**WebApi Essential**

**Escopo**: Criar uma web api para um catalogo de produtos/categoria

* Criar um serviço RESTFull que permite que aplicativos clientes gerenciem o catalogo e produtos
* Expor endpoints para ler, criar, editar e excluir
* Para categoria precisamos armazenar o nome e o caminho da imagem
* Para Produtos precisamos armazenar nome: descrição valor unitário, caminho da imagem, estoque e data do cadastro e categoria

**Definição dos recursos, dos endpoints e do mapeamento**

ENDPOINTS 🡪 é a URL onde sei que os serviços podem ser acessados por uma aplicação cliente.

Endpoints da API: /v1/api/produtos

GET - /v1/api/produtos

GET - /v1/api/produtos/1

POST – /v1/api/produtos

PUT – v1/api/produtos/1

DELETE - /v1/api/produtos1

Endpoint: v1/api/categorias

GET - /v1/api/categorias

GET - /v1/api/categorias /1/produtos

POST – /v1/api/categorias

PUT – v1/api/categorias /1

DELETE - /v1/api/categorias/1

Respostas no formato json (para requests GET)

{

[

{“id”:1, “nome”:”categoria1”, “imagemurl”:”http:///teste/imagem1.jpg”},

{“id”:2, “nome”:”categoria1”, “imagemurl”:”http:///teste/imagem2.jpg”},

//outras

]

}

**Implementar a segurança**

- Permitir acesso às APIs somente usuários autenticados

- Definir uma política de acesso aos usuários

Criar um serviço RESTFull que permita gerenciar usuários

Expor endpoints para criar, ler, editar e excluir usuários e também pra consultar usuários e um usuário especifico

Para os usuários precisamos armazenar: nome, email e senha

**Mapeamento Endpoint da API: /v1/api/usuarios**

* GET - /v1/api/ usuarios
* GET - /v1/api/ usuarios /1
* POST – /v1/api/usuarios
* PUT – v1/api/categorias /1
* DELETE - /v1/api/ usuarios /1

**Códigos de status HTTP usados no Response**

* 200 – OK
* 201 – CREATED
* 202 – ACCEPTED
* 204 – No Content
* 304 – Not Modified
* 400 – Bad Request
* 401 – Unauthorized
* 403 – Forbidden
* 404 – Not Found
* 409 – Conflit
* 500 Internal Server Error

**Persistência dos dados**

Banco de dados relacional: SQL SERVER

Entity Framework para o ORM

Abordagem Code-First

Padrão Repository

**Nomenclatura**

Nome do Projeto: CatalogoApi

Nome dos Controllers – No Plural

ProdutosController, CategoriasController, UsuariosController

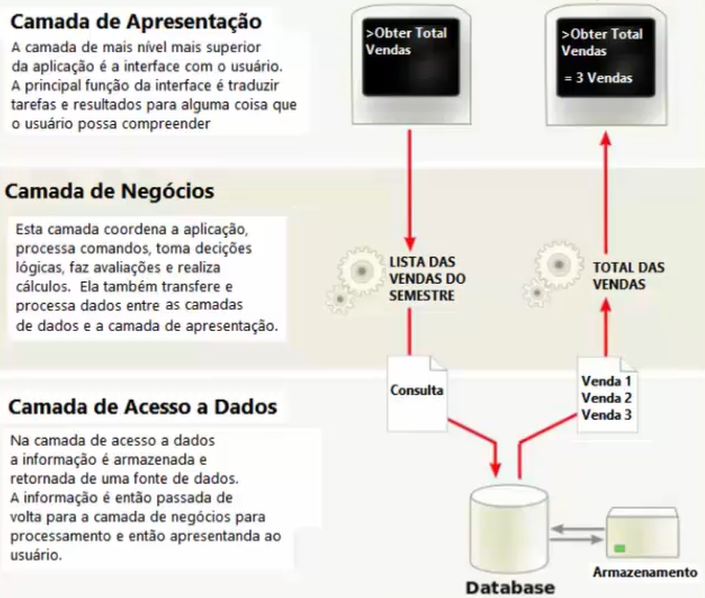
Nome das Actions – Acesso via verbos Http: Get, Put, Post, Delete

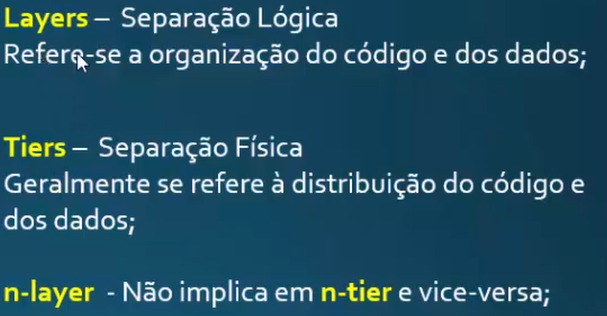
Usar DataAnottations: HttpGet. HttpPost, HttpPut, HttpDelete

Uma instancia de um recurso – no plural

Usar substantivos e não verbos - **/v1/api/produtos** e não /v1/api/ObterTodosProdutos

**Estrutura do Projeto**





Criar projetos distintos usando Class Library ou separar por pastas no mesmo projeto?

Usamos geralmente pastas no mesmo servidor.

**Estrutura de pastas**

1 – Presentation – React, Views, Mobile, Desktop

2 – Services – Classes de Serviços

3 – Bussiness Logic/ApplicationCore – repositórios, domínios serviços

4 – Data Access/Persistence – EfCore Sql server

Criando o projeto

1 – Criar o projeto – ApiCatalogo

2 – Criar o o projeto com opção para habilitar o OPEN API e usar Controllers

3 – Criar o modelo de Entidades – Produtos e Categoria

4 – Configurar o projeto para o EF CORE e incluir referencias

5 – definir o Banco de dados

6 – Definir a Classe de Context – AppDbContext

7 – Definit mapeamento das entidades – DbSet<T>

8 – Registrar o contexto como um serviços – Program

9 – Definit String de Conexão – AppSettings.json

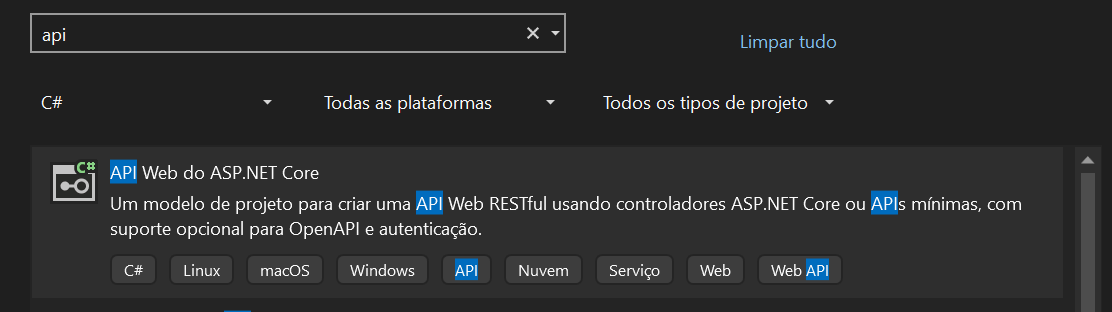
10 – Definir Provedor do banco de dados (SQL SERVER) e Obter String de Conexão

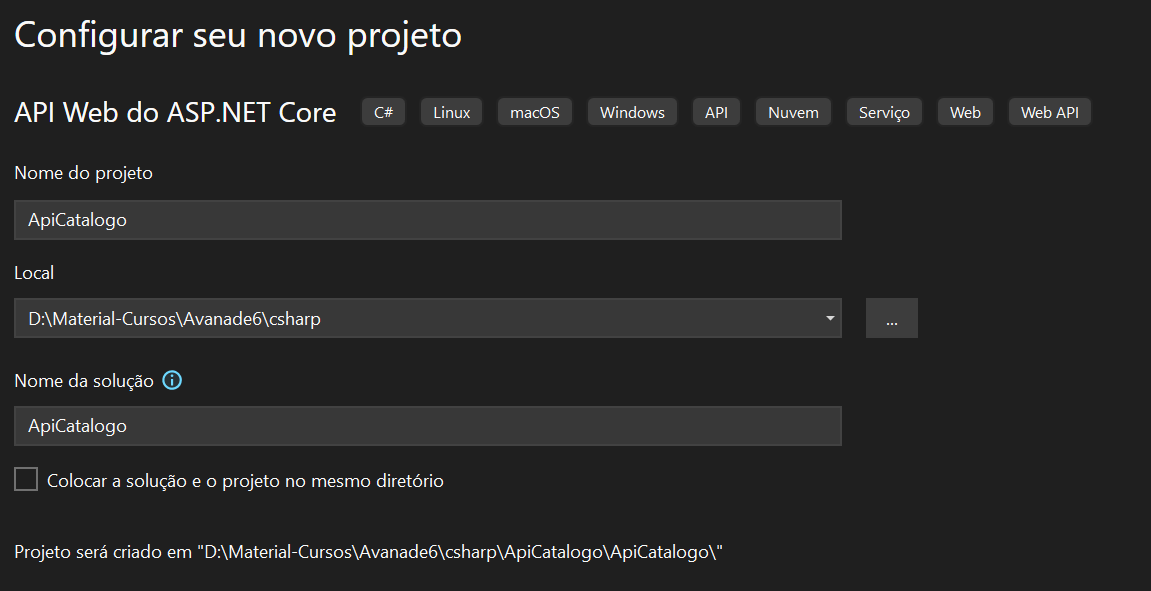
11 – Aplicar Migrations e Update-database

12 – Criar os Controllers: ProdutosController e CategoriasController

13 – Definir os EndPoints ou métodos Actions para realizar o CRUD

Criar o o projeto





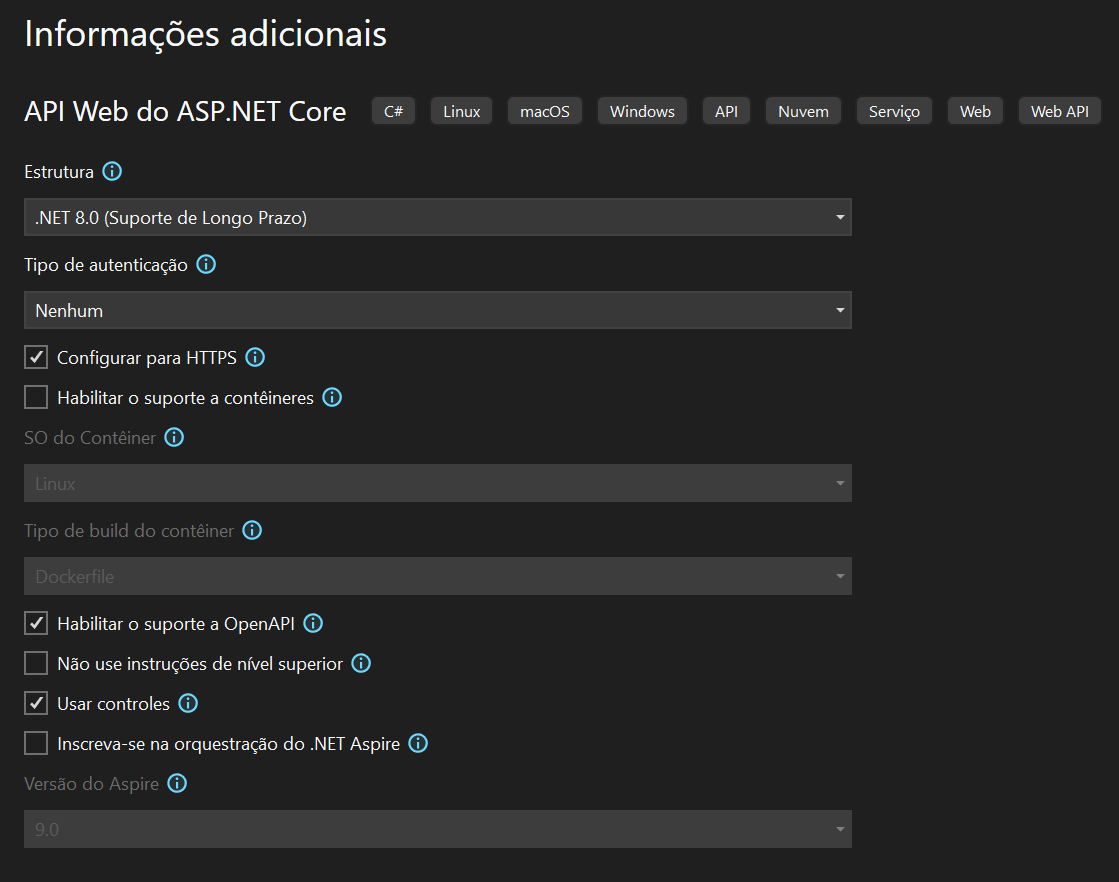


imagem mostra a estrutura de arquivos e pastas de um projeto **ASP.NET Core Web API**. Esta é uma estrutura padrão e muito comum para a criação de APIs na plataforma .NET.

Descriação de cada um dos componentes e sua importância dentro do projeto:

**Estrutura Principal**

* **Solução 'ApiCatalogo' (1 de 1 projeto)**
  + **Função:** Uma "Solução" (.sln) no Visual Studio é um contêiner que agrupa um ou mais projetos relacionados. Neste caso, a solução chamada ApiCatalogo contém apenas um projeto, que também se chama ApiCatalogo. Em cenários mais complexos, uma solução poderia conter o projeto da API, um projeto de testes, uma biblioteca de classes, etc.
  + **Importância:** Organiza e gerencia todos os projetos que compõem uma aplicação completa.
* **ApiCatalogo (o Projeto)**
  + **Função:** Este é o projeto principal da sua Web API. Ele contém todo o código, configurações e dependências necessárias para que a API funcione.
  + **Importância:** É o coração da sua aplicação, onde a lógica de negócio, o acesso a dados e os endpoints da API são definidos.

