \_uow.CategoriaRepository.FindByIdAsync(produto.CategoriaId);

// ENTRA

var categoriaExistente = await \_uow.CategoriaRepository.FindByIdAsync(produtoDto.CategoriaId);

if (categoriaExistente == null)

{

return BadRequest("A CategoriaId fornecida não existe.");

}

// Mapeamento do DTO para a Entidade

var produto = new Produto

{

Nome = produtoDto.Nome,

Preco = produtoDto.Preco,

CategoriaId = produtoDto.CategoriaId

};

await \_uow.ProdutoRepository.AddAsync(produto);

await \_uow.CommitAsync();

// Mapeamento da entidade criada para um DTO de resposta

var produtoResultDto = new ProdutoDto

{

Id = produto.Id,

Nome = produto.Nome,

Preco = produto.Preco

};

return Ok(produtoResultDto);

}

}

**Passo 3: Testando a Solução no Swagger**

Execute a aplicação (dotnet run) e volte para o Swagger.

1. **Criar Categoria:** Use o POST /api/categorias. O corpo da requisição agora é um CriarCategoriaDto.

JSON

{

"nome": "Material de Escritório"

}

Anote o id retornado.

1. **Criar Produto:** Use o POST /api/produtos. O corpo agora é um CriarProdutoDto.

JSON

{

"nome": "Caneta Azul",

"preco": 1.50,

"categoriaId": 2 // Use o ID que você anotou

}

1. **Verificar a Solução:** Agora, o teste final. Vá para GET /api/categorias/{id} e use o id da categoria que você criou.

**O Resultado Esperado: Sucesso!**

Desta vez, em vez de um erro 500, você receberá uma resposta 200 OK com um JSON limpo e bem estruturado, parecido com este:

JSON

{

"id": 2,

"nome": "Material de Escritório",

"produtos": [

{

"id": 2,

"nome": "Caneta Azul",

"preco": 1.5

}

]

}

**Capítulo 5: Garantindo a Integridade com Unit of Work (UoW)**

**Objetivo:** Neste capítulo final, vamos solidificar o conceito de Unit of Work que já introduzimos. O objetivo é entender profundamente por que ele é crucial para garantir a **atomicidade** das operações, ou seja, garantir que uma sequência de ações no banco de dados aconteça por completo ou não aconteça de forma alguma, mantendo os dados sempre consistentes.

**O Problema que o UoW Resolve**

Imagine uma operação mais complexa. Por exemplo, ao cadastrar um novo produto, você também precisa atualizar um campo na categoria, como QuantidadeDeProdutos. A operação completa seria:

1. Adicionar o novo Produto à tabela de produtos.
2. Buscar a Categoria correspondente.
3. Incrementar o campo QuantidadeDeProdutos na categoria.
4. Salvar as alterações.

O que acontece se o passo 1 (adicionar produto) funcionar, mas o passo 3 (atualizar categoria) falhar por algum motivo? Se cada repositório salvasse suas próprias alterações (SaveChanges()), teríamos um produto novo no banco, mas a contagem na categoria estaria errada. Os dados ficariam **inconsistentes**.

O padrão Unit of Work resolve isso. Ele age como um "invólucro" para uma transação de negócio. Todas as alterações (adições, atualizações, exclusões) são rastreadas dentro de um único DbContext. A chamada ao método CommitAsync() (ou SaveChangesAsync()) é feita **uma única vez** no final, o que garante que todas as alterações sejam salvas juntas em uma única transação no banco de dados. Se qualquer parte falhar antes do CommitAsync(), nada é salvo.

**Passo 1: Revisando e Refinando nossa Implementação**

Nós já criamos uma implementação funcional no Capítulo 3 como uma necessidade para fazer o padrão Repository funcionar corretamente. Agora, vamos revisá-la e garantir que ela está robusta.

