**Muitos-para-Muitos no asp.net Core**

**Estrutura do Projeto**

Primeiro, vamos visualizar a estrutura de arquivos e pastas que iremos criar. Esta é uma organização padrão e recomendada para projetos ASP.NET Core Web API.

HotelApi/

├── Controllers/

│ ├── ComodidadesController.cs

│ └── HoteisController.cs

├── Data/

│ └── AppDbContext.cs

├── Dtos/

│ └── HotelDto.cs

├── Models/

│ ├── Comodidade.cs

│ └── Hotel.cs

├── appsettings.json

├── HotelApi.csproj

└── Program.cs

**Passo 1: Criar o Projeto e Instalar Pacotes**

Se estiver começando do zero, crie o projeto e adicione os pacotes necessários.

1. **Crie um novo projeto WEB API - HotelApi (o template padrão já usa controladores):**

*Obs: Se o template criar um WeatherForecastController.cs, pode apagá-lo.*

1. **Instale os pacotes do Entity Framework Core:**

Microsoft.EntityFrameworkCore.InMemory

Microsoft.EntityFrameworkCore.Design

**Passo 2: Definir Entidades e DTOs (Pastas Models e Dtos)**

Crie as pastas Models e Dtos.

**1. Arquivo Models/Hotel.cs** *(Esta classe representa a tabela do banco de dados)*

C#

namespace HotelApi.Models;

public class Hotel

{

public int Id { get; set; }

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

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

// Propriedade de navegação para a coleção de Comodidades

public ICollection<Comodidade> Comodidades { get; set; } = new List<Comodidade>();

}

**2. Arquivo Models/Comodidade.cs**

C#

using System.Text.Json.Serialization;

namespace HotelApi.Models;

public class Comodidade

{

public int Id { get; set; }

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

// Usamos [JsonIgnore] para evitar um erro de referência circular

// ao serializar a resposta da API (Comodidade -> Hoteis -> Comodidade...).

// Uma abordagem ainda melhor é usar DTOs para modelar as respostas da API.

[JsonIgnore]

public ICollection<Hotel> Hoteis { get; set; } = new List<Hotel>();

}

**3. Arquivo Dtos/HotelDto.cs (Data Transfer Object)** *Criar DTOs é uma boa prática para desacoplar o modelo do banco de dados da resposta da API. Usaremos isso para retornar uma lista limpa de hotéis.*

C#

namespace HotelApi.Dtos;

// Usamos um 'record' para um DTO imutável e conciso

public record HotelDto(int Id, string Nome, string Cidade);

|  |
| --- |
| **Explicação:**  Esta é a declaração do tipo HotelDto. Vamos dividi-la em partes:   * **public**: É um modificador de acesso que indica que o HotelDto pode ser acessado por qualquer outro código no mesmo projeto ou em outros projetos que referenciem este. * **record**: Esta é uma palavra-chave introduzida no C# 9 que define um tipo de referência especial. Ao usar a sintaxe posicional (com parênteses), o compilador C# gera automaticamente:   + **Propriedades init-only:** Id, Nome e Cidade são propriedades públicas que só podem ser definidas durante a inicialização do objeto. Isso torna o objeto **imutável** – uma vez criado, seu estado não pode ser alterado.   + **Construtor:** Um construtor que aceita os três parâmetros (Id, Nome, Cidade) para inicializar o objeto.   + **Desconstrutor:** Permite extrair os valores das propriedades em variáveis separadas.   + **Igualdade baseada em valor:** Diferente das classes tradicionais, que comparam referências de memória, dois records são considerados iguais se todos os seus valores de propriedade forem iguais.   + **Método ToString() otimizado:** Uma representação em string do objeto que mostra o nome do tipo e os valores de suas propriedades, o que é útil para depuração. * **HotelDto**: É o nome do nosso tipo. O sufixo "Dto" indica claramente que se trata de um *Data Transfer Object*. * **(int Id, string Nome, string Cidade)**: Estes são os parâmetros posicionais que definem as propriedades do record. Cada um especifica um tipo de dado (int, string) e um nome de propriedade (Id, Nome, Cidade).   **Importância:**  O uso de um record para um DTO é uma prática moderna em C# com várias vantagens significativas:   * **Imutabilidade:** Objetos imutáveis são mais seguros em ambientes com múltiplas threads e ajudam a prevenir efeitos colaterais inesperados, tornando o código mais previsível e menos propenso a bugs. Um DTO, cujo único propósito é carregar dados, raramente precisa ser modificado após sua criação. * **Concisão e Legibilidade:** A sintaxe do record é extremamente enxuta. A declaração de uma linha substitui dezenas de linhas de código que seriam necessárias para criar uma classe imutável tradicional com as mesmas funcionalidades (propriedades, construtor, métodos de igualdade, etc.). * **Segurança de Dados:** Como um DTO é frequentemente usado para transferir dados entre as camadas de uma aplicação (por exemplo, do serviço para o cliente), garantir que esses dados não possam ser alterados durante o trânsito é uma boa prática de design. * **Separação de Responsabilidades (Princípio do SRP):** O HotelDto tem uma única responsabilidade: transportar dados sobre um hotel de uma forma simples e estruturada. Ele não contém lógica de negócio ou de acesso a dados, o que o desacopla das entidades de domínio do banco de dados e da lógica da interface do usuário. Isso permite que a estrutura do banco de dados e a da API evoluam de forma independente. |