**Pastas e Arquivos do Projeto**

* **Connected Services (Serviços Conectados)**
  + **Função:** Gerencia a conexão com serviços externos, como serviços Azure, contêineres de publicação (Docker), e outras APIs. O Visual Studio fornece uma interface para adicionar e configurar essas conexões.
  + **Importância:** Facilita a integração com serviços em nuvem e outras dependências externas, automatizando parte da configuração necessária.
* **Dependências**
  + **Função:** Mostra todas as dependências do projeto. Isso inclui os pacotes NuGet (bibliotecas de terceiros), SDKs do .NET e referências a outros projetos dentro da mesma solução.
  + **Importância:** É fundamental para gerenciar as bibliotecas e frameworks que seu projeto utiliza, permitindo a reutilização de código e a adição de novas funcionalidades de forma modular.
* **Properties (Propriedades)**
  + **Função:** Contém arquivos de configuração relacionados ao processo de compilação e execução do projeto.
  + **launchSettings.json**: Este arquivo é um dos mais importantes dentro desta pasta. Ele define como a aplicação deve ser iniciada no ambiente de desenvolvimento. Por exemplo, ele especifica as URLs que a aplicação usará (com e sem HTTPS), as variáveis de ambiente a serem carregadas e se o navegador deve ser aberto automaticamente ao iniciar o projeto.

|  |  |  |
| --- | --- | --- |
| Para impedir que a página do Swagger seja aberta automaticamente ao executar seu projeto, você precisa fazer uma pequena alteração no arquivo launchSettings.json.  A forma mais direta é simplesmente desabilitar a inicialização do navegador.  **Passo a Passo:**   1. **Abra o arquivo launchSettings.json** que está dentro da pasta Properties do seu projeto. 2. **Encontre o perfil de inicialização que você está usando.** Geralmente é o perfil "httpss". Dentro desse perfil, você encontrará a linha "launchBrowser": true. 3. **Altere o valor de launchBrowser de true para false**.  |  |  | | --- | --- | | //...  "profiles": {  "httpss": {  "commandName": "Project",  "dotnetRunMessages": true,  "launchBrowser": true, // <-- Esta linha  "launchUrl": "swagger",  "applicationUrl": "https://localhost:7116;http://localhost:5005",  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Development"  }  }  }  //... | //...  "profiles": {  "httpss": {  "commandName": "Project",  "dotnetRunMessages": true,  "launchBrowser": false, // <-- Altere para false  "launchUrl": "swagger",  "applicationUrl": "https://localhost:7116;http://localhost:5005",  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Development"  }  }  }  //... | |

* + **Importância:** Permite configurar diferentes perfis de execução para depuração e teste, tornando o processo de desenvolvimento mais flexível.
* **Controllers**
  + **Função:** Esta é uma das pastas mais importantes em uma Web API. Ela contém as classes "Controladoras" (Controller). Cada controller agrupa um conjunto de "endpoints" ou "actions" relacionados a um recurso específico da sua aplicação.
  + **WeatherForecastController.cs**: Este é um controller de exemplo que a Microsoft inclui em novos projetos de Web API. Ele define endpoints para obter dados de previsão do tempo.
  + **Importância:** É a porta de entrada da sua API. Os controllers recebem as requisições HTTP (GET, POST, PUT, DELETE, etc.), processam a lógica necessária (muitas vezes com a ajuda de outros serviços) e retornam uma resposta para o cliente (geralmente em formato JSON).
* **ApiCatalogo.http**
  + **Função:** Este é um arquivo de texto que permite fazer requisições HTTP diretamente de dentro do Visual Studio para testar os endpoints da sua API. É uma alternativa a ferramentas externas como o Postman ou o Insomnia.
  + **Importância:** Agiliza o ciclo de desenvolvimento e teste, permitindo que você chame e verifique o comportamento dos seus endpoints sem sair do editor de código.
* **appsettings.json**
  + **Função:** Arquivo de configuração principal da aplicação no formato JSON. Ele é usado para armazenar configurações que não devem estar fixas no código, como strings de conexão com o banco de dados, chaves de API, e outras configurações de ambiente.
  + **Importância:** Centraliza as configurações da aplicação, permitindo que elas sejam facilmente alteradas sem a necessidade de recompilar o código. É comum ter arquivos variantes como appsettings.Development.json e appsettings.Production.json para diferentes ambientes.
* **Program.cs**
  + **Função:** É o ponto de entrada da sua aplicação. A partir das versões mais recentes do .NET, este arquivo ficou mais simplificado. É aqui que o servidor web é configurado e iniciado. Também é o local onde os serviços são registrados no contêiner de injeção de dependência e o pipeline de middleware HTTP é configurado (por exemplo, adicionar autenticação, autorização, roteamento, etc.).
  + **Importância:** É o arquivo de "bootstrap" da aplicação. Sem ele, a aplicação não sabe como iniciar, quais serviços carregar ou como responder às requisições.

|  |
| --- |
| Este arquivo, Program.cs, é o coração da sua aplicação ASP.NET Core. É aqui que tudo é configurado e iniciado. Nas versões modernas do .NET, ele foi simplificado para ser mais direto e legível.  O código pode ser dividido em duas partes principais:   1. **Configuração dos Serviços (Dependency Injection)** 2. **Configuração do Pipeline de Requisições HTTP (Middlewares)**   **1. Configuração dos Serviços (o "Builder")**  Esta é a primeira metade do arquivo, onde você "prepara" sua aplicação, registrando todos os serviços que ela precisará.  C#  var builder = WebApplication.CreateBuilder(args);  // Add services to the container.  builder.Services.AddControllers();  builder.Services.AddEndpointsApiExplorer();  builder.Services.AddSwaggerGen();   * var builder = WebApplication.CreateBuilder(args); Cria o "construtor" da aplicação (builder). É o objeto principal que você usa para configurar tudo, como o servidor web Kestrel, logs e injeção de dependência. * builder.Services.AddControllers(); Registra os serviços necessários para que os seus **Controllers** (como o WeatherForecastController.cs que vimos antes) funcionem corretamente. Isso inclui coisas como o roteamento de atributos e a validação de modelos. * builder.Services.AddEndpointsApiExplorer(); Este serviço é necessário para que ferramentas como o Swagger consigam "descobrir" os endpoints (as URLs) da sua API e entender como eles funcionam (quais parâmetros aceitam, o que retornam, etc.). * builder.Services.AddSwaggerGen(); Adiciona o gerador do **Swagger**. Ele usa as informações do AddEndpointsApiExplorer para criar automaticamente a documentação da sua API no formato OpenAPI (um arquivo chamado swagger.json).   **2. Configuração do Pipeline HTTP (o "App")**  Depois de registrar todos os serviços, você "constrói" a aplicação e define como ela deve lidar com as requisições HTTP que chegam. Isso é feito configurando um *pipeline* de middlewares. Pense em um pipeline como uma linha de montagem: cada requisição passa por cada um desses app.Use... em ordem.  C#  var app = builder.Build();  // Configure the HTTP request pipeline.  if (app.Environment.IsDevelopment())  {  app.UseSwagger();  app.UseSwaggerUI();  }  app.UseHttpsRedirection();  app.UseAuthorization();  app.MapControllers();  app.Run();   * var app = builder.Build(); Finaliza a fase de configuração e cria a instância da sua aplicação pronta para ser executada. * if (app.Environment.IsDevelopment()) Verifica se a aplicação está rodando no ambiente de **desenvolvimento**. Isso é definido no launchSettings.json. O código dentro deste if só será executado quando você estiver depurando no seu computador. * app.UseSwagger(); Adiciona o middleware que expõe o arquivo swagger.json gerado anteriormente. Sem isso, a documentação existe, mas não há como acessá-la pela web. * app.UseSwaggerUI(); Adiciona o middleware que serve a **interface gráfica** e interativa do Swagger. É a página web bonita que você vê e usa para testar sua API. Colocar esses dois dentro do if é uma boa prática de segurança, pois evita que a documentação da sua API seja exposta publicamente no ambiente de produção. * app.UseHttpsRedirection(); Redireciona automaticamente qualquer requisição http:// para https://. É uma camada importante de segurança. * app.UseAuthorization(); Ativa o sistema de autorização. Ele verifica se um usuário tem permissão para acessar um determinado endpoint (por exemplo, se ele está logado ou se tem a permissão "admin"). * app.MapControllers(); Este é o passo que efetivamente conecta as URLs às suas classes Controller. É ele quem lê os atributos de rota (como [HttpGet], [Route("api/[controller]")]) e garante que uma requisição para /WeatherForecast seja entregue ao WeatherForecastController. * app.Run(); Inicia o servidor e faz com que a aplicação comece a "ouvir" as requisições HTTP. Este é o comando final que coloca sua API no ar. |

* **WeatherForecast.cs**
  + **Função:** Esta é uma classe de modelo (Model) ou um Objeto de Transferência de Dados (DTO - Data Transfer Object). Ela define a estrutura dos dados que serão manipulados pela aplicação. No caso do exemplo, ela define as propriedades de um objeto de previsão do tempo (Data, Temperatura, Resumo, etc.).
  + **Importância:** Modela os dados da sua aplicação, garantindo uma estrutura consistente e tipada para as informações que são recebidas e enviadas através da API.

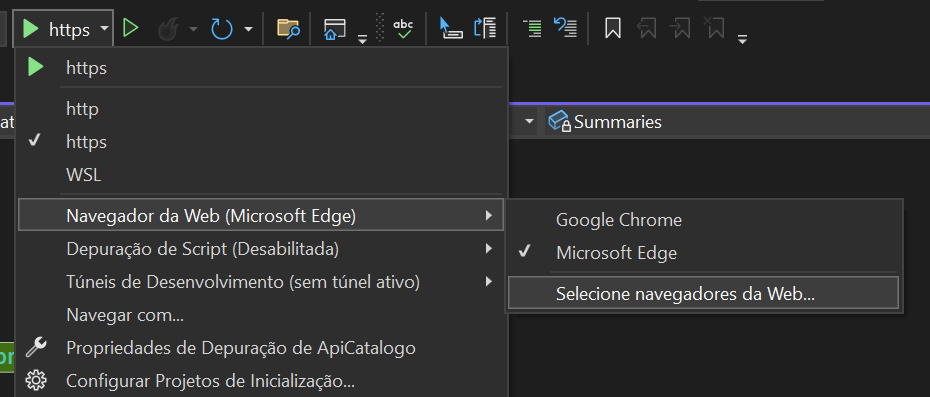
**Resumo da Importância na Estrutura Web API**

Essa estrutura segue o padrão **Model-View-Controller (MVC)**, adaptado para APIs.

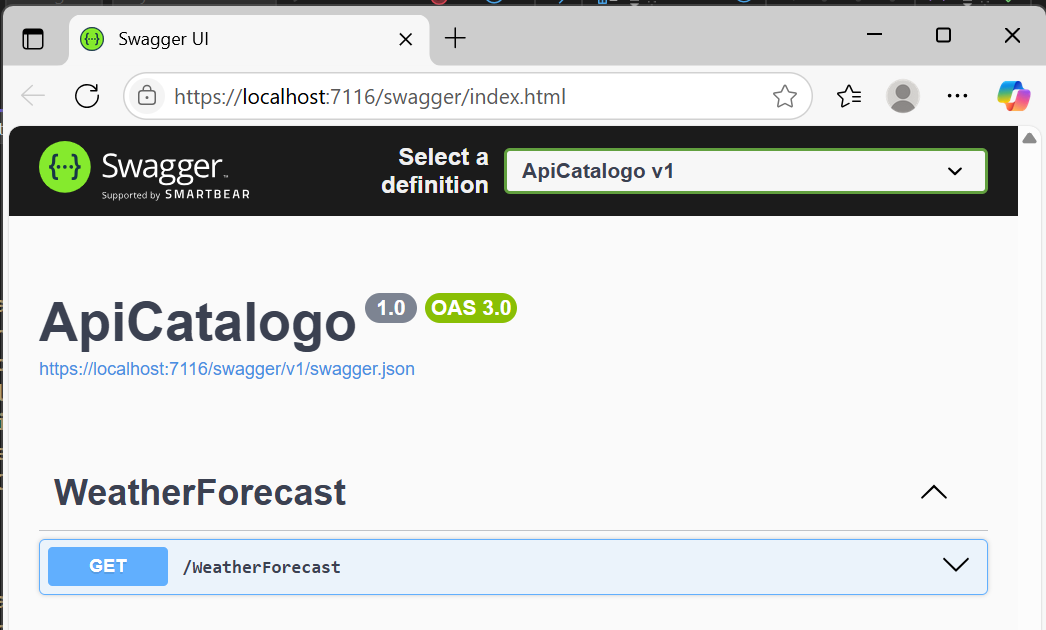
* O **Model** (WeatherForecast.cs) representa os dados.
* O **Controller** (WeatherForecastController.cs) manipula as requisições do usuário e retorna os dados.
* A **View** (Visão), em uma Web API, é a própria representação dos dados retornados, geralmente em formato JSON ou XML.

O restante dos arquivos (Program.cs, appsettings.json, etc.) forma a infraestrutura de suporte que configura, inicia e gerencia a aplicação e suas dependências, tornando o desenvolvimento robusto, escalável e organizado.

O navegador pode ser escolhido:



O que é apresentado ao executar a primeira vez o projeto



Criando o Modelo de Dominio

Delete a Classe e o controlador WeatherForeCast

Criar 2 classes Produto e Categoria

Criar uma pasta chamada Models (poderia ser Domain também)

|  |
| --- |
| namespace ApiCatalogo.Models  {  public class Produto  {  public int ProdutoId { get; set; }  public string? Nome { get; set; }  public string? Descricao { get; set; }  public decimal Preco { get; set; }  public string? ImagemUrl { get; set; }  public float Estoque { get; set; }  public DateTime DataCadastro { get; set; }  }  } |

|  |
| --- |
| namespace ApiCatalogo.Models  {  public class Categoria  {  public int CategoriaId { get; set; }  public string? Nome { get; set; }  public string? ImagemUrl { get; set; }  }  } |

Classes Anêmicas - No contexto do **C# e da programação orientada a objetos (POO)**, uma **classe anêmica** (ou **modelo anêmico**, do inglês Anemic Domain Model) é uma classe que **possui apenas propriedades (getters e setters)**, mas **não contém lógica de negócio** associada aos dados que ela representa.

