**CURSO .NET6: CRIANDO UMA WEB API**

**DOWNLOADS**

VisualStudio Community 2022

.NET 6.0.402

MySQL Community 8.0.31 (MySQL Server e MySQLWorkbench) ***INICIAR O SERVIÇO SEMPRE ANTES DE EXECUTAR O SISTEMA***

Postman - most recent version

**### MACETES**

-Alt + Enter com mouse posicionado sobre a abertura dos colchetes- transforma a estrutura do código para estrutura de arquivo

```

namespace FilmesApi.Models

{

public class Filme

{

}

}

```

PARA

```

namespace FilmesApi.Models;

public class Filme

{

}

```

-Alt + Enter com mouse posicionado sobre o erro de importação de arquivo - constrói a importação automaticamente

**### TEORIA**

**#### API** - Application Programming Interface

Uma interface de serviço - seguindo suas regras, a API disponibiliza serviços sem que o cliente precise se preocupar em como ela faz isso no servidor desse respectivo serviço.

**#### Rest** - um tipo de arquitetura de API - um padrão. Uma API que segue esse padrão é chamada de Restful.

**### ESTRUTURA FRAMEWORK .NET 6.0**

**####** Program - Onde a aplicação é iniciada, configurações, definições, dependências,...

**####** WeatherForecast.cs - Modelo de API base

**####** Appsettings.Development.json e appsettings.json - definições de bancos de dados, senhas, usuários,...

**####** Controllers -> WeatherForecastController.cs - responsável por receber as requisições do usuário

**####** launchSettings.json -> profiles - define a url da aplicação

**### CRIANDO UMA APLICAÇÃO DO 0 - FILMESAPI**

APAGAR Controllers -> WeatherForecastController

APAGAR WeatherForecast

CRIAR Controllers -> FilmeController.cs (classe)

```

using FilmesApi.Models;

using Microsoft.AspNetCore.Mvc;//USO DO [Route("[controller]")] E [ApiController]

namespace FilmesApi.Controllers

{

[ApiController]

[Route("[controller]")]//DESIGNA A ROTA Filme (VEM DO NOME DA CLASS)

public class FilmeController : ControllerBase

{

//LISTA DE FILMES

private static List<Filme> filmes = new List<Filme>();

//MÉTODO QUE ADICIONA UM OBJETO FILME À LISTA

[HttpPost] // DESIGNA QUE O MÉTODO ABAIXO INSERE INFORMAÇÕES NA APLICAÇÃO

public void adicionaFilme([FromBody] Filme filme)

{ //[FromBody] DESGINA QUE O PARÂMETRO VIRÁ DO CORPO DA REQUISIÇÃO

filmes.Add(filme);

Console.WriteLine($"{filme.Titulo}\n{filme.Duracao}");//VERIFICAÇÃO DA INFORMAÇÃO

}

}

}

```

CRIAR Models->Filme.cs

```

namespace FilmesApi.Models;

public class Filme

{

public string Titulo { get; set; }

public string Genero { get; set; }

public int Duracao { get; set; }//DURAÇÃO EM MINUTOS

}

```

Em launchSettings.json - MUDAR

```

"launchBrowser": false, //BROWSER NÃO ABRE AUTOMATICAMENTE AO EXECUTAR A APLICAÇÃO

```

**### INSERÇÃO DE DADOS NA APLICAÇÃO - POST**

Criação de um método que adiciona um objeto filme na memória, dado um estrutura json passada pelo Post. Isso é definido pelo método adicionaFilme no controlador

```

private static int id = 0;//VARIÁVEL QUE SERÁ ATRIBUÍDA AO CAMPO Id do objeto Filme

//MÉTODO QUE ADICIONA UM OBJETO FILME À LISTA

[HttpPost] // DESIGNA QUE O MÉTODO ABAIXO INSERE INFORMAÇÕES NA APLICAÇÃO

public void AdicionaFilme([FromBody] Filme filme)

{ //[FromBody] DESGINA QUE O PARÂMETRO VIRÁ DO CORPO DA REQUISIÇÃO

filme.Id = id++;// 0, 1, 2....

filmes.Add(filme);

}

```

Porém, ao seguir o padrão Rest, sempre que algo for adicionado, é necessário, além da inserção, retornar para o usuário o objeto adicionado e o caminho para ele. Isso pode ser feito pelo método CreatedAtAction().

```

private static int id = 0;//VARIÁVEL QUE SERÁ ATRIBUÍDA AO CAMPO Id do objeto Filme

//MÉTODO QUE ADICIONA UM OBJETO FILME À LISTA

[HttpPost] // DESIGNA QUE O MÉTODO ABAIXO INSERE INFORMAÇÕES NA APLICAÇÃO

public IActionResult AdicionaFilme([FromBody] Filme filme)

{ //[FromBody] DESGINA QUE O PARÂMETRO VIRÁ DO CORPO DA REQUISIÇÃO

filme.Id = id++;// 0, 1, 2....

filmes.Add(filme);

return CreatedAtAction(nameof(RecuperaFilmePorId),

new { id = filme.Id },

filme);

//CreatedAtAction - MÉTODO PADRÃO REST - RETORNA O OBJETO ADICIONADO E O SEU CAMINHO

//nameof(RecuperaFilmePorId) new { id = filme.Id } - CAMINHO DO OBJETO CRIADO

//filme - OBJETO CRIADO

}