Nossas interfaces e classes atuais já estão bem estruturadas:

**Repositories/Interfaces/IUnitOfWork.cs** (Revisão)

C#

// Nenhuma alteração necessária aqui. A estrutura está correta.

public interface IUnitOfWork : IDisposable

{

ICategoriaRepository CategoriaRepository { get; }

IProdutoRepository ProdutoRepository { get; }

Task<int> CommitAsync();

}

**Repositories/UnitOfWork.cs** (Revisão)

C#

// Nenhuma alteração necessária aqui. A estrutura está correta.

public class UnitOfWork : IUnitOfWork

{

private readonly AppDbContext \_context;

// O uso de 'lazy loading' pode ser um refinamento, mas para nosso escopo,

// a instanciação direta no construtor é clara e eficaz.

public ICategoriaRepository CategoriaRepository { get; private set; }

public IProdutoRepository ProdutoRepository { get; private set; }

public UnitOfWork(AppDbContext context)

{

\_context = context;

CategoriaRepository = new CategoriaRepository(\_context);

ProdutoRepository = new ProdutoRepository(\_context);

}

public async Task<int> CommitAsync()

{

return await \_context.SaveChangesAsync();

}

public void Dispose()

{

\_context.Dispose();

}

}

A implementação que fizemos já segue os princípios corretamente. O mais importante agora é ver como ela brilha na prática.

**Passo 2: Usando o UoW em um Cenário Prático**

Vamos simular o cenário que descrevemos. Primeiro, vamos adicionar o campo QuantidadeDeProdutos na nossa entidade Categoria.

**Models/Categoria.cs (Alteração)**

C#

// Models/Categoria.cs

namespace GerenciadorDeProdutos.Models;

public class Categoria

{

public int Id { get; set; }

public string Nome { get; set; }

// ENTRA

// Novo campo para demonstrar a transação

public int QuantidadeDeProdutos { get; set; }

public ICollection<Produto> Produtos { get; set; }

}

Como alteramos um modelo do banco, precisamos criar e aplicar uma nova **Migration**:

1. No terminal: dotnet ef migrations add "AdicionaQuantidadeDeProdutosNaCategoria"
2. Depois: dotnet ef database update

Agora, vamos alterar o método CriarProduto para que ele também atualize essa contagem.

**Controllers/ProdutosController.cs (Alteração)**

C#

// ... usings

using GerenciadorDeProdutos.DTOs;

[ApiController]

[Route("api/[controller]")]

public class ProdutosController : ControllerBase

{

private readonly IUnitOfWork \_uow;

public ProdutosController(IUnitOfWork uow)

{

\_uow = uow;

}

[HttpPost]

public async Task<IActionResult> CriarProduto([FromBody] CriarProdutoDto produtoDto)

{

// 1. Busca a categoria usando o repositório de categoria

var categoria = await \_uow.CategoriaRepository.FindByIdAsync(produtoDto.CategoriaId);

if (categoria == null)

{

return BadRequest("A CategoriaId fornecida não existe.");

}

// Mapeamento do DTO para a Entidade

var produto = new Produto

{

Nome = produtoDto.Nome,

Preco = produtoDto.Preco,

CategoriaId = produtoDto.CategoriaId

};

// 2. Adiciona o novo produto usando o repositório de produto

await \_uow.ProdutoRepository.AddAsync(produto);

// 3. Modifica a entidade Categoria

categoria.QuantidadeDeProdutos++;

// Não precisamos de um método "Update" no repositório para isso,

// pois o EF Core já está rastreando a entidade 'categoria' que buscamos.

// Qualquer alteração nela será detectada.

// 4. Salva TODAS as alterações de uma vez só!

// Tanto a adição do novo produto quanto a atualização da categoria

// serão enviadas ao banco de dados em uma única transação.

// Se qualquer uma das operações falhar aqui, nada será salvo.

await \_uow.CommitAsync();

var produtoResultDto = new ProdutoDto

{

Id = produto.Id,

Nome = produto.Nome,

Preco = produto.Preco

};

return Ok(produtoResultDto);

}

}

Como Exibir esta informação no Controller para ser vista no Swagger?