**🛠 Consequências do Modelo Anêmico**

**❌ Problemas:**

* **Falta de coesão:** o objeto não é responsável por seu próprio comportamento.
* **Baixo encapsulamento:** as regras ficam espalhadas em serviços externos.
* **Dificuldade para manutenção:** lógica duplicada ou mal localizada.
* **Quebra da orientação a objetos:** transforma o código em um estilo procedural disfarçado.

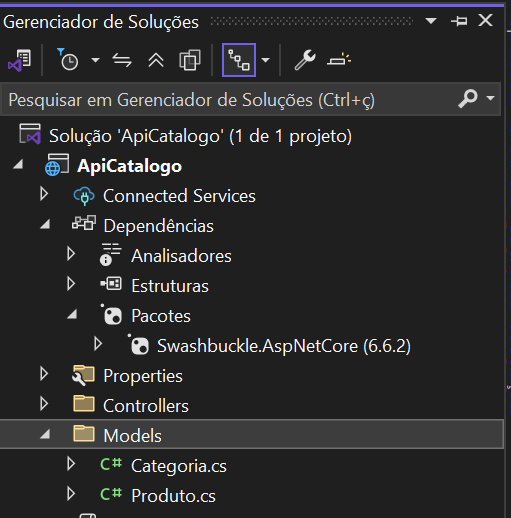
**✅ Quando pode ser aceitável:**

* Em **arquiteturas anêmicas intencionais**, como em **aplicações CRUD simples** ou **DTOs (Data Transfer Objects)**.
* Em **padrões arquiteturais como CQRS**, onde comandos e queries são separados.

NO nosso caso as informações estarão no BD.

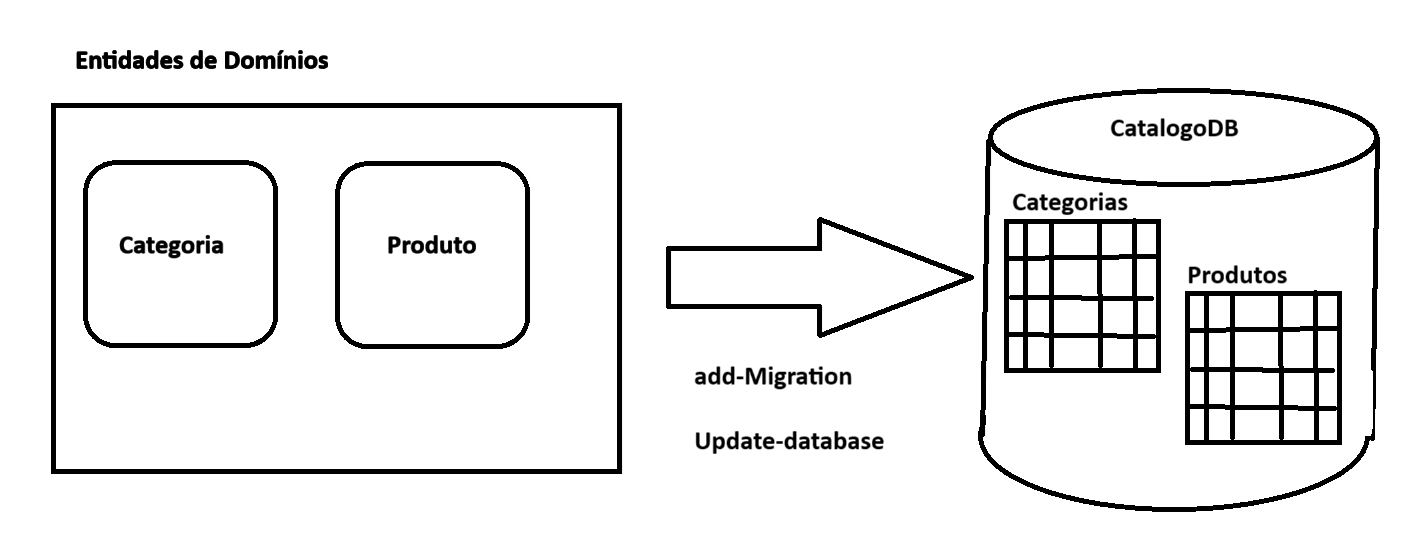
Configuação do EF Core

Verifique os pacotes instalados



Usaremos a abordagem Code-First – Vamos das classes para o Banco –

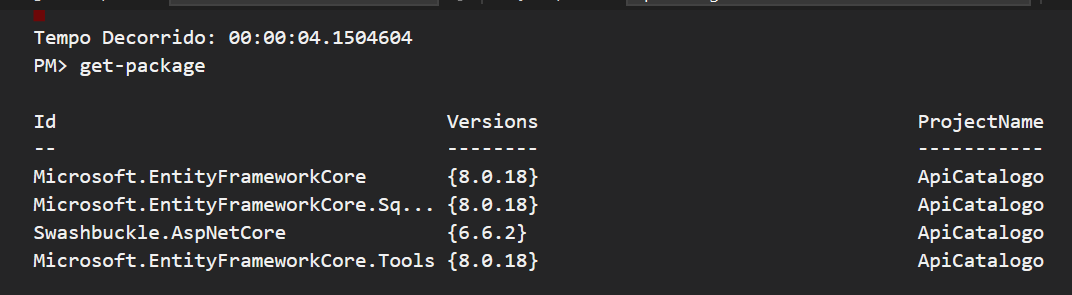
Existem convenções adotadas pelo EF Core que devemos seguir



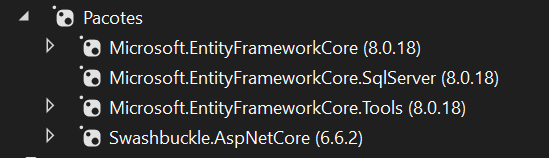
Instatalar os packages necessários



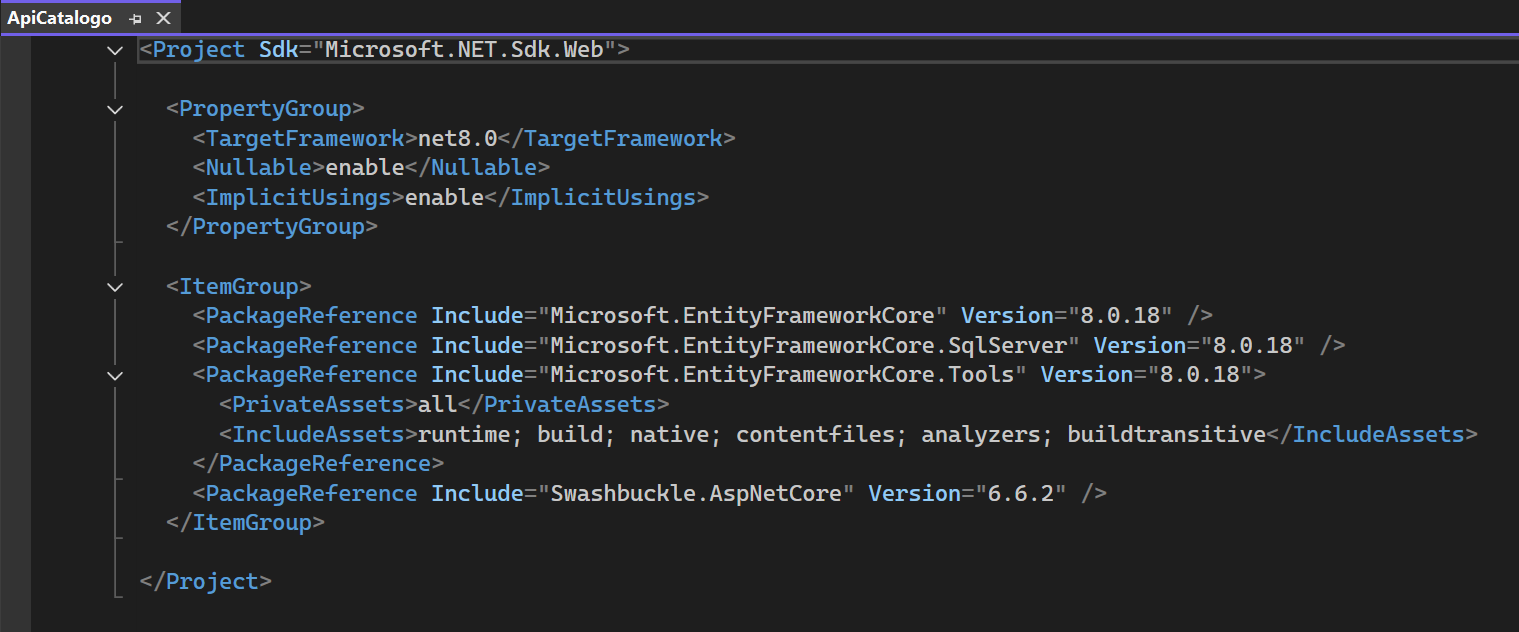
Usando o PMC (package Manage Console) digite o comando get-package



Pode verificar também



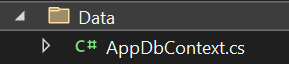
Ou ainda vendo no arquivo



**Criando a Classe Context**

Crie uma pasta chamada Data (poderia ser também Contexto, por exemplo)

Desntro desta pasta crie um arquivo chamado AppDbContext



|  |
| --- |
| using ApiCatalogo.Models;  using Microsoft.EntityFrameworkCore;  namespace ApiCatalogo.Data  {  public class AppDbContext: DbContext  {  public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)  {  }  public DbSet<Produto> Produtos { get; set; }  public DbSet<Categoria> Categorias { get; set; }    }  } |

**📦 Arquivo: AppDbContext.cs**

using ApiCatalogo.Models;

🔹 **Importa o namespace onde estão definidos os modelos de domínio**, neste caso Produto e Categoria.  
Essas classes representam as tabelas no banco de dados.

using Microsoft.EntityFrameworkCore;

🔹 Importa o **namespace do Entity Framework Core**, necessário para usar as classes base como DbContext, DbSet<T>, e os recursos de mapeamento ORM (Object-Relational Mapping).

namespace ApiCatalogo.Data

🔹 Define o **namespace do arquivo**, organizando o código logicamente. Neste caso, a classe AppDbContext está agrupada sob ApiCatalogo.Data, indicando que faz parte da camada de dados.

public class AppDbContext : DbContext

🔹 Define a classe AppDbContext que **herda de DbContext**, a classe base do EF Core usada para:

* Estabelecer a conexão com o banco
* Mapear os modelos (entidades) para tabelas
* Permitir consultas e alterações com LINQ

csharp

CopiarEditar

public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)

🔹 Construtor que **recebe as configurações do contexto via injeção de dependência** (ex: string de conexão, provedor SQL Server).  
🔸 DbContextOptions<AppDbContext> encapsula essas configurações.  
🔸 : base(options) repassa essas configurações para a classe base DbContext.

➡️ **Essa linha é fundamental** para permitir que o EF Core seja configurado corretamente em Program.cs ou Startup.cs com services.AddDbContext<AppDbContext>(...).

csharp

CopiarEditar

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

🔹 Define uma **propriedade do tipo DbSet<Produto>**, que representa uma **tabela Produtos** no banco.  
🔸 O EF Core irá mapear a entidade Produto para essa tabela, permitindo realizar operações CRUD.

➡️ Exemplo de uso:

var lista = context.Produtos.ToList(); // SELECT \* FROM Produtos

public DbSet<Categoria> Categorias { get; set; }

🔹 Mesmo conceito anterior, mas agora para a entidade Categoria.  
➡️ Essa tabela representará as **categorias associadas aos produtos**, possibilitando implementar relações como **1:N** (uma categoria → vários produtos).

**🧠 Estrutura completa explicada**

| **Elemento** | **Função** |
| --- | --- |
| DbContext | Classe base do EF Core para manipulação do banco |
| DbSet<T> | Representa uma coleção (tabela) de entidades do tipo T |
| options | Configurações do contexto (string de conexão, provider etc.) |
| AppDbContext | Classe personalizada que representa o banco no código (ORM) |

**🔄 Conexão no Program.cs (exemplo para completar)**

E no appsettings.json:

{

"ConnectionStrings": {

"**DefaultConnection**": "Server=localhost;Database=CatalogoDb;Trusted\_Connection=True; TrustServerCertificate=True"

}

}

Para que tudo funcione, você configura esse contexto no Program.cs assim:

|  |
| --- |
| builder.Services.AddDbContext<AppDbContext>(options =>  options.UseSqlServer(builder.Configuration.GetConnectionString("**DefaultConnection**"))); |

**✅ Resumo Final**

Este código define a **ponte entre seu sistema e o banco de dados**, onde as entidades Produto e Categoria são mapeadas para tabelas. O EF Core usa essa classe para **consultar, salvar, atualizar e deletar** dados sem você escrever SQL diretamente.