**Passo 3: Configurar o DbContext (Pasta Data)**

Crie a pasta Data.

**Arquivo Data/AppDbContext.cs** *(A configuração do DbContext permanece a mesma)*

C#

using HotelApi.Models;

using Microsoft.EntityFrameworkCore;

namespace HotelApi.Data;

public class AppDbContext : DbContext

{

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

public DbSet<Hotel> Hoteis { get; set; }

public DbSet<Comodidade> Comodidades { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

// Configura a relação Muitos-para-Muitos

modelBuilder.Entity<Hotel>()

.HasMany(h => h.Comodidades)

.WithMany(c => c.Hoteis);

}

}

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vamos analisar linha por linha:  C#  protected override void OnModelCreating(ModelBuilder modelBuilder)   * **protected override void OnModelCreating(...)**: Este é um método da classe DbContext do EF Core que você pode **sobrescrever** (override) para personalizar o modelo de dados. Ele é chamado pelo EF Core uma única vez, quando o modelo para um contexto de banco de dados está sendo criado pela primeira vez na memória. É aqui que você define regras que não podem ser inferidas automaticamente pelo EF Core a partir das suas classes. * **ModelBuilder modelBuilder**: Este é o parâmetro principal do método. O ModelBuilder é uma classe poderosa que age como uma "API fluente" (Fluent API). "Fluente" significa que você pode encadear chamadas de método de forma legível para construir a configuração do seu modelo de dados. É através do modelBuilder que você informa ao EF Core sobre chaves primárias, chaves estrangeiras, nomes de tabelas, relacionamentos, etc.   C#  {  base.OnModelCreating(modelBuilder);   * **base.OnModelCreating(modelBuilder);**: Esta linha chama a implementação original (base) do método OnModelCreating. É uma boa prática mantê-la para garantir que qualquer lógica de configuração padrão do EF Core seja executada antes das suas personalizações.   C#  // Configura a relação Muitos-para-Muitos   * Este é um comentário, explicando a intenção do código que vem a seguir. Ele indica que o objetivo é configurar um relacionamento do tipo **Muitos-para-Muitos** (Many-to-Many).   C#  modelBuilder.Entity<Hotel>()  .HasMany(h => h.Comodidades)  .WithMany(c => c.Hoteis);  }  Esta é a parte central da configuração e utiliza a Fluent API para definir o relacionamento. Vamos quebrar essa cadeia de chamadas:   1. **modelBuilder.Entity<Hotel>()**: Aqui, você está dizendo ao EF Core: "Vou começar a configurar a entidade Hotel". Todo o encadeamento a seguir se aplicará a esta entidade. 2. **.HasMany(h => h.Comodidades)**:    * **HasMany** (Tem Muitos): Este método indica que a entidade Hotel tem um relacionamento de "um-para-muitos" com outra entidade.    * **h => h.Comodidades**: Esta é uma expressão lambda. Ela especifica a **propriedade de navegação** na classe Hotel que representa o lado "muitos" da relação. Ou seja, está dizendo que **um Hotel (h) tem uma coleção de Comodidades**. A classe Hotel provavelmente tem uma propriedade como public ICollection<Comodidade> Comodidades { get; set; }. 3. **.WithMany(c => c.Hoteis)**:    * **WithMany** (Com Muitos): Este método completa a configuração, definindo o outro lado do relacionamento. Ele informa que a entidade Comodidade também tem uma relação de "um-para-muitos" com a entidade Hotel.    * **c => c.Hoteis**: Similarmente, esta expressão lambda especifica a propriedade de navegação na classe Comodidade. Está dizendo que **uma Comodidade (c) pode estar presente em uma coleção de Hoteis**. A classe Comodidade deve ter uma propriedade como public ICollection<Hotel> Hoteis { get; set; }.   **Qual a Importância e o Resultado Prático?**  Ao combinar HasMany com WithMany, você está informando ao EF Core que existe um relacionamento **Muitos-para-Muitos** entre Hotel e Comodidade.   * Um hotel pode ter várias comodidades (piscina, Wi-Fi, academia). * Uma comodidade (como Wi-Fi) pode estar presente em vários hotéis.   No banco de dados relacional, um relacionamento Muitos-para-Muitos não pode ser representado diretamente. Ele requer uma terceira tabela, conhecida como **tabela de junção** (ou tabela de ligação).  Graças a este código, o EF Core automaticamente (por convenção) criará essa tabela de junção para você. Ela se pareceria com algo assim:  **Tabela: HotelComodidade**   | HotelId (Chave Estrangeira) | ComodidadeId (Chave Estrangeira) | | --- | --- | | 1 | 1 | | 1 | 2 | | 2 | 1 | | ... | ... |     Cada linha nesta tabela representa uma associação entre um hotel específico e uma comodidade específica. O EF Core gerencia essa tabela "nos bastidores", permitindo que você trabalhe diretamente com as coleções (h.Comodidades e c.Hoteis) no seu código C#, sem precisar se preocupar com a complexidade da tabela de junção |