A informação QuantidadeDeProdutos **é uma propriedade da Categoria**. Portanto, o controller responsável por exibir essa informação deve ser o **CategoriasController**.

No entanto, a **atualização** desse número acontece no **ProdutosController**, no momento em que um novo produto é criado.

Vamos detalhar:

1. **Onde a informação é ATUALIZADA?**
   * No **ProdutosController**, dentro do método CriarProduto.
   * **Por quê?** Porque a "ação de negócio" que dispara a necessidade de atualizar a contagem é a criação de um produto. A cada produto novo, nós incrementamos a contagem na sua respectiva categoria.
   * **Trecho do Código (ProdutosController.cs):**

C#

// ...

// 3. Modifica a entidade Categoria

categoria.QuantidadeDeProdutos++; // AQUI a contagem é atualizada

// 4. Salva TODAS as alterações de uma vez só!

await \_uow.CommitAsync();

// ...

1. **Onde a informação deve ser EXIBIDA?**
   * No **CategoriasController**.
   * **Por quê?** Porque QuantidadeDeProdutos é um dado descritivo de uma Categoria. Quando alguém pede os detalhes de uma categoria, faz todo o sentido que essa informação venha junto.

**Como Fazer a Informação Aparecer?**

Você deve ter notado que, se chamar o endpoint GET /api/categorias/{id} agora, o campo quantidadeDeProdutos não aparece. Isso acontece porque nós ainda não o adicionamos ao nosso DTO de resposta.

Vamos corrigir isso em 2 passos simples:

**Passo 1: Adicionar o Campo ao DTO**

Abra o DTO que usamos para exibir os detalhes da categoria e adicione a nova propriedade.

**Abra o arquivo DTOs/CategoriaComProdutosDto.cs:**

C#

// DTOs/CategoriaComProdutosDto.cs

namespace GerenciadorDeProdutos.DTOs;

public class CategoriaComProdutosDto

{

public int Id { get; set; }

public string Nome { get; set; }

// ENTRA

// Adicionamos o campo que queremos expor na nossa API.

public int QuantidadeDeProdutos { get; set; }

public ICollection<ProdutoDto> Produtos { get; set; }

}

**Passo 2: Popular o Campo no Controller**

Agora, precisamos "preencher" essa nova propriedade do DTO com o valor que vem da nossa entidade do banco de dados.

**Abra o Controllers/CategoriasController.cs e ajuste o mapeamento no método BuscarCategoriaPorId:**

C#

// Controllers/CategoriasController.cs

// ...

[HttpGet("{id}")]

public async Task<IActionResult> BuscarCategoriaPorId(int id)

{

var categoria = await \_uow.CategoriaRepository.GetByIdWithProductsAsync(id);

if (categoria == null)

{

return NotFound("Categoria não encontrada.");

}

// Mapeamento manual da entidade Categoria para o nosso DTO.

var categoriaDto = new CategoriaComProdutosDto

{

Id = categoria.Id,

Nome = categoria.Nome,

// ENTRA

// Aqui nós passamos o valor da entidade para o DTO.

QuantidadeDeProdutos = categoria.QuantidadeDeProdutos,

Produtos = categoria.Produtos.Select(p => new ProdutoDto

{

Id = p.Id,

Nome = p.Nome,

Preco = p.Preco

}).ToList()

};

return Ok(categoriaDto);

}

// ...

**Teste Final**

Execute a aplicação novamente e chame o endpoint **GET /api/categorias/{id}** no Swagger para uma categoria que já tenha produtos. O resultado agora será:

Só que o resultado não ocorreu como eseperado

Por que?

Você identificou perfeitamente uma das armadilhas de se manter um campo de contagem manualmente: ele pode facilmente ficar dessincronizado com a realidade dos dados.

A resposta é **sim**, existe uma maneira muito melhor e mais confiável de fazer isso, que garante que o número retornado seja **sempre** o valor real de elementos na coleção de produtos.