**Definindo o relacionamento entre tabelas**

Tabela Categoria

|  |  |  |
| --- | --- | --- |
| CategoriaId | Nome | ImagemUrl |
| 1 | Bebidas | Categoria1.jpg |
| 2 | Sanduíches | Categoria2.jpg |

Tabela Produto

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ProdutoId | Nome | Descricao | preco | ImagemUrl | Estoque | DataCadastro |
| 1 | Produto1 | Desc1 | 10 | Prod1.jpg | 5 | 01/02/2025 |
| 2 | Produto2 | Desc2 | 11 | Prod2.jpg | 7 | 05/02/2025 |
| 3 | Produto3 | Desc3 | 12 | Prod3.jpg | 9 | 11/02/205 |
| 4 | Produto4 | Desc4 | 15 | Prod4.jpg | 10 | 13/02/2025 |

Uma categoria pode ter um ou mais Produtos

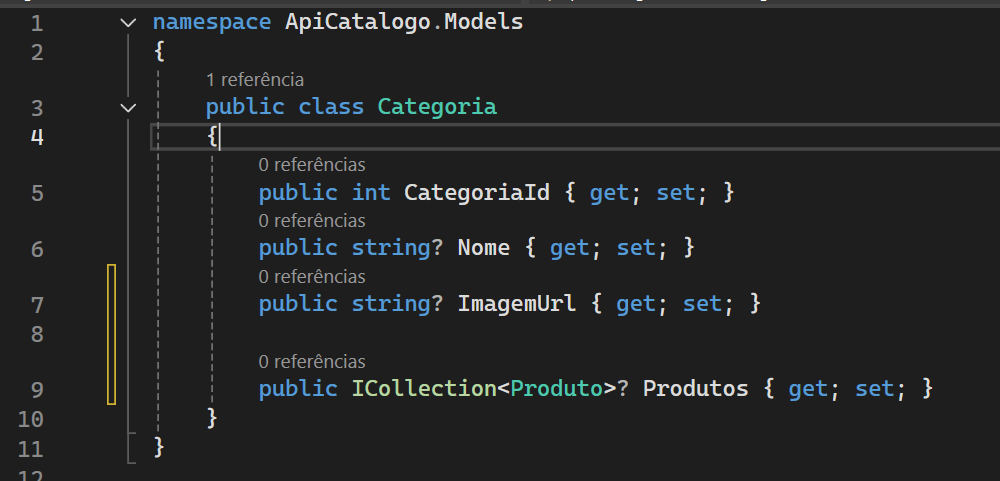
Relacionamento um-para-muitos entre as tabelas categoria e a tabela produtos

Acrescentar a coluna categoriaId na tabela Produto

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ProdutoId | Nome | Descricao | preco | ImagemUrl | Estoque | DataCadastro | CategoriaId |
| 1 | Produto1 | Desc1 | 10 | Prod1.jpg | 5 | 01/02/2025 | 1 |
| 2 | Produto2 | Desc2 | 11 | Prod2.jpg | 7 | 05/02/2025 | 1 |
| 3 | Produto3 | Desc3 | 12 | Prod3.jpg | 9 | 11/02/205 | 2 |
| 4 | Produto4 | Desc4 | 15 | Prod4.jpg | 10 | 13/02/2025 | 1 |

Como fazer?

O mínimo necessário é incluir uma propriedade de navegação na tabela principal no nosso caso Categoria



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

📌 Significa uma relação de 1 para muitos (1:N) — ou seja, uma Categoria possui vários Produtos.

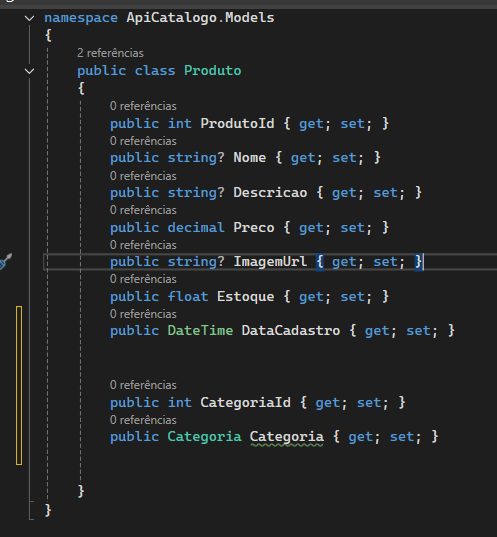
📘 Explicação técnica:

ICollection<Produto>: indica que esta propriedade é uma coleção de objetos Produto.

EF Core interpreta isso como uma navegação reversa: a entidade Produto deve conter uma propriedade de chave estrangeira CategoriaId e, opcionalmente, uma referência do tipo Categoria.

O uso de ? (nullable) permite que a propriedade seja opcional (usada em C# 8+ com nullable enabled).

Incluímos na tabela muitos da relação um propriedade qeu vai mapear a chave estrangeia indicando quem um produto esta relacionada a uma categoria



public int CategoriaId { get; set; }

public Categoria Categoria { get; set; }

formam, no contexto do Entity Framework Core, a definição de uma chave estrangeira (FK) e a propriedade de navegação que representa a relação muitos-para-um (N:1), ou seja:

📌 O que significa no EF Core:

🔹 public int CategoriaId { get; set; }

Representa a chave estrangeira da tabela Produtos.

Cada produto estará associado a uma categoria existente.

O EF Core reconhece esse campo como o FK ligando Produto → Categoria.

🔹 public Categoria Categoria { get; set; }

É a propriedade de navegação.

Permite acessar a categoria relacionada diretamente pelo objeto Produto, ex.:

var nomeCategoria = produto.Categoria.Nome;

🧭 Relação definida:

Essa configuração permite que o EF Core entenda que:

Um Produto pertence a uma Categoria (N:1)

Uma Categoria tem vários Produtos (1:N) — isso está representado na entidade Categoria com:

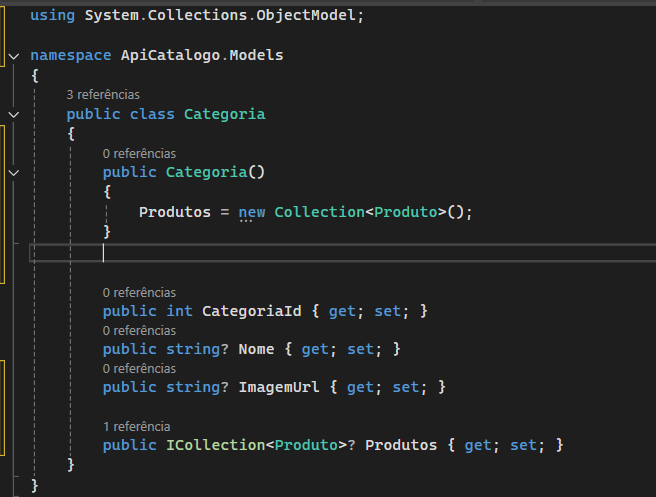
public ICollection<Produto>? Produtos { get; set; }

🔄 Carregamento com EF:

Você pode usar Include() para carregar a categoria junto com o produto:

var produtos = context.Produtos.Include(p => p.Categoria).ToList();

Devemos modifica o Construto da classe Categoria



A razão de termos o seguinte **construtor na classe Categoria**:

public Categoria()

{

Produtos = new Collection<Produto>();

}

é garantir que a **propriedade de navegação Produtos** (que representa a coleção de produtos associados à categoria) **seja sempre inicializada ao instanciar a classe**, mesmo antes de acessar o banco de dados.

## 🎯 Finalidade do Construtor

### 🔒 1. Evitar NullReferenceException

Sem o construtor, ao tentar adicionar um produto à coleção, teríamos que checar se ela está nula:

if (categoria.Produtos == null)

categoria.Produtos = new Collection<Produto>();

categoria.Produtos.Add(produto);

Com o construtor, isso é desnecessário:

categoria.Produtos.Add(produto); // Seguro, pois a lista já está inicializada

### 🔁 2. Permitir uso imediato da lista

Isso é especialmente útil em operações de criação de dados em memória, por exemplo:

var novaCategoria = new Categoria

{

Nome = "Eletrônicos"

};

novaCategoria.Produtos.Add(new Produto { Nome = "Notebook", Preco = 3000 });

Se a coleção Produtos **não fosse inicializada**, o código acima falharia.

## 📚 Por que Collection<Produto>?

* Collection<T> é uma implementação concreta de ICollection<T>, da namespace System.Collections.ObjectModel.
* Poderia ser List<Produto> também. Ambos são válidos como implementação de ICollection<T>, mas Collection<T> é preferido quando se quer mais controle (por exemplo, sobrescrever métodos como Insert, Remove etc.).

## ✅ Conclusão

O construtor:

* **Previne erros de execução**
* **Facilita a manipulação da entidade antes da persistência**
* **É uma boa prática no mapeamento 1:N do EF Core**

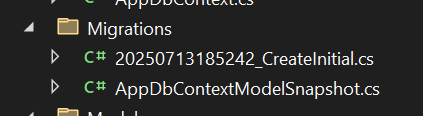
Confirm a declaração da injeção no Program.cs

|  |
| --- |
| using ApiCatalogo.Data;  using Microsoft.EntityFrameworkCore;  var builder = WebApplication.CreateBuilder(args);  // Add services to the container.  // Recupera a string de conexão do appsettings.json  var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");  // Configura o DbContext com SQL Server e o certificado confiável  builder.Services.AddDbContext<AppDbContext>(options =>  options.UseSqlServer(connectionString));  builder.Services.AddControllers();  // Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle  builder.Services.AddEndpointsApiExplorer();  builder.Services.AddSwaggerGen();  //restante do código |

Pode fazer o Migration e Update-database

1. Add-migration CreateInitial
2. Update-database

Após a execução destes comandos teremos um novo arquivo dentro de uma nova pasta



O arquivo possui a data de criação e um hash de identificação que tb existe no banco de dados

Esse arquivo gerado (CreateInitial) é uma **classe de migração** criada automaticamente pelo Entity Framework Core após o comando:

bash

CopiarEditar

dotnet ef migrations add CreateInitial

Essa migração serve para **criar a estrutura inicial do banco de dados** com base nas classes de modelo (Categoria e Produto) definidas no projeto.

**Estrutura Técnica da Classe**

A classe herda de Migration e sobrescreve dois métodos:

**🔼 Up(MigrationBuilder migrationBuilder)**

Este método define o que deve acontecer **ao aplicar a migração**, ou seja, ao criar ou atualizar o banco. Aqui, são criadas duas tabelas:

**📌 Tabela Categorias**

* **CategoriaId** (chave primária, auto incremento)
* **Nome** (nullable)
* **ImagemUrl** (nullable)

migrationBuilder.CreateTable(

name: "Categorias",

columns: table => new {

CategoriaId = table.Column<int>(type: "int", nullable: false)

.Annotation("SqlServer:Identity", "1, 1"),

Nome = table.Column<string>(type: "nvarchar(max)", nullable: true),

ImagemUrl = table.Column<string>(type: "nvarchar(max)", nullable: true)

},

constraints: table => {

table.PrimaryKey("PK\_Categorias", x => x.CategoriaId);

});

**📌 Tabela Produtos**

* **ProdutoId** (chave primária, auto incremento)
* **Nome**, **Descricao**, **Preco**, **ImagemUrl**, **Estoque**, **DataCadastro**
* **CategoriaId**: Chave estrangeira relacionada à tabela Categorias

migrationBuilder.CreateTable(

name: "Produtos",

columns: table => new {

ProdutoId = table.Column<int>(type: "int", nullable: false)

.Annotation("SqlServer:Identity", "1, 1"),

Nome = table.Column<string>(type: "nvarchar(max)", nullable: true),

Descricao = table.Column<string>(type: "nvarchar(max)", nullable: true),

Preco = table.Column<decimal>(type: "decimal(18,2)", nullable: false),

ImagemUrl = table.Column<string>(type: "nvarchar(max)", nullable: true),

Estoque = table.Column<float>(type: "real", nullable: false),

DataCadastro = table.Column<DateTime>(type: "datetime2", nullable: false),

CategoriaId = table.Column<int>(type: "int", nullable: false)

},

constraints: table => {

table.PrimaryKey("PK\_Produtos", x => x.ProdutoId);

table.ForeignKey(

name: "FK\_Produtos\_Categorias\_CategoriaId",

column: x => x.CategoriaId,

principalTable: "Categorias",

principalColumn: "CategoriaId",

onDelete: ReferentialAction.Cascade);

});

Também é criado um índice para melhorar performance nas consultas por categoria:

migrationBuilder.CreateIndex(

name: "IX\_Produtos\_CategoriaId",

table: "Produtos",

column: "CategoriaId");

**🔽 Down(MigrationBuilder migrationBuilder)**

Esse método **reverte as alterações** feitas no Up, útil quando for necessário desfazer a migração. Aqui, ele apenas exclui as tabelas criadas:

migrationBuilder.DropTable(name: "Produtos");

migrationBuilder.DropTable(name: "Categorias");

**📚 Localização no Projeto**

Esse arquivo geralmente se encontra na pasta:

/Migrations/CreateInitial.cs

Ele é gerado automaticamente pelo EF Core e versiona o estado atual do banco.

**DATA ANNOTTATIONS**

Data Annottations – São Conjuntos de atributos que podem ser plicados a classes e seus membros ara fornecer metadados sobre como esses recursos deve ser tratados pelo sistema.

Para que servem?

-Realizar validações de entrada de dados

- influenciar o comportamento do modelo de dados

- Geração de código

- Especificar o relacionamento entre as entidades

- Sobrescrever as convenções do EF Core

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Segue uma **lista das Data Annotations mais utilizadas no C# com Entity Framework Core (EF Core)**, organizadas por finalidade (validação, mapeamento de banco de dados, relacionamento, etc.), com **explicações técnicas**, **exemplos práticos de uso** e sua **localização típica (Model)**.  **📌 1. Validação de Dados (System.ComponentModel.DataAnnotations)**  Essas anotações ajudam na validação dos dados antes de persistir no banco e funcionam bem com formulários no ASP.NET.   | **Anotação** | **Descrição** | **Exemplo** | | --- | --- | --- | | [Required] | Campo obrigatório | [Required] public string Nome { get; set; } | | [StringLength] | Define o tamanho máximo (e mínimo) de uma string | [StringLength(100, MinimumLength = 5)] | | [MaxLength] | Tamanho máximo (não valida, apenas mapeia no banco) | [MaxLength(50)] | | [MinLength] | Tamanho mínimo | [MinLength(10)] | | [Range] | Valida intervalo de valores | [Range(0, 100)] public int Idade { get; set; } | | [EmailAddress] | Valida formato de email | [EmailAddress] public string Email { get; set; } | | [Phone] | Valida número de telefone | [Phone] public string Telefone { get; set; } | | [Compare] | Compara dois campos (ex: senha e confirmação) | [Compare("Senha")] public string ConfirmarSenha { get; set; } | | [RegularExpression] | Validação por expressão regular | [RegularExpression(@"^\d{5}-\d{3}$")] (CEP) |   **🗂️ 2. Mapeamento de Banco de Dados (System.ComponentModel.DataAnnotations.Schema)**  Essas são úteis para personalizar a forma como as classes são mapeadas no banco (ex: nomes de tabelas e colunas).   | **Anotação** | **Descrição** | **Exemplo** | | --- | --- | --- | | [Key] | Define a chave primária (se não seguir a convenção Id ou ClassNameId) | [Key] public int Codigo { get; set; } | | [Column] | Personaliza o nome e tipo da coluna | [Column("NomeProduto", TypeName = "varchar(100)")] | | [Table] | Personaliza o nome da tabela | [Table("Clientes")] antes da classe | | [NotMapped] | Ignora a propriedade no mapeamento do EF | [NotMapped] public string NomeCompleto => $"{Nome} {Sobrenome}"; | | [DatabaseGenerated] | Define como o valor da chave será gerado | [DatabaseGenerated(DatabaseGeneratedOption.Identity)] |   **🔗 3. Relacionamentos (EF Core)**  Essas anotações ajudam a definir relacionamentos entre tabelas.   | **Anotação** | **Descrição** | **Exemplo** | | --- | --- | --- | | [ForeignKey] | Define a chave estrangeira explicitamente | [ForeignKey("CategoriaId")] | | [InverseProperty] | Resolve ambiguidade em relações | [InverseProperty("Produtos")] | |

OBS IMPORTANTE

o **EF Core permite configurar valores decimais com precisão e escala** diretamente no método OnModelCreating usando a **Fluent API**, por meio da chamada ao método .HasColumnType("decimal(precision, scale)").