**Passo 4: Criar os Controladores (Pasta Controllers)**

Esta é a principal mudança. Os endpoints agora são métodos dentro de classes de controlador. Crie a pasta Controllers. Caso não exista.

**1. Arquivo Controllers/HoteisController.cs**

C#

using HotelApi.Data;

using HotelApi.Models;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

namespace HotelApi.Controllers;

[ApiController]

[Route("api/[controller]")] // Rota base: /api/hoteis

public class HoteisController : ControllerBase

{

private readonly AppDbContext \_context;

public HoteisController(AppDbContext context)

{

\_context = context;

}

// GET: api/hoteis

[HttpGet]

public async Task<ActionResult<IEnumerable<Hotel>>> GetHoteis()

{

var hoteis = await \_context.Hoteis

.Include(h => h.Comodidades) // Carrega as comodidades relacionadas

.AsNoTracking()

.ToListAsync();

return Ok(hoteis);

}

// GET: api/hoteis/5

[HttpGet("{id}")]

public async Task<ActionResult<Hotel>> GetHotel(int id)

{

var hotel = await \_context.Hoteis

.Include(h => h.Comodidades)

.AsNoTracking()

.FirstOrDefaultAsync(h => h.Id == id);

if (hotel == null)

{

return NotFound("Hotel não encontrado.");

}

return Ok(hotel);

}

}

**2. Arquivo Controllers/ComodidadesController.cs**

C#

using HotelApi.Data;

using HotelApi.Dtos;

using HotelApi.Models;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

namespace HotelApi.Controllers;

[ApiController]

[Route("api/[controller]")] // Rota base: /api/comodidades

public class ComodidadesController : ControllerBase

{

private readonly AppDbContext \_context;

public ComodidadesController(AppDbContext context)

{

\_context = context;

}

// GET: api/comodidades

[HttpGet]

public async Task<ActionResult<IEnumerable<Comodidade>>> GetComodidades()

{

// Note que este endpoint não carrega os hotéis por padrão para evitar dados excessivos

return await \_context.Comodidades.AsNoTracking().ToListAsync();

}

// GET: api/comodidades/1/hoteis

[HttpGet("{id}/hoteis")]

public async Task<ActionResult<IEnumerable<HotelDto>>> GetHoteisPorComodidade(int id)

{

var comodidadeExiste = await \_context.Comodidades.AnyAsync(c => c.Id == id);

if (!comodidadeExiste)

{

return NotFound("Comodidade não encontrada.");

}

var hoteis = await \_context.Hoteis

.Where(h => h.Comodidades.Any(c => c.Id == id))

.AsNoTracking()

.Select(h => new HotelDto(h.Id, h.Nome, h.Cidade)) // Usando o DTO

.ToListAsync();

return Ok(hoteis);