A melhor abordagem é **calcular a contagem dinamicamente** no momento da consulta, em vez de depender de um incremento. A coleção Produtos é a "fonte da verdade", então vamos usá-la diretamente.

Com esta alteração, o quantidadeDeProdutos++ se torna **desnecessário e deve ser removido**.

Vamos ajustar o código em 2 passos.

**Passo 1: Remover a Lógica de Incremento (Torná-la mais simples)**

Primeiro, vamos ao ProdutosController e remover a linha que atualiza a contagem manualmente. A responsabilidade de criar um produto não deve ser misturada com a de atualizar a categoria.

**Abra o arquivo Controllers/ProdutosController.cs:**

C#

// ... (usings)

[HttpPost]

public async Task<IActionResult> CriarProduto([FromBody] CriarProdutoDto produtoDto)

{

var categoria = await \_uow.CategoriaRepository.FindByIdAsync(produtoDto.CategoriaId);

if (categoria == null)

{

return BadRequest("A CategoriaId fornecida não existe.");

}

var produto = new Produto

{

Nome = produtoDto.Nome,

Preco = produtoDto.Preco,

CategoriaId = produtoDto.CategoriaId

};

await \_uow.ProdutoRepository.AddAsync(produto);

// SAI

// A lógica de negócio de atualizar a categoria não pertence mais aqui.

// O código fica mais limpo e com uma única responsabilidade.

// categoria.QuantidadeDeProdutos++;

// Apenas salvamos o novo produto.

await \_uow.CommitAsync();

var produtoResultDto = new ProdutoDto

{

Id = produto.Id,

Nome = produto.Nome,

Preco = produto.Preco

};

return Ok(produtoResultDto);

}

**Passo 2: Calcular a Contagem Dinamicamente no CategoriasController**

Agora, no CategoriasController, quando formos montar nosso DTO de resposta, em vez de ler o valor do campo QuantidadeDeProdutos (que pode estar errado), vamos simplesmente contar quantos itens existem na coleção Produtos que o Entity Framework já carregou para nós.

**Abra o arquivo Controllers/CategoriasController.cs:**

C#

// ... (usings)

[HttpGet("{id}")]

public async Task<IActionResult> BuscarCategoriaPorId(int id)

{

// O método GetByIdWithProductsAsync já faz o .Include(c => c.Produtos),

// então a lista de produtos já está disponível na memória.

var categoria = await \_uow.CategoriaRepository.GetByIdWithProductsAsync(id);

if (categoria == null)

{

return NotFound("Categoria não encontrada.");

}

var categoriaDto = new CategoriaComProdutosDto

{

Id = categoria.Id,

Nome = categoria.Nome,

// SAI

// QuantidadeDeProdutos = categoria.QuantidadeDeProdutos,

// ENTRA

// Aqui está a mágica!

// Usamos a propriedade .Count da coleção de Produtos para obter o número real.

// A contagem é sempre precisa e baseada nos dados atuais.

QuantidadeDeProdutos = categoria.Produtos.Count,

Produtos = categoria.Produtos.Select(p => new ProdutoDto

{

Id = p.Id,

Nome = p.Nome,

Preco = p.Preco

}).ToList()

};

return Ok(categoriaDto);

}

**Passo Opcional (Recomendado): Limpando o Banco de Dados**

Já que não estamos mais usando o campo QuantidadeDeProdutos na tabela Categorias, é uma boa prática removê-lo para evitar confusão.

1. **Remova a propriedade** do modelo Models/Categoria.cs:

C#

// Models/Categoria.cs

public class Categoria

{

public int Id { get; set; }

public string Nome { get; set; }

// SAI (Remova esta linha)

// public int QuantidadeDeProdutos { get; set; }

public ICollection<Produto> Produtos { get; set; }

}

1. **Crie uma nova Migration** para aplicar a remoção da coluna no banco:

Bash

dotnet ef migrations add "RemoveQuantidadeDeProdutosFromCategoria"

dotnet ef database update

**Vantagens desta Nova Abordagem**

1. **Sempre Correto:** A contagem nunca mais ficará dessincronizada. Ela reflete o estado real do banco de dados.
2. **Código Mais Simples:** A lógica de criação de produto ficou mais simples e focada em sua única responsabilidade. Você não precisa se preocupar em decrementar a contagem se um produto for excluído, por exemplo.
3. **Menos Propenso a Erros:** Remove a possibilidade de esquecer de atualizar o contador em alguma outra operação futura (como mover um produto de categoria).