**✅ Definição técnica**

csharp

CopiarEditar

modelBuilder.Entity<Classe>()

.Property(c => c.PropriedadeDecimal)

.HasColumnType("decimal(18,2)");

* **decimal(18,2)**: define um tipo decimal com:
  + **18 dígitos no total (precisão)**
  + **2 dígitos após a vírgula (escala)**

**🧪 Exemplo prático completo**

**Model Produto.cs**

public class Produto

{

public int ProdutoId { get; set; }

[Required]

public string Nome { get; set; }

public decimal Preco { get; set; } // será configurado no OnModelCreating

}

**DbContext com configuração Fluent**

csharp

CopiarEditar

public class AppDbContext : DbContext

{

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

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Produto>()

.Property(p => p.Preco)

.HasColumnType("decimal(10,2)");

}

}

⚠️ **Importante**: Sem essa configuração, o EF Core pode gerar o campo Preco como decimal(18,2) por padrão (ou até float em alguns bancos como SQLite, o que pode causar perda de precisão!).

### 📍 Localização dessa configuração

Normalmente está em:

Projeto/

│

├── Data/

│ └── AppDbContext.cs ← onde fica o OnModelCreating

Após estas modificações atribua os data annottations e faça uma nova Migration

Add-migration AlteraTabelas

Update-database

**Popular Tabelas com Dados**

Podemos fazer de 4 formas:

* + Incluir dados manualmente
  + Cria no método OnModelCreating do arquivo de contexto e definir o código usando a propriedade HasData do Ef Core para preencher as tabelas com dados

|  |
| --- |
| using Microsoft.EntityFrameworkCore;  using ApiCatalogo.Models;  namespace ApiCatalogo.Context  {  public class AppDbContext : DbContext  {  public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) {}  public DbSet<Produto> Produtos { get; set; }  public DbSet<Categoria> Categorias { get; set; }  protected override void OnModelCreating(ModelBuilder modelBuilder)  {  // Categorias  modelBuilder.Entity<Categoria>().HasData(  new Categoria { CategoriaId = 1, Nome = "Bebidas", ImagemUrl = "bebidas.jpg" },  new Categoria { CategoriaId = 2, Nome = "Lanches", ImagemUrl = "lanches.jpg" },  new Categoria { CategoriaId = 3, Nome = "Sobremesas", ImagemUrl = "sobremesas.jpg" }  );  // Produtos  modelBuilder.Entity<Produto>().HasData(  new Produto  {  ProdutoId = 1,  Nome = "Coca-Cola",  Descricao = "Refrigerante 2L",  Preco = 7.5m,  ImagemUrl = "coca.jpg",  Estoque = 100,  DataCadastro = DateTime.Now, // ou uma data fixa  CategoriaId = 1  },  new Produto  {  ProdutoId = 2,  Nome = "X-Burguer",  Descricao = "Pão, carne e queijo",  Preco = 15.0m,  ImagemUrl = "xburguer.jpg",  Estoque = 50,  DataCadastro = DateTime.Now,  CategoriaId = 2  },  new Produto  {  ProdutoId = 3,  Nome = "Sorvete Napolitano",  Descricao = "1L sabor napolitano",  Preco = 20.0m,  ImagemUrl = "sorvete.jpg",  Estoque = 30,  DataCadastro = DateTime.Now,  CategoriaId = 3  }  );  }  }  } |

* + Criar um método estático e definir o código para incluir dados usando o método AddRange() do EF Core

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A seguir, vou mostrar **como criar um método estático separado para fazer o *data seeding* com AddRange()** utilizando EF Core.  Esse tipo de abordagem é ideal para executar *seeds* programáticos de forma manual (fora do OnModelCreating) — por exemplo, no Program.cs, durante a inicialização da aplicação.  **✅ 1. Criar a classe DbInitializer.cs**  Local: Context/DbInitializer.cs  csharp  CopiarEditar  using ApiCatalogo.Models;  using System;  using System.Collections.Generic;  using System.Linq;  namespace ApiCatalogo.Context  {  public static class DbInitializer  {  public static void Seed(AppDbContext context)  {  // Aplica migrações pendentes (opcional)  context.Database.EnsureCreated();  // Verifica se já existem categorias  if (context.Categorias.Any())  return; // já foi populado  // Categorias  var categorias = new List<Categoria>  {  new Categoria { CategoriaId = 1, Nome = "Bebidas", ImagemUrl = "bebidas.jpg" },  new Categoria { CategoriaId = 2, Nome = "Lanches", ImagemUrl = "lanches.jpg" },  new Categoria { CategoriaId = 3, Nome = "Sobremesas", ImagemUrl = "sobremesas.jpg" }  };  context.Categorias.AddRange(categorias);  context.SaveChanges();  // Produtos  var produtos = new List<Produto>  {  new Produto  {  ProdutoId = 1,  Nome = "Coca-Cola",  Descricao = "Refrigerante de Cola 2L",  Preco = 7.5m,  ImagemUrl = "coca.jpg",  Estoque = 100,  DataCadastro = DateTime.Now,  CategoriaId = 1  },  new Produto  {  ProdutoId = 2,  Nome = "X-Burguer",  Descricao = "Lanche com carne e queijo",  Preco = 15.0m,  ImagemUrl = "xburguer.jpg",  Estoque = 50,  DataCadastro = DateTime.Now,  CategoriaId = 2  },  new Produto  {  ProdutoId = 3,  Nome = "Sorvete Napolitano",  Descricao = "Sorvete 1L sabor napolitano",  Preco = 20.0m,  ImagemUrl = "sorvete.jpg",  Estoque = 30,  DataCadastro = DateTime.Now,  CategoriaId = 3  }  };  context.Produtos.AddRange(produtos);  context.SaveChanges();  }  }  }  **✅ 2. Chamar o método no Program.cs**  Abra o arquivo Program.cs e adicione a chamada para o seed manual.  **Para ASP.NET Core 7 ou superior (com WebApplication.CreateBuilder):**  csharp  CopiarEditar  using ApiCatalogo.Context;  using Microsoft.Extensions.DependencyInjection;  var builder = WebApplication.CreateBuilder(args);  // Adiciona os serviços  builder.Services.AddDbContext<AppDbContext>();  builder.Services.AddControllers();  var app = builder.Build();  // Cria escopo e executa o seed  using (var scope = app.Services.CreateScope())  {  var services = scope.ServiceProvider;  var context = services.GetRequiredService<AppDbContext>();  DbInitializer.Seed(context); // <-- chamada do método estático  }  app.UseHttpsRedirection();  app.UseAuthorization();  app.MapControllers();  app.Run();  **📌 Comparativo: AddRange vs HasData**   | **Característica** | **AddRange()** | **HasData()** | | --- | --- | --- | | Onde é executado | Código de runtime (Program.cs) | OnModelCreating (durante migração) | | Permite valores dinâmicos | ✅ (como DateTime.Now) | ❌ (somente valores fixos) | | Verificações lógicas | ✅ (if (!context.Table.Any())) | ❌ | | Rollback automático | ❌ (não faz parte das migrations) | ✅ (via migrations) | | Ideal para | Seeds complexos, condicionais, testes | Seeds simples, reprodutíveis | |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Essa construção:  using (var scope = app.Services.CreateScope())  {  var services = scope.ServiceProvider;  var context = services.GetRequiredService<AppDbContext>();  DbInitializer.Seed(context);  }  é **essencial para executar tarefas de forma segura e isolada** no ciclo de vida do Program.cs, especialmente quando estamos lidando com **injeção de dependência (DI)** no ASP.NET Core.  **🧠 Explicação detalhada: linha por linha**  **🔹 Linha 1: using (var scope = app.Services.CreateScope())**  **🔧 O que faz:**   * Cria **um escopo de serviços temporário** dentro do sistema de **injeção de dependências** (DI) do ASP.NET Core. * Isso é necessário porque o container principal (app.Services) fornece objetos com ciclo de vida controlado, e DbContext normalmente é registrado como Scoped, ou seja, ele **só deve existir dentro de um escopo controlado**.   **📌 Por que usar using aqui?**   * Esse using é diferente daquele usado no topo do arquivo (using Namespace). * Aqui, é o **using statement** (declaração de escopo), que serve para garantir que os **recursos sejam liberados corretamente** (via IDisposable) após o uso. * Ele se comporta como um bloco try-finally que chama Dispose() no final:   csharp  CopiarEditar  var scope = app.Services.CreateScope();  try {  // usa scope  }  finally {  scope.Dispose();  }  **✅ É necessário para garantir que:**   * O DbContext seja **corretamente instanciado e descartado** após a execução do seed. * Nenhum recurso (como conexões com o banco) fique **vazando ou aberto indevidamente**.   **🔹 Linha 2: var services = scope.ServiceProvider;**  **🔧 O que faz:**   * Obtém o resolvedor de dependências (IServiceProvider) **dentro daquele escopo**. * Isso permite resolver serviços como AppDbContext, ILogger, UserManager, etc., **dentro de um contexto isolado**.   **🔹 Linha 3: var context = services.GetRequiredService<AppDbContext>();**  **🔧 O que faz:**   * Resolve (instancia) o AppDbContext registrado no DI. * O método GetRequiredService<T>():   + Lança erro se o serviço não foi registrado (diferente de GetService<T>() que retorna null).   + Garante que você **sempre receba uma instância válida**.   **🧵 Isso é equivalente a:**  csharp  CopiarEditar  AppDbContext context = scope.ServiceProvider.GetRequiredService<AppDbContext>();  **🔹 Linha 4: DbInitializer.Seed(context);**  **🔧 O que faz:**   * Chama o método estático que você criou (DbInitializer.Seed) e passa o contexto como argumento. * Dentro desse método é onde você usa AddRange() e SaveChanges() para popular o banco.   **📘 Comparativo entre os dois tipos de using**   | **Tipo de using** | **Exemplo** | **Finalidade** | | --- | --- | --- | | **Using declaração de namespace** | using Microsoft.EntityFrameworkCore; | Torna os tipos disponíveis no arquivo | | **Using statement (bloco de escopo)** | using (var scope = app.Services.CreateScope()) | Gerencia ciclo de vida de objetos com Dispose() |   **💡 Situação prática onde isso é usado**  Ao rodar a aplicação, você precisa popular o banco de dados com alguns dados iniciais. Como AppDbContext é um serviço Scoped, **não pode ser instanciado diretamente no Program.cs fora de um escopo controlado**, por isso:   1. Criamos um escopo com CreateScope(); 2. Resolvemos AppDbContext; 3. Executamos a lógica de seed; 4. Garantimos a liberação dos recursos.   **✅ Ilustração da estrutura**  csharp  CopiarEditar  Program.cs  ├── Cria o `builder`  ├── Cria o `app`  └── using (var scope = app.Services.CreateScope())  ├── Obtém o AppDbContext  └── Executa o método Seed() |

* + Cria uma migration vazia usando o Migrations e usar o métodos Up() e Down() definindo nestes métodos o insert into para incluir dados nas tabela

|  |
| --- |
| Vamos criar **uma migration vazia** com o comando:  bash  CopiarEditar  dotnet ef migrations add SeedData  Essa migration será usada exclusivamente para **inserir dados nas tabelas Categorias e Produtos**, utilizando os métodos Up() e Down() para controlar o versionamento (migrar/adicionar e reverter/deletar os dados inseridos).  **✅ Exemplo completo da Migration SeedData.cs**  csharp  CopiarEditar  using Microsoft.EntityFrameworkCore.Migrations;  using System;  #nullable disable  namespace ApiCatalogo.Migrations  {  public partial class SeedData : Migration  {  protected override void Up(MigrationBuilder migrationBuilder)  {  // Inserindo Categorias  migrationBuilder.Sql(@"  INSERT INTO Categorias (CategoriaId, Nome, ImagemUrl)  VALUES  (1, 'Bebidas', 'bebidas.jpg'),  (2, 'Lanches', 'lanches.jpg'),  (3, 'Sobremesas', 'sobremesas.jpg');  ");  // Inserindo Produtos (relacionados às categorias pelo ID)  migrationBuilder.Sql(@"  INSERT INTO Produtos (ProdutoId, Nome, Descricao, Preco, ImagemUrl, Estoque, DataCadastro, CategoriaId)  VALUES  (1, 'Coca-Cola', 'Refrigerante de Cola 2L', 7.50, 'coca.jpg', 100, GETDATE(), 1),  (2, 'X-Burguer', 'Pão, carne e queijo', 15.00, 'xburguer.jpg', 50, GETDATE(), 2),  (3, 'Sorvete Napolitano', 'Sorvete 1L sabor napolitano', 20.00, 'sorvete.jpg', 30, GETDATE(), 3);  ");  }  protected override void Down(MigrationBuilder migrationBuilder)  {  // Remover os dados inseridos (rollback)  migrationBuilder.Sql("DELETE FROM Produtos WHERE ProdutoId IN (1, 2, 3);");  migrationBuilder.Sql("DELETE FROM Categorias WHERE CategoriaId IN (1, 2, 3);");  }  }  }  **📌 Localização e propósito**   * Essa classe é gerada automaticamente dentro da pasta Migrations. * O conteúdo é editado **manualmente**, como no exemplo acima. * **Up()**: serve para aplicar a inserção dos dados. * **Down()**: desfaz os inserts (útil para rollback).   **⚠️ Importante**   * Certifique-se de que o banco já esteja com as tabelas criadas antes de aplicar esta migration. * A sequência de inserção respeita a integridade relacional: primeiro Categorias, depois Produtos. * Use GETDATE() no SQL Server para registrar a data atual.   **✅ Executar a migration no banco**  dotnet ef database update |

**Criando os Controllers para uso na WebApi**

Os controllers são arquivos aramazenados dentro da pasta controllers criada na raiz de uma aplicação Asp.Net Core

Estes controller vao ser filhas da Classe ControllerBase

O nome de um controller é formado pelo nome do Controller (no Plural) + a palavra Controller: ProdutosController, CategoriasController...