}

}

|  |
| --- |
| Este código define um método de ação em um controlador da API ASP.NET Core. Sua função é retornar uma lista de hotéis que estão associados a uma comodidade específica, identificada por seu Id.  **1. Atributo de Rota: [HttpGet("{id}/hoteis")]**   * **[HttpGet]**: Este é um atributo que designa o método para responder exclusivamente a requisições HTTP do tipo GET. Requisições GET são, por padrão, idempotentes e usadas para recuperar dados de um servidor. * **("{id}/hoteis")**: Este é o template da rota que define o padrão da URL para acionar este método.   + Ele é relativo à rota base do controlador. Se a rota do controlador for api/comodidades, a URL final será api/comodidades/{id}/hoteis.   + {id} é um parâmetro de rota. O valor fornecido nesta posição na URL (por exemplo, 1 em api/comodidades/1/hoteis) será automaticamente vinculado ao parâmetro id do método.   **2. Assinatura do Método**  C#  public async Task<ActionResult<IEnumerable<HotelDto>>> GetHoteisPorComodidade(int id)   * **public**: É o modificador de acesso que torna o método publicamente acessível, permitindo que o framework de roteamento da ASP.NET Core o encontre e execute em resposta a uma requisição HTTP. * **async Task<>**: Declara o método como assíncrono. Isso permite o uso do operador await dentro do método para operações de longa duração (I/O-bound), como chamadas de banco de dados. Durante a espera, o thread do servidor é liberado para processar outras requisições, melhorando a escalabilidade e a capacidade de resposta da aplicação. * **ActionResult<IEnumerable<HotelDto>>**: Define o tipo de retorno. Esta construção oferece flexibilidade e forte tipagem.   + ActionResult: Permite retornar diferentes tipos de resultados HTTP que não são os dados em si (ex: NotFoundObjectResult para um status 404, BadRequestResult para um 400).   + IEnumerable<HotelDto>: Especifica que, em caso de sucesso, o método retornará uma coleção de objetos do tipo HotelDto. Isso informa ao framework sobre o tipo de dado esperado, o que auxilia na geração de documentação da API (como Swagger/OpenAPI). * **GetHoteisPorComodidade(int id)**: É o nome do método e seu parâmetro de entrada. O valor do parâmetro id é preenchido pelo mecanismo de model binding da ASP.NET Core a partir do parâmetro de rota {id}.   **3. Validação de Existência**  C#  var comodidadeExiste = await \_context.Comodidades.AnyAsync(c => c.Id == id);  if (!comodidadeExiste)  {  return NotFound("Comodidade não encontrada.");  }   * **await \_context.Comodidades.AnyAsync(...)**: Esta linha executa uma consulta de forma eficiente no banco de dados. O método AnyAsync do Entity Framework Core gera uma consulta SQL (como SELECT TOP 1 ... ou EXISTS) para verificar se existe ao menos um registro na tabela Comodidades que satisfaça a condição da expressão lambda (c => c.Id == id). Ele retorna um bool e é mais performático do que buscar uma entidade inteira. * **if (!comodidadeExiste)**: Uma verificação condicional. Se a consulta anterior retornar false, o bloco de código é executado. * **return NotFound(...)**: O método é encerrado e uma resposta **HTTP 404 Not Found** é enviada ao cliente. O objeto NotFoundObjectResult é retornado, contendo a mensagem de erro especificada. Esta é uma prática recomendada para evitar a execução de consultas mais complexas com um Id inválido.   **4. Execução da Consulta Principal**  C#  var hoteis = await \_context.Hoteis  .Where(h => h.Comodidades.Any(c => c.Id == id))  .AsNoTracking()  .Select(h => new HotelDto(h.Id, h.Nome, h.Cidade))  .ToListAsync();  Esta é uma consulta LINQ to Entities, que o EF Core traduz em uma única e otimizada instrução SQL.   * **\_context.Hoteis**: Inicia a consulta na tabela Hoteis, representada pelo DbSet<Hotel>. * **.Where(h => h.Comodidades.Any(c => c.Id == id))**: Filtra os resultados. O EF Core traduz esta expressão para uma cláusula SQL WHERE que utiliza uma subconsulta ou um JOIN para selecionar apenas os Hoteis (h) que possuem uma relação na tabela de junção com uma Comodidade (c) cujo Id corresponde ao id fornecido. * **.AsNoTracking()**: É uma otimização de performance. Instrui o DbContext a não rastrear as entidades retornadas. Para operações de apenas leitura, isso desativa o rastreamento de alterações, resultando em menor consumo de memória e execução mais rápida da consulta. * **.Select(h => new HotelDto(h.Id, h.Nome, h.Cidade))**: Realiza uma projeção do lado do servidor. Em vez de buscar todas as colunas da tabela Hoteis, ele gera uma consulta SQL SELECT que recupera apenas as colunas Id, Nome e Cidade. Os dados retornados são então usados para instanciar objetos HotelDto diretamente. Isso minimiza o tráfego de dados da rede e o acoplamento com a entidade de domínio. * **.ToListAsync()**: Executa a consulta SQL construída contra o banco de dados de forma assíncrona e materializa os resultados em um objeto List<HotelDto>.   **Observe este comando link direto pelo sql**  -- Supondo que a variável 'id' do C# seja passada como um parâmetro @Id  DECLARE @Id INT = 1; -- Exemplo, para a comodidade com ID 1  SELECT h.Id, h.Nome, h.Cidade FROM Hoteis AS h  WHERE EXISTS (  SELECT 1 FROM HotelComodidade AS hc  WHERE hc.HoteisId = h.Id AND hc.ComodidadesId = @Id  );  **5. Retorno de Sucesso**  C#  return Ok(hoteis);   * **return Ok(hoteis)**: Encapsula a lista de resultados hoteis em um OkObjectResult. Isso envia uma resposta **HTTP 200 OK** ao cliente, indicando que a requisição foi processada com sucesso. O framework serializa o objeto hoteis (a lista de DTOs) para o formato apropriado (geralmente JSON) e o inclui no corpo da resposta. |