Agora, ao testar, o valor de quantidadeDeProdutos será sempre o número exato de produtos associados àquela categoria.

Testes

**Configurando o Projeto de Teste no Visual Studio**

**Passo 1: Onde e Como Criar o Projeto de Teste**

O projeto de teste será criado dentro da sua **Solução** atual, ficando ao lado do seu projeto principal GerenciadorDeProdutos.

1. Abra o Visual Studio com sua solução. Na janela do **Gerenciador de Soluções** (Solution Explorer), que geralmente fica no lado direito, você verá seu projeto GerenciadorDeProdutos.
2. Clique com o **botão direito do mouse** sobre a linha da **Solução** (a primeira linha, que tem o nome da sua solução, por exemplo, "Solution 'GerenciadorDeProdutos'").
3. No menu que aparecer, vá em **Adicionar (Add)** -> **Novo Projeto... (New Project...)**.

!([https://i.imgur.com/8i275k9.png](https://www.google.com/search?q=https://i.imgur.com/8i275k9.png&authuser=1" \t "_blank))

1. Uma nova janela chamada "Adicionar um novo projeto" vai abrir. Na barra de busca no topo, digite **xUnit**.
2. Selecione a opção **"Projeto de Teste do xUnit" (xUnit Test Project)**. Certifique-se de que é a versão para C#. Clique em **Próximo (Next)**.

!([https://i.imgur.com/3vB8Sws.png](https://www.google.com/search?q=https://i.imgur.com/3vB8Sws.png&authuser=1" \t "_blank))

1. Na tela seguinte, "Configure seu novo projeto", dê o nome ao projeto. O padrão é usar o nome do projeto principal seguido de .Tests. Então, nomeie-o como: **GerenciadorDeProdutos.Tests**. Mantenha o local padrão. Clique em **Próximo (Next)**.

!([https://i.imgur.com/e5k05Vv.png](https://www.google.com/search?q=https://i.imgur.com/e5k05Vv.png&authuser=1" \t "_blank))

1. Na última tela, "Informações adicionais", apenas confirme se a versão do .NET (Framework) é a mesma do seu projeto principal (por exemplo, .NET 8.0) e clique em **Criar (Create)**.

Pronto! Agora você verá no seu Gerenciador de Soluções a seguinte estrutura:

Solution 'GerenciadorDeProdutos'

|-- GerenciadorDeProdutos (Seu projeto principal da API)

|-- GerenciadorDeProdutos.Tests (Seu novo projeto de testes)

**Passo 2: Adicionando a Referência ao Projeto Principal**

Seu projeto de teste precisa "enxergar" as classes do seu projeto da API (Controllers, DTOs, etc.). Para isso, adicionamos uma referência.

1. No Gerenciador de Soluções, dentro do projeto GerenciadorDeProdutos.Tests, clique com o **botão direito do mouse** na pasta **Dependências (Dependencies)**.
2. No menu, clique em **Adicionar Referência do Projeto... (Add Project Reference...)**.

!([https://i.imgur.com/GZ2nFj2.png](https://www.google.com/search?q=https://i.imgur.com/GZ2nFj2.png&authuser=1" \t "_blank))

1. Uma janela se abrirá. Marque a caixa de seleção ao lado do seu projeto principal, **GerenciadorDeProdutos**.
2. Clique em **OK**.

**Passo 3: Como Instalar Pacotes pelo Gerenciador do NuGet**

Agora, vamos instalar o Moq e o FluentAssertions usando a interface gráfica do NuGet.