Nesta forma de trabalho o controller deve ser representado desta forma

[Route(“[controller]”)]

[ApiController]

Public class TestesController : ControllerBase

{

//métodos Action

}

O Atributo [Route] especifica um apdrão de URL para acessar umcontroller ou action

**a DataAnnottation [ApiController] fornece**

- requisitos de roteamento de atributo

Respostas Http 400 automáticas (validação do ModelState)

Inferencia de patametros de associação

Inferencia de solicitação de dados de várias Partes/Formulário

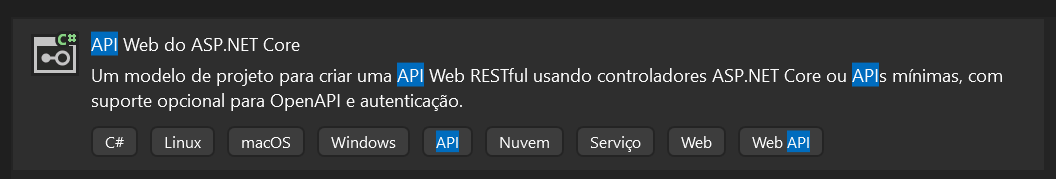
Uso de Problem/Detail para códigos de status de erros

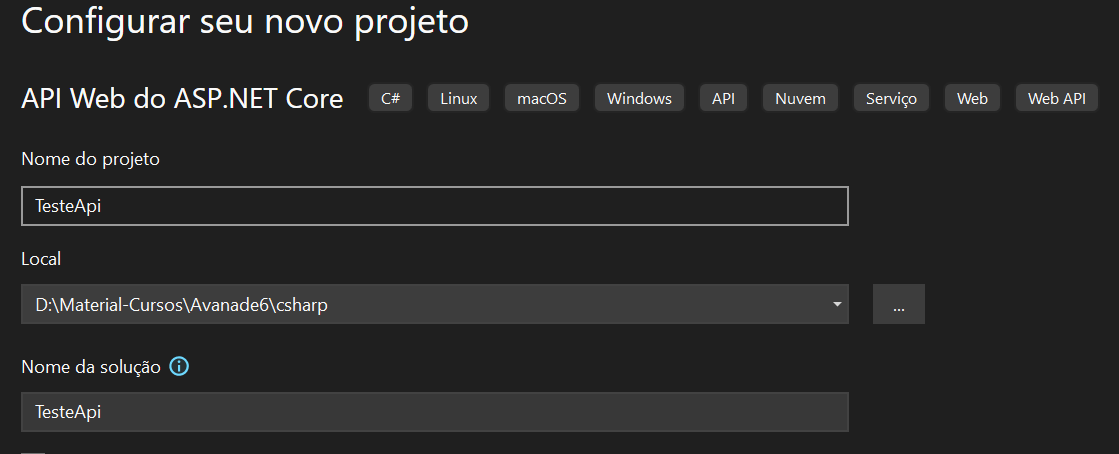
A classe base (ControllerBase) que está sendo referenciada por Herança fornece muitas propriedades e métodos que são úteis para lidar com requisilçoes Htpp

Isto só é possível se e4xisitr no Program.cs o builder.Services. AddControlles() estiver na injeção de serviços do container e o app.MapControllers().

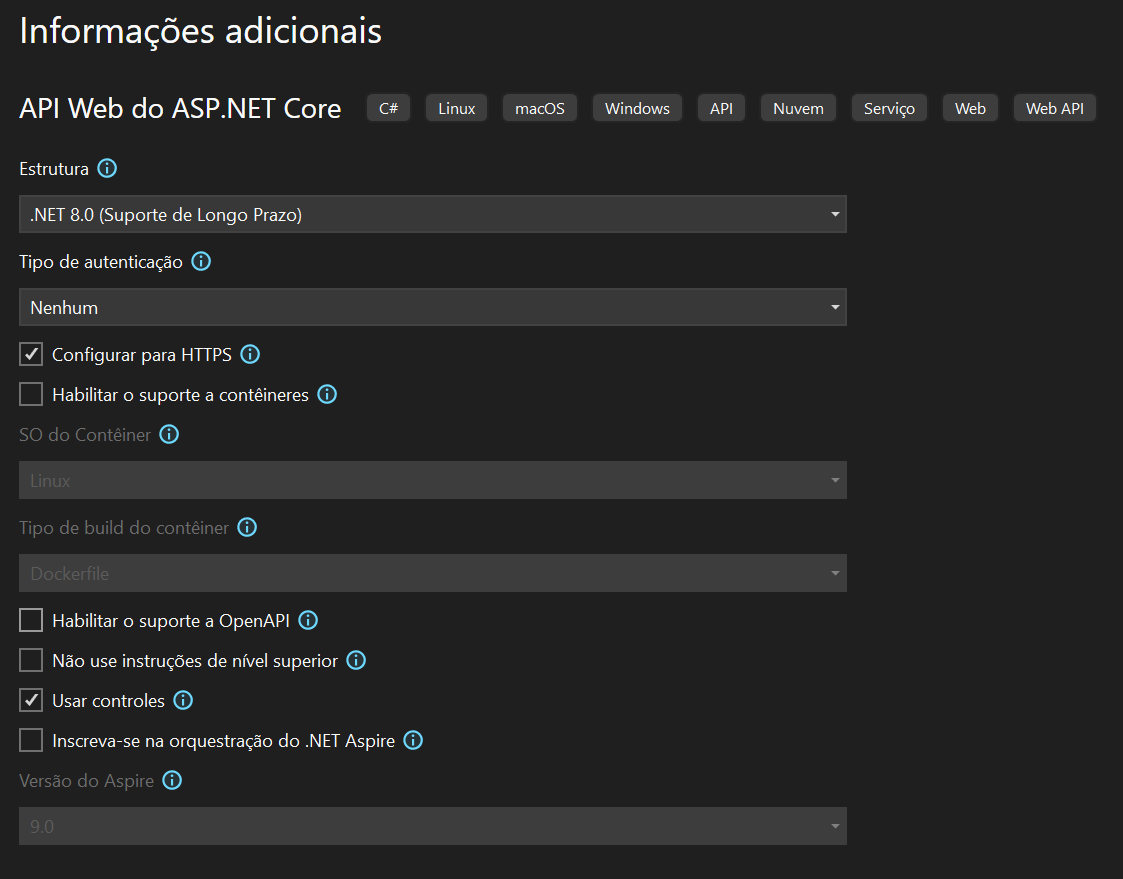
Prática

Crie um novo projeto





No próximo passo desmarque o usar o a OPENAPi



Clique em criar

Abr ao controller

Acrescente o segunte método

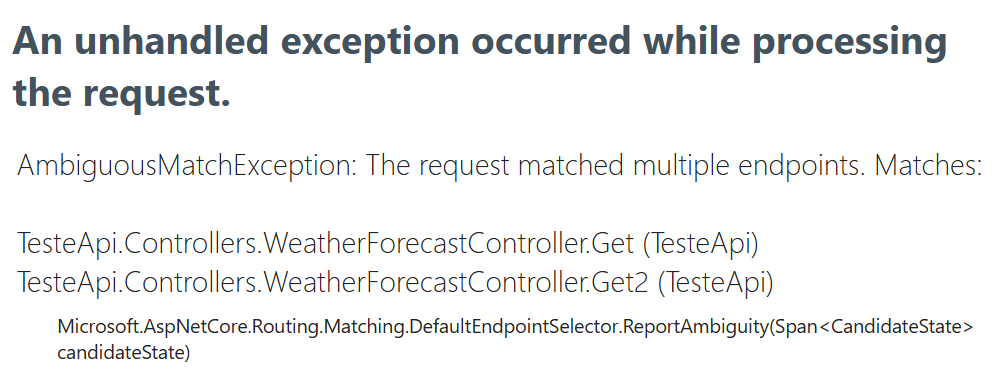
|  |
| --- |
| public string Get2()  {  return "Hello World!";  } |

Ao executar perceba que ele não á acessado

Preciso indicar a qual verbo Http ele responde. Acrescente o [HttpGet]

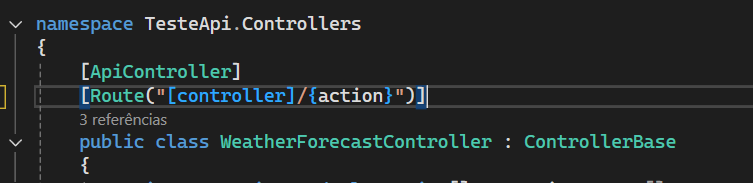
|  |
| --- |
| [HttpGet]  public string Get2()  {  return "Hello World!";  } |

Ao exeutar

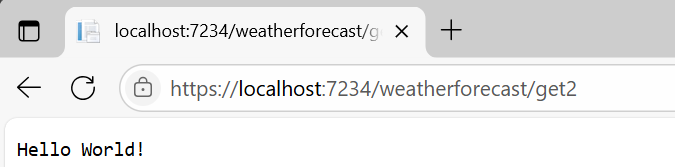


Precisamos corrigir na chamada para que el enteden que ele pode interpretar cada requisição Http como única com o valor passado.

Basta acrescentar ao DataAnnottation



Ao executar



**Criando o controlador ProdutosController**

- Criar métodos actions no controller usando recursos das WebApis

-Definir os recursos, os endpoints e o mapeamento para gerenciar informações de produtos CRUD

Usaremos a seguinte escrita para Produtos

|  |  |  |  |
| --- | --- | --- | --- |
| **Verbos Http** | **EndPoints** | **StatusCode** | **Descria~]ap** |
| POST | /produtos | 201, Created | Cria um Produto |
| GET | /produtos/{id:int} | 200, OK, 404 Not FOund | Obtem um produto pelo id |
| GET | /produtos | 200 OK | Obtem todos os produtos |
| PUT | /produtos/{id:int} | 200, OK, 404 Not FOund 400 Bad request | Atualiza um produto pelo id |
| DELETE | produtos/{id:int} | 204 No Content 200 ok | Deleta produto pelo Id |

Crie o Controllador ProdutosController na pasta Controllers

|  |
| --- |
| using ApiCatalogo.Data;  using Microsoft.AspNetCore.Http;  using Microsoft.AspNetCore.Mvc;  namespace ApiCatalogo.Controllers  {  [Route("[controller]")]  [ApiController]  public class ProdutosController : ControllerBase  {  //INJETAR A INSTÂNCIA DO CONTEXTO DO BANCO DE DADOS  private readonly AppDbContext \_context;  public ProdutosController(AppDbContext context)  {  \_context = context;  }  }  } |

**Produtos Controller GET**

|  |
| --- |
| [HttpGet]  public IEnumerable<Produto> Get()  {  //BUSCAR TODOS OS PRODUTOS  var produtos = \_context.Produtos.ToList();  if (produtos is null)  {  return NotFound();  }  return (produtos);  } |

Observe o retorno que é uma lista de produtos não pode me retonar um NOT FOUND() vamos acrescentar o ACTION RESULT que este sim aceita o tipo de retorno Not Found().

|  |
| --- |
| [HttpGet]  public ActionResult<IEnumerable<Produto>> Get()  {  //BUSCAR TODOS OS PRODUTOS  var produtos = \_context.Produtos.ToList();  if (produtos is null)  {  return NotFound();  }  return (produtos);  } |

PressioneF12 sobre o nome de cada método ou Action para visualizar sua dependência

O Not Found() 🡪 retornar um not found result 🡪 que retorna um StatusCoderesult 🡪 que é ActionResult

Execute e chame o Swagger

**ProdutosController – Get por Id**

|  |
| --- |
| [HttpGet("{id:int}")]  public ActionResult<Produto> Get(int id) {  //usar o first ele retorna o primeiro elemento que satisfaz a condição ou Lança InvalidOperationException  //usar o fristordefault ele retorna oRetorna null (referência) ou default(T) padrão dele Produto  var produto = \_context.Produtos.FirstOrDefault(p => p.ProdutoId == id);  if (produto is null)  {  return NotFound("Produto não encontrado");  }  return (produto);  } |

## ✅ Estrutura da expressão

Essa é uma **expressão lambda** — uma forma concisa de escrever **funções anônimas** (sem nome), muito usada em C# com LINQ (Language Integrated Query).

| **Parte** | **Significado** |
| --- | --- |
| p | Representa **um item individual** da coleção (ex: Produto) |
| => | **Operador lambda**, separa o parâmetro da expressão |
| p.ProdutoId == id | **Condição**: compara a propriedade ProdutoId com a variável id |

## 🧠 Interpretação por partes

## 📘 Exemplo em contexto completo

Imagine a seguinte consulta com Entity Framework:

var produto = context.Produtos.FirstOrDefault(p => p.ProdutoId == id);

### O que ela faz?

* Acessa a **tabela Produtos** (representada por DbSet<Produto>).
* Para cada item da tabela (cada Produto), representa ele com o **apelido p**.
* Avalia se o ProdutoId daquele item (p.ProdutoId) é igual ao valor da variável id.
* Retorna o **primeiro produto encontrado com esse ID**, ou null se não houver.