**Passo 5: Configurar o Program.cs**

O Program.cs ficará mais limpo. Sua principal função será registrar os serviços (como DbContext e controladores) e configurar o pipeline de requisições.

C#

using HotelApi.Data;

using HotelApi.Models;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// 1. Adicionar serviços ao contêiner.

// Adiciona o suporte para controladores de API

builder.Services.AddControllers();

// Adiciona o DbContext

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseInMemoryDatabase("HotelDb"));

// Adiciona os serviços do Swagger para documentação da API

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// 2. Configurar o pipeline de requisição HTTP.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

// Bloco para popular o banco de dados com dados de exemplo

using (var scope = app.Services.CreateScope())

{

var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

SeedData(dbContext);

}

}

app.UseHttpsRedirection();

app.UseAuthorization();

// Mapeia as rotas definidas nos controladores

app.MapControllers();

app.Run();

// Função para popular dados (pode ser movida para uma classe separada)

void SeedData(AppDbContext context)

{

if (context.Hoteis.Any()) return;

var piscina = new Comodidade { Descricao = "Piscina" };

var wifi = new Comodidade { Descricao = "Wi-Fi Grátis" };

var academia = new Comodidade { Descricao = "Academia" };

var spa = new Comodidade { Descricao = "Spa" };

context.Comodidades.AddRange(piscina, wifi, academia, spa);

var hotelPraia = new Hotel { Nome = "Hotel Sol da Praia", Cidade = "Rio de Janeiro", Comodidades = new List<Comodidade> { piscina, wifi } };

var hotelCentro = new Hotel { Nome = "Plaza Business Hotel", Cidade = "São Paulo", Comodidades = new List<Comodidade> { wifi, academia } };

var hotelLuxo = new Hotel { Nome = "Grand Resort & Spa", Cidade = "Campos do Jordão", Comodidades = new List<Comodidade> { piscina, wifi, academia, spa } };

context.Hoteis.AddRange(hotelPraia, hotelCentro, hotelLuxo);

context.SaveChanges();

}

**Como Executar e Testar**

1. **Execute o projeto:**

Comando ou Código

dotnet run

1. **Acesse os endpoints:** Abra seu navegador ou ferramenta de API e teste as novas rotas, que agora seguem o padrão api/[controller].
   * https://localhost:<porta>/api/hoteis (Busca todos os hotéis com suas comodidades)
   * https://localhost:<porta>/api/hoteis/1 (Busca o hotel de ID 1)
   * https://localhost:<porta>/api/comodidades (Busca todas as comodidades)
   * https://localhost:<porta>/api/comodidades/1/hoteis (Busca os hotéis que têm a comodidade de ID 1 - "Piscina")