1. No Gerenciador de Soluções, clique com o **botão direito do mouse** sobre o **projeto GerenciadorDeProdutos.Tests**.
2. No menu, selecione a opção **Gerenciar Pacotes do NuGet... (Manage NuGet Packages...)**.

!([https://i.imgur.com/1B7E20E.png](https://www.google.com/search?q=https://i.imgur.com/1B7E20E.png&authuser=1" \t "_blank))

1. Uma nova aba se abrirá no Visual Studio. Clique na guia **Procurar (Browse)** no topo.

!([https://i.imgur.com/6XwYj2o.png](https://www.google.com/search?q=https://i.imgur.com/6XwYj2o.png&authuser=1" \t "_blank))

1. **Para instalar o Moq:**
   * Na barra de busca, digite **Moq**.
   * Clique em "Moq" na lista de resultados à esquerda.
   * No painel da direita, clique no botão **Instalar (Install)**.
   * Uma janela de confirmação pode aparecer. Clique em "OK" e "Aceito" para confirmar.
2. **Para instalar o FluentAssertions:**
   * Ainda na mesma tela, apague "Moq" da busca e digite **FluentAssertions**.
   * Clique em "FluentAssertions" na lista.
   * Clique no botão **Instalar (Install)** à direita e aceite as confirmações.

**Criar a Classe de Teste**

Dentro do projeto GerenciadorDeProdutos.Tests, renomeie o arquivo UnitTest1.cs para ProdutosControllerTests.cs. Vamos escrever nosso teste nesta classe.

**Passo 3: Escrevendo o Teste para ListarTodosProdutos**

Nosso objetivo é testar se o controller se comporta corretamente: ele deve chamar o repositório, mapear as entidades para DTOs e retornar um OkObjectResult (Status 200) com a lista.

Vamos usar o padrão **Arrange, Act, Assert (AAA)**, que organiza o teste em 3 partes claras.

**Abra o arquivo ProdutosControllerTests.cs e substitua o conteúdo pelo código abaixo:**

C#

// GerenciadorDeProdutos.Tests/ProdutosControllerTests.cs

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using FluentAssertions;

using GerenciadorDeProdutos.Controllers;

using GerenciadorDeProdutos.DTOs;

using GerenciadorDeProdutos.Models;

using GerenciadorDeProdutos.Repositories.Interfaces;

using Microsoft.AspNetCore.Mvc;

using Moq;

using Xunit;

namespace GerenciadorDeProdutos.Tests;

public class ProdutosControllerTests

{

[Fact]

// Padrão de Nomenclatura: MetodoTestado\_Cenario\_ComportamentoEsperado

public async Task ListarTodosProdutos\_QuandoExistemProdutos\_DeveRetornarOkComListaDeProdutosDto()

{

// --- ARRANGE (Organizar) ---

// Aqui nós preparamos tudo o que o teste precisa: mocks, dados falsos, etc.

// 1. Criar dados falsos (uma lista de entidades 'Produto')

var produtosFalsos = new List<Produto>

{

new Produto { Id = 1, Nome = "Produto A", Preco = 10, CategoriaId = 1 },

new Produto { Id = 2, Nome = "Produto B", Preco = 20, CategoriaId = 1 }

};

// 2. Criar os Mocks das nossas dependências

var mockProdutoRepo = new Mock<IProdutoRepository>();

var mockUoW = new Mock<IUnitOfWork>();

// 3. Configurar o comportamento dos Mocks

// "Quando o método GetAllAsync do mockProdutoRepo for chamado,

// retorne a nossa lista de produtos falsos."

mockProdutoRepo.Setup(repo => repo.GetAllAsync()).ReturnsAsync(produtosFalsos);

// "Quando a propriedade ProdutoRepository do mockUoW for acessada,

// retorne o nosso mock de repositório de produtos."

mockUoW.Setup(uow => uow.ProdutoRepository).Returns(mockProdutoRepo.Object);

// 4. Instanciar o Controller, injetando o nosso Mock do Unit of Work.