## 🔁 Equivalência com um foreach

A expressão lambda acima é equivalente a isso:

Produto produtoEncontrado = null;

foreach (var p in context.Produtos)

{

if (p.ProdutoId == id)

{

produtoEncontrado = p;

break;

}

}

**ProdutosController – POST**

|  |
| --- |
| [HttpPost]  public ActionResult<Produto> Post(Produto produto)  {  if (produto is null) return BadRequest();  \_context.Produtos.Add(produto);  //SALVAR AS ALTERAÇÕES NO BANCO DE DADOS  \_context.SaveChanges();  return new CreatedAtRouteResult("ObterProduto", new { id = produto.ProdutoId }, produto);  //return StatusCode(201);  } |

## ✅ Linha 1: \_context.Produtos.Add(produto);

### ✔ O que faz:

* Adiciona o objeto produto ao **rastreamento de alterações** do EF Core no contexto atual (\_context).
* Esse objeto será marcado como **"a ser inserido"** no banco quando SaveChanges() for chamado.

### 🔍 Em termos práticos:

* EF Core ainda **não envia nenhum comando SQL para o banco nessa linha**.
* Ele apenas registra internamente que você quer **inserir** o objeto produto na tabela Produtos.

### 🧠 Equivalente a:

-- Ainda não executado

INSERT INTO Produtos (...) VALUES (...)

## ✅ Linha 2: \_context.SaveChanges();

### ✔ O que faz:

* **Envia ao banco** todas as operações pendentes (como Add, Update, Remove) que foram acumuladas no contexto.
* Neste caso, envia a instrução **INSERT INTO Produtos**, persistindo o novo produto no banco de dados.
* Depois de persistir, o EF atualiza o objeto produto com a **chave primária gerada pelo banco** (produto.ProdutoId), se ela for IDENTITY.

### 🧠 Importante:

* Se o campo ProdutoId for auto-incremento ([Key] com banco SQL Server IDENTITY(1,1)), essa linha irá preencher automaticamente a propriedade produto.ProdutoId.

## ✅ Linha 3:

return new CreatedAtRouteResult("ObterProduto", new { id = produto.ProdutoId }, produto);

IMPORTANTE, caso vc queira que ele retorne ao cadastrar e liste o produto inserido acrescente no método GET por id o name da rota OBTERProduto

[HttpGet("{id:int}", Name="ObterProduto")]

public ActionResult<Produto> Get(int id) {

### ✔ O que faz:

* Retorna um **HTTP 201 (Created)** para o cliente da API, com os dados do recurso recém-criado.
* É uma boa prática RESTful.

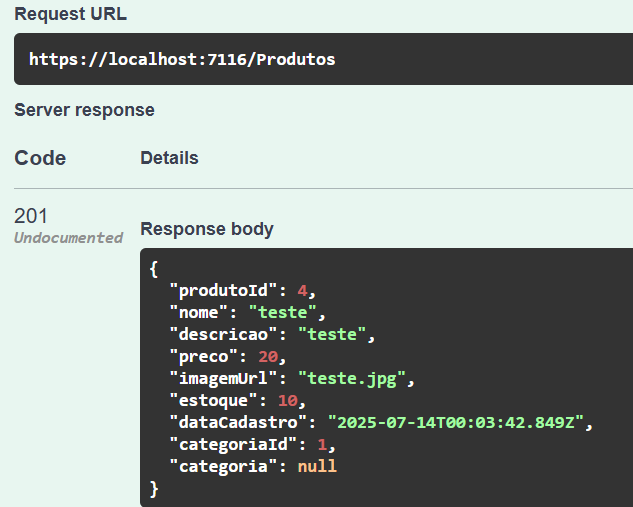
### 📘 Desmembrando:

| **Parte** | **Explicação** |
| --- | --- |
| CreatedAtRouteResult(...) | Cria a resposta HTTP 201 com **Location header** apontando para o novo recurso. |
| "ObterProduto" | Nome da rota definida na controller (ex: [HttpGet("{id}", Name = "ObterProduto")]) |
| new { id = produto.ProdutoId } | Parâmetros que preenchem a rota (gera a URL final, ex: /api/produtos/5) |
| produto | Corpo da resposta JSON com os dados recém-cadastrados |

Para testar rode o Swagger e no POST insira

|  |
| --- |
| {  "nome": "teste",  "descricao": "teste",  "preco": 20,  "imagemUrl": "teste.jpg",  "estoque":10,  "dataCadastro": "2025-07-14T00:03:42.849Z",  "categoriaId": 1    } |

O resultado deve ser



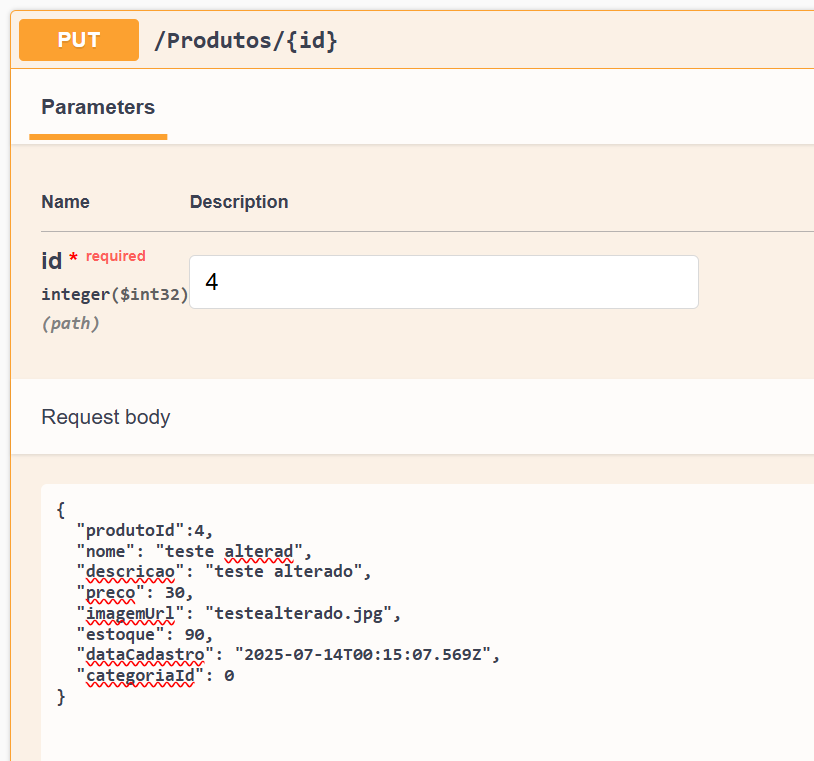
IMPORTANTE

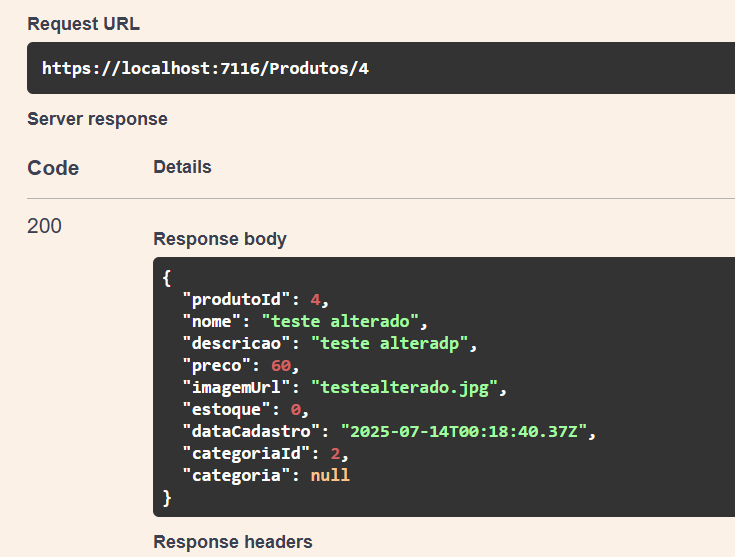
OBSERVE SE NA MODEL PRODUTO O CAMPO DE NAVEGAÇÃO ESTÁ COM A “?”

public Categoria? Categoria { get; set; }

**ProdutosController – PUT**

|  |
| --- |
| [HttpPut("{id:int}")]  public ActionResult Put(int id, Produto produto) {  if (id != produto.ProdutoId) {  return BadRequest();  }  \_context.Entry(produto).State = EntityState.Modified;  \_context.SaveChanges();  return Ok(produto);  } |

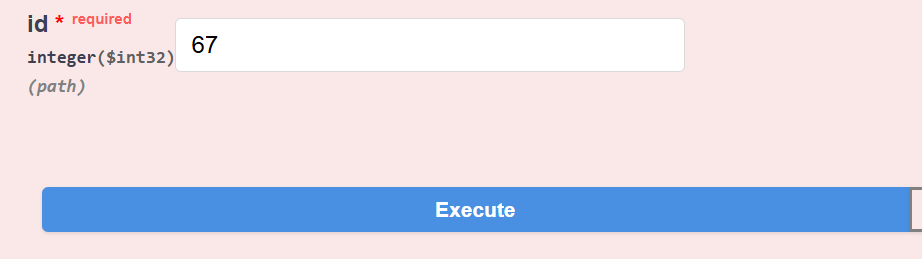


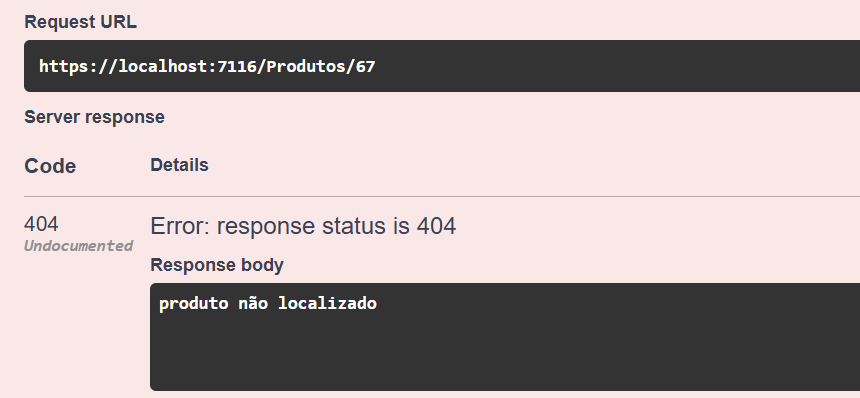




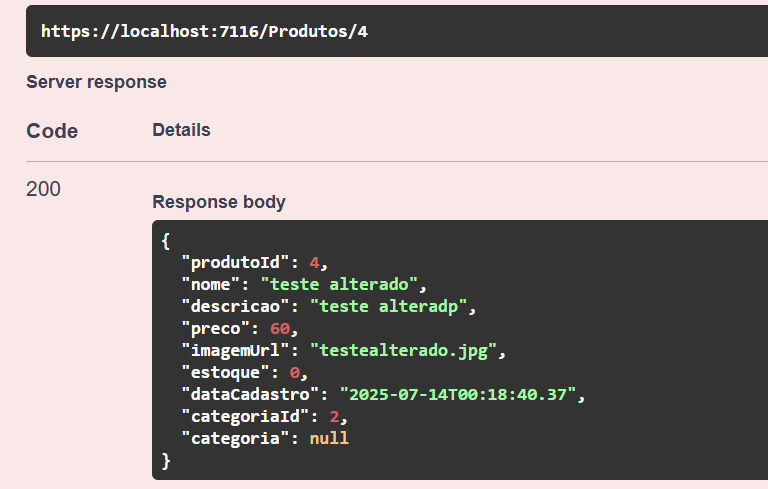
**ProdutosController – Delete**

|  |
| --- |
| [HttpDelete("{id:int}")]  public ActionResult Delete(int id)  {  var produto = \_context.Produtos.FirstOrDefault(p => p.ProdutoId == id);  if (produto is null)  {  return NotFound("produto não localizado");  }  \_context.Produtos.Remove(produto);  \_context.SaveChanges();  return Ok(produto);  } |









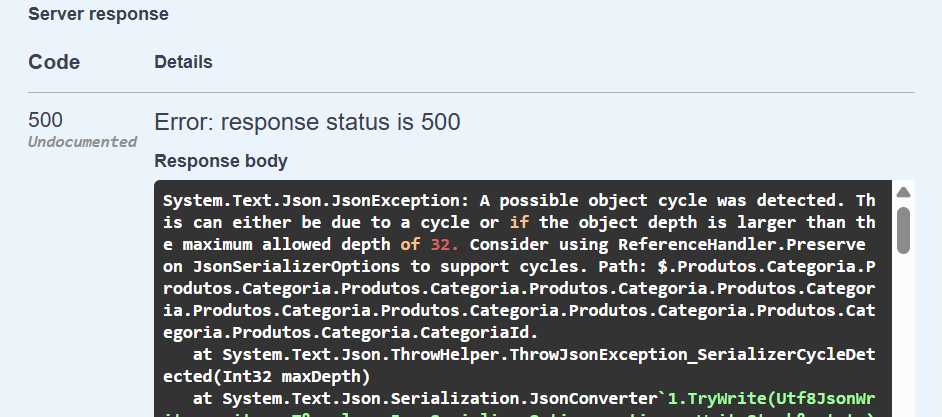
CategoriasController- Código Completo

|  |
| --- |
| using ApiCatalogo.Data;  using ApiCatalogo.Models;  using Microsoft.AspNetCore.Mvc;  using Microsoft.EntityFrameworkCore;  namespace APICatalogo.Controllers;  [Route("[controller]")]  [ApiController]  public class CategoriasController : ControllerBase  {  private readonly AppDbContext \_context;  public CategoriasController(AppDbContext context)  {  \_context = context;  }  [HttpGet]  public ActionResult<IEnumerable<Categoria>> Get()  {  return \_context.Categorias.ToList();  }  [HttpGet("{id:int}", Name = "ObterCategoria")]  public ActionResult<Categoria> Get(int id)  {  var categoria = \_context.Categorias.FirstOrDefault(p => p.CategoriaId == id);  if (categoria == null)  {  return NotFound("Categoria não encontrada...");  }  return Ok(categoria);  }  [HttpPost]  public ActionResult Post(Categoria categoria)  {  if (categoria is null)  return BadRequest();  \_context.Categorias.Add(categoria);  \_context.SaveChanges();  return new CreatedAtRouteResult("ObterCategoria",  new { id = categoria.CategoriaId }, categoria);  }  [HttpPut("{id:int}")]  public ActionResult Put(int id, Categoria categoria)  {  if (id != categoria.CategoriaId)  {  return BadRequest();  }  \_context.Entry(categoria).State = EntityState.Modified;  \_context.SaveChanges();  return Ok(categoria);  }  [HttpDelete("{id:int}")]  public ActionResult Delete(int id)  {  var categoria = \_context.Categorias.FirstOrDefault(p => p.CategoriaId == id);  if (categoria == null)  {  return NotFound("Categoria não encontrada...");  }  \_context.Categorias.Remove(categoria);  \_context.SaveChanges();  return Ok(categoria);  }  } |