// O controller vai pensar que está falando com o UoW de verdade, mas está

// falando com o nosso objeto falso e controlado.

var controller = new ProdutosController(mockUoW.Object);

// --- ACT (Agir) ---

// Aqui nós executamos a ação que queremos testar.

var resultado = await controller.ListarTodosProdutos();

// --- ASSERT (Verificar) ---

// Aqui nós verificamos se o resultado da ação foi o esperado.

// 1. O resultado deve ser do tipo 'OkObjectResult' (que corresponde ao status 200 OK)

resultado.Should().BeOfType<OkObjectResult>();

var okResult = resultado as OkObjectResult;

// 2. O valor dentro do resultado não deve ser nulo

okResult.Value.Should().NotBeNull();

// 3. O valor deve ser uma coleção de 'ProdutoDto'

okResult.Value.Should().BeAssignableTo<IEnumerable<ProdutoDto>>();

var listaDeDtos = okResult.Value as IEnumerable<ProdutoDto>;

// 4. A lista de DTOs deve ter a mesma quantidade de itens que nossa lista falsa

listaDeDtos.Should().HaveCount(produtosFalsos.Count);

// 5. O mapeamento de Entidade para DTO deve estar correto

listaDeDtos.First().Nome.Should().Be(produtosFalsos.First().Nome);

listaDeDtos.First().Preco.Should().Be(produtosFalsos.First().Preco);

}

}

**Análise do Código de Teste**

* **[Fact]**: É um atributo do xUnit que marca o método como um teste que deve ser executado.
* **new Mock<I...>()**: Cria um objeto falso que "imita" a interface. Podemos dizer a ele como se comportar.
* **.Setup(...)**: É aqui que configuramos o comportamento do mock. Dizemos "quando este método for chamado, faça isso".
* **.ReturnsAsync(...)**: Usado para métodos assíncronos, especifica o valor que o método do mock deve retornar.
* **mockUoW.Object**: Acessa o objeto falso configurado, que será passado para o construtor do controller.
* **.Should()**: É o ponto de entrada para as asserções do FluentAssertions, tornando a leitura quase como uma frase em inglês.

**Passo 4: Executar o Teste**

Agora que o teste está escrito, vamos executá-lo.

1. **Abra o terminal** na pasta raiz da solução ou na pasta do projeto de teste.

**Resumo Final**

Pronto! Agora seu projeto GerenciadorDeProdutos.Tests está 100% configurado através da interface do Visual Studio.

* O projeto de teste foi criado na sua solução.
* A referência ao projeto da API foi adicionada.
* Os pacotes Moq e FluentAssertions foram instalados.

**Conclusão da Aula**

Chegamos ao final da nossa jornada! Vamos recapitular o caminho que percorremos:

1. **Capítulo 1:** Começamos com uma base sólida, configurando o projeto, os modelos de banco de dados (Produto, Categoria) e o DbContext.
2. **Capítulo 2:** Fomos pela rota mais simples, usando o DbContext diretamente no Controller. Isso nos levou a um beco sem saída com o erro de **referência circular**, nos mostrando por que essa abordagem direta tem limitações.
3. **Capítulo 3:** Demos o primeiro passo para uma arquitetura limpa, introduzindo o **Padrão Repository**. Desacoplamos a lógica de acesso a dados dos controllers, tornando nosso código mais organizado e testável.
4. **Capítulo 4:** Resolvemos o problema da referência circular de forma elegante com **DTOs**. Aprendemos a moldar os dados para o cliente da API, criando um contrato seguro e estável.
5. **Capítulo 5:** Solidificamos o **Padrão Unit of Work**, entendendo seu papel vital em garantir a consistência dos dados através de operações atômicas, mesmo quando múltiplas entidades são modificadas em uma única requisição.

Você agora tem o conhecimento para evoluir de uma API simples para uma aplicação bem arquitetada, seguindo padrões de mercado que promovem manutenibilidade, testabilidade e robustez.

Espero que esta aula tenha sido clara e útil para você e seus alunos!