Se que quiser listas duas entidades

|  |
| --- |
| [HttpGet("produtos")]  public ActionResult<IEnumerable<Categoria>> GetCategoriasProdutos() {  return \_context.Categorias  .Include(p => p.Produtos)  .ToList();  } |

Erro



**Ajustes e Otimizações**

Json Serialização e Desserialização

NewtonSoft.json (antiga)

A biblioteca padrão é System.text.Json (referencia Ciclica)

Como corrigir

Ignorar o ciclo de referencia no AddControllers()

builder.Services.AddControllers().AddJsonOptions(options=>

options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles);

|  |
| --- |
| 🔁 Problema: Referências circulares (ciclos) Imagine o seguinte modelo com navegação entre Categoria e Produto:  public class Categoria  {  public int Id { get; set; }  public string Nome { get; set; }  public ICollection<Produto> Produtos { get; set; }  }  public class Produto  {  public int Id { get; set; }  public string Nome { get; set; }  public Categoria Categoria { get; set; } // <-- referência de volta  }  Se você retornar Categoria, o JSON tenta serializar também os Produtos, e cada Produto referencia a Categoria novamente, criando **um loop infinito**:  {  "id": 1,  "nome": "Informática",  "produtos": [  {  "id": 101,  "nome": "Notebook",  "categoria": {  "id": 1,  "nome": "Informática",  "produtos": [ ...loop... ]  }  }  ]  }  Isso resulta em erro:  ❌ System.Text.Json.JsonException: A possible object cycle was detected ✅ Solução com ReferenceHandler.IgnoreCycles Quando você configura isso:  builder.Services.AddControllers().AddJsonOptions(options =>  options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles); 🛠 O que acontece?  * O serializador **detecta ciclos** e **ignora** reserializar os objetos já processados. * Assim, evita a exceção e mantém a estrutura JSON válida.  🔄 Exemplo do JSON gerado após usar IgnoreCycles: json  CopiarEditar  {  "id": 1,  "nome": "Informática",  "produtos": [  {  "id": 101,  "nome": "Notebook",  "categoria": null // ou omitido, dependendo da config  }  ]  } ☑️ Considerações  * ✅ Ideal para APIs com EF Core e relacionamentos complexos. * ⚠️ Perde-se um pouco da **navegação completa** nos objetos serializados (evita ciclo, mas corta a profundidade). * 🔁 Alternativas: usar DTOs ou AutoMapper para **controlar o que será retornado**. |

Acrescente tb em Produtto.cs (model)

|  |
| --- |
| [JsonIgnore(Condition =JsonIgnoreCondition.Always)]  public Categoria? Categoria { get; set; } |

Isto vai facilitar na criação dos dados e manipulação sem a necessidade de declaração

Melhoria nos gets

[HttpGet]

public ActionResult<IEnumerable<Categoria>> Get()

{

return \_context.Categorias.AsNoTracking().ToList();

}

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| O uso de AsNoTracking() é uma **otimização importante no Entity Framework Core**, especialmente em operações de **leitura** (consultas GET). 🔍 O que é AsNoTracking()? Por padrão, quando você faz uma consulta como:  \_context.Categorias.ToList();  ➡️ O Entity Framework **"rastreia"** os objetos retornados. Ou seja, ele armazena internamente os dados originais para poder detectar alterações futuras (para um possível Update ou Remove). 🧠 AsNoTracking() desabilita esse rastreamento \_context.Categorias.AsNoTracking().ToList(); ✔ Benefícios:  | **Situação** | **Vantagem com AsNoTracking()** | | --- | --- | | Leitura (GET) | Mais **rápido**, menos uso de **memória** | | Sem intenção de atualizar | Evita rastreamento desnecessário | | Dados só para exibir (readonly) | Ideal para APIs REST |   🚀 Ele melhora o desempenho, principalmente em **grandes volumes de dados**. 📊 Comparação prática🔁 Sem AsNoTracking():  * O EF Core carrega os objetos e os **mantém no ChangeTracker**. * Pode impactar a **performance** em APIs de alta carga.  🧘 Com AsNoTracking():  * Objetos são carregados **sem serem "monitorados"**. * Ideal para **APIs de consulta** (como no seu Get()).  🧪 Exemplo de impacto:Sem AsNoTracking(): var categorias = \_context.Categorias.ToList();  // EF rastreia cada Categoria → ocupa memória Com AsNoTracking(): var categorias = \_context.Categorias.AsNoTracking().ToList();  // EF só lê e descarta logo após ⚠ Quando **não usar** AsNoTracking():  * Quando for realizar operações de **edição**, como:   + context.Update(obj)   + context.Remove(obj) * Pois o EF precisa rastrear o objeto para detectar o que mudou.  ✅ Conclusão resumida:  | **Use AsNoTracking() quando...** | **Evite quando...** | | --- | --- | | Está **lendo dados apenas para exibição** | Vai **modificar e salvar** os dados | | Precisa de **desempenho em leitura** | Precisa de **rastreio de alterações** | |

**Revisão técnica do controller CategoriasController** com foco em:

1. **Boas práticas de desempenho**
2. **Clareza no código**
3. **Aderência à arquitetura REST**
4. **Segurança e validações**
5. **Preparação para produção**

**Não Precisa aplicar ainda ok**

## 🔍 REVISÃO GERAL E MELHORIAS SUGERIDAS

### ✅ 1. ****Evitar rastreamento quando não necessário (****AsNoTracking****)****

No GET e GET por ID, **não há necessidade de rastrear entidades** (tracking do EF Core), o que consome mais memória e CPU.

**Antes:**

return \_context.Categorias.ToList();

**Depois:**

csharp

CopiarEditar

return \_context.Categorias.AsNoTracking().ToList();

Mesma lógica aplica-se para:

var categoria = \_context.Categorias.AsNoTracking().FirstOrDefault(p => p.CategoriaId == id);

### ✅ 2. ****Evitar retorno de entidades completas com relacionamentos diretos****

Evitar retornar Categoria com a lista completa de Produtos diretamente, pois pode causar problemas de desempenho e serialização circular.

**Sugestão:**  
Criar uma **DTO (Data Transfer Object)** para a resposta de categorias com produtos, retornando apenas os campos necessários.

### ✅ 3. ****Adicionar validação de modelo (****ModelState****) no**** POST ****e**** PUT

**Antes:**

if (categoria is null)

return BadRequest();

**Melhor:**

csharp

CopiarEditar

if (!ModelState.IsValid)

return BadRequest(ModelState);

E use [ApiController] corretamente para deixar o framework validar automaticamente os modelos com data annotations.

### ✅ 4. ****Evitar duplicação de código no**** DELETE ****e**** GET ****por ID****

Tanto o GET quanto o DELETE usam o mesmo FirstOrDefault(...). Extraia para um método privado para manter o DRY (Don't Repeat Yourself):

private Categoria? BuscarCategoria(int id) =>

\_context.Categorias.AsNoTracking().FirstOrDefault(p => p.CategoriaId == id);

### ✅ 5. ****Adicionar retorno HTTP adequado para**** POST ****e**** PUT

O PUT deve retornar 204 No Content se não houver necessidade de retornar o recurso atualizado:

return NoContent(); // mais RESTful

### ✅ 6. ****Adicionar tratamento de exceções no acesso ao banco (try/catch)****

Exemplo de tratamento genérico para falhas inesperadas:

try

{

\_context.SaveChanges();

}

catch (DbUpdateException ex)

{

return StatusCode(500, "Erro ao salvar os dados no banco.");

}

### ✅ 7. ****Boa prática: Nomeie o endpoint com plural no**** [Route]

**Antes:**

[Route("[controller]")]

**Melhor:**

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

## 🧪 CÓDIGO REVISADO COM MELHORIAS

csharp

CopiarEditar

using ApiCatalogo.Data;

using ApiCatalogo.Models;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

namespace ApiCatalogo.Controllers;

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

[ApiController]

public class CategoriasController : ControllerBase

{

private readonly AppDbContext \_context;

public CategoriasController(AppDbContext context)

{

\_context = context;

}

[HttpGet("produtos")]

public ActionResult<IEnumerable<Categoria>> GetCategoriasProdutos()

{

return \_context.Categorias

.Include(p => p.Produtos)

.AsNoTracking()

.ToList();

}

[HttpGet]

public ActionResult<IEnumerable<Categoria>> Get()

{

return \_context.Categorias.AsNoTracking().ToList();

}

[HttpGet("{id:int}", Name = "ObterCategoria")]

public ActionResult<Categoria> Get(int id)

{

var categoria = BuscarCategoria(id);

if (categoria == null)

{

return NotFound("Categoria não encontrada...");

}

return Ok(categoria);

}

[HttpPost]

public ActionResult Post([FromBody] Categoria categoria)

{

if (!ModelState.IsValid)

return BadRequest(ModelState);

\_context.Categorias.Add(categoria);

\_context.SaveChanges();

return CreatedAtRoute("ObterCategoria", new { id = categoria.CategoriaId }, categoria);

}

[HttpPut("{id:int}")]

public ActionResult Put(int id, [FromBody] Categoria categoria)

{

if (id != categoria.CategoriaId)

return BadRequest("Id da categoria não confere.");

if (!ModelState.IsValid)

return BadRequest(ModelState);

\_context.Entry(categoria).State = EntityState.Modified;

\_context.SaveChanges();

return NoContent();

}

[HttpDelete("{id:int}")]

public ActionResult Delete(int id)

{

var categoria = \_context.Categorias.FirstOrDefault(p => p.CategoriaId == id);

if (categoria == null)

return NotFound("Categoria não encontrada...");

\_context.Categorias.Remove(categoria);

\_context.SaveChanges();

return NoContent();

}

private Categoria? BuscarCategoria(int id)

{

return \_context.Categorias

.AsNoTracking()

.FirstOrDefault(p => p.CategoriaId == id);

}

}

## 📌 Considerações Finais

Essas melhorias tornam sua Web API:

* Mais eficiente (menos memória, menos processamento)
* Mais segura e robusta
* Alinhada com boas práticas REST
* Preparada para crescimento e produção

Tratamentos de erros

Considere o Ambiente de Desenvolvimento e Produção

Desenvolvimento

* Usa por padrão a página de excessão do desenvolvedor que exibe informações detalhadas sobre as exceções como:
* Stack trace (rastreamento de pilha)
* Parâmetros de cadeias de caracteres de consulta
* Coockies
* Headers (cabeçalhos)

Vamos simular um erro

[HttpGet]

public ActionResult<IEnumerable<Categoria>> Get()

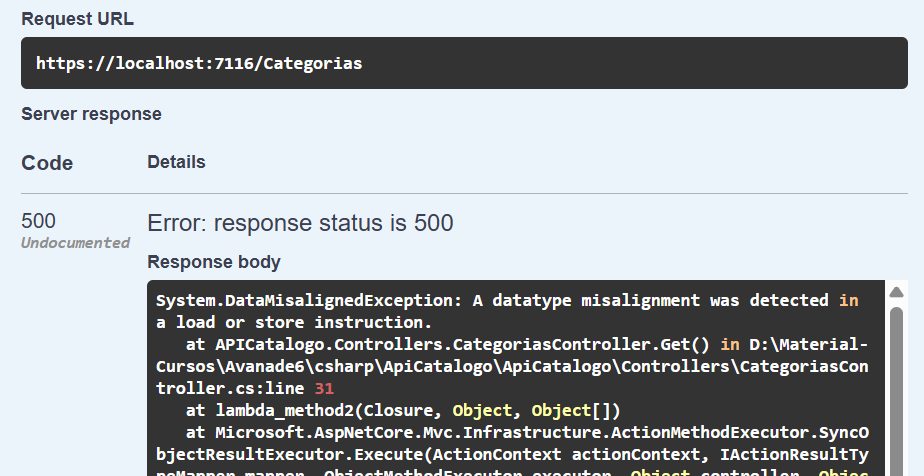
{

throw new DataMisalignedException();

// return \_context.Categorias.ToList();

}

Ao executar e chamar o método get de categoria



Como tratamos

|  |
| --- |
| [HttpGet]  public ActionResult<IEnumerable<Categoria>> Get()  {  try  {  throw new DataMisalignedException();  //return \_context.Categorias.ToList()  }  catch (Exception)  {  return StatusCode(StatusCodes.Status500InternalServerError,  "Erro ao tentar recuperar os dados do banco de dados...");  }    } |

Erxecute novamente



Pode fazer o mesmo no método get por id

|  |
| --- |
| [HttpGet("{id:int}", Name = "ObterCategoria")]  public ActionResult<Categoria> Get(int id)  {  try  {  var categoria = \_context.Categorias.FirstOrDefault(p => p.CategoriaId == id);  if (categoria == null)  {  return NotFound("Categoria não encontrada...");  }  return Ok(categoria);  }  catch (Exception)  {  return StatusCode(StatusCodes.Status500InternalServerError,  "Erro ao tentar recuperar os dados do banco de dados...");  }    } |