```

NO POSTMAN

Execução da inserção



Retorno dela



Com caminho



**### VALIDAR DADOS DO USUÁRIO COM Data Annotations**

Validar a entrada de dados do usuário pode ser feita na própria classe em Models

```

using System.ComponentModel.DataAnnotations;

namespace FilmesApi.Models;

public class Filme

{

[Required(ErrorMessage = "Título do filme obrigatório")]

[MaxLength(50, ErrorMessage = "Título do filme não pode exceder 50 caractéres")]

public string Titulo { get; set; }

[Required(ErrorMessage = "Gênero do filme obrigatório")]

[MaxLength(50, ErrorMessage = "Gênero do filme não pode exceder 50 caractéres")]

public string Genero { get; set; }

[Required(ErrorMessage = "Duração do filme obrigatória")]

[Range(50,600, ErrorMessage ="Duração do filme precisa ser de 50 minutos a 10H")]

public int Duracao { get; set; }//DURAÇÃO EM MINUTOS

}

```

Assim, toda vez que um objeto Filme for usado, seus atributos/propriedades seguirão estas restrições de

**[Required]** - atributo não pode ser vazio

**[MaxLength] ou [StringLength]** - tamanho máximo da string

**[Range]** - valor mínimo e máximo

**(ErrorMessage = “...”)** - personaliza a mensgem de erro mostrada na tela

**### LEITURA DE DADOS NA APLICAÇÃO - GET**

**### TODOS OS DADOS**

Utilizando a mesma classe (FilmeController), agora com um método que recebe HttpGet

```

[HttpGet] //DESIGNA QUE O MÉTODO ABAIXO OBTEM INFORMAÇÕES DA APLICAÇÃO

public IEnumerable<Filme> recuperaFilmes()

{

return filmes;//LISTA DE FILMES

}

```

E no Postman





**### DADOS QUE CORRESPONDEM A UM CRITÉRIO**

Utilizando a mesma classe (FilmeController), agora com um método que recebe HttpGet e um parâmetro id

```

//MÉTODO QUE RETORNA O PRIMEIRO FILME ENCONTRADO, DADO SEU ID

[HttpGet("{id}")]//MÉTODO ABAIXO USA O VERBO GET, MAS COM ID, DIFERENTE DO ACIMA

public Filme? RecuperaFilmePorId(int id)

{ //Filme? - NULLABLE - RETORNO PODE ASSUMIR VALOR NULL

return filmes.FirstOrDefault(filme => filme.Id == id);

}

```

Mudada também a estrutura da classe filme

```

public int Id { get; set; }

```

E NO POSTMAN





**### PAGINAÇÃO DE DADOS COM SKIP E TAKE**

Quando a quantidade de dados em memória for muito grande, pode ser útil dar ao usuário a opção de mostrar partes desses dados somente. Isso pode ser feito com os métodos Skip e Take aplicados na função que usar o verbo Get no Controlador.

```

//MÉTODO QUE LISTA VÁRIOS FILMES DA APLICAÇÃO - PULANDO skip FILMES INICIAIS

//E MOSTRANDO OS PRÓXIMOS take FILMES

[HttpGet] //DESIGNA QUE O MÉTODO ABAIXO OBTEM INFORMAÇÕES DA APLICAÇÃO

public IEnumerable<Filme> RecuperaFilmes([FromQuery] int skip = 0, //SEM DEFINIR, skip É 0

[FromQuery] int take = 50) //SEM DEFINIR, take É 50

{

return filmes.Skip(skip).Take(take);//LISTA DE FILMES

}

```

E NO POSTMAN



PULA OS PRIMEIROS 5 FILMES E MOSTRA OS 10 PRÓXIMOS.

**### CONEXÃO COM BASE DE DADOS**

Downloads necessários

Ferramentas -> Gerenciador de Pacotes NuGet- Gerenciar pacotes solução->Procurar-> Baixar

Microsoft.EntityFrameworkCore - versão. 6.0.10

Microsoft.EntityFrameworkCore.Tools versão. 6.0.10

Pomelo.EntityFrameworkCore.MySql - versão 6.0.2

CRIAR PASTA Data -> Vai intermediar a ligação entre a base de dados e as entidades da aplicação

Data-> class FilmeContext

```

public class FilmeContext : DbContext//CLASSE QUE REMEDIA LIGAÇÃO FILME (ENTIDADE) <-> FILME (BD)

{

//PROPRIEDADES

//PROPRIEDADE DE ACESSO AOS FILMES DA BD

public DbSet<Filme> Filmes{ get; set; }

//CONSTRUTOR QUE USA O CONSTRUTOR BASE (CONSTRUTOR DE DbContext)

public FilmeContext(DbContextOptions<FilmeContext> opts)

: base(opts)

{

}

}

```

Configurar arquivo appsettings.json, considerando

Servidor local, nome da base de dados filme, usuário root e senha root

```

"AllowedHosts": "\*",

"ConnectionStrings": {

"FilmeConnection": "server=localhost;database=filme;user=root;password=root"

}

```

Configurar program.cs

```

using FilmesApi.Data;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

//VARIÁVEL PARA DEFINIÇÕES ABAIXO

var connectionString = builder.Configuration.GetConnectionString("FilmeConnection");

builder.Services.AddDbContext<FilmeContext>(opts =>

opts.UseMySql(connectionString,

ServerVersion.AutoDetect(connectionString)));

```

CONFIGURAR Models->Class Filme

```

public class Filme

{

[Key]//CAMPO Id É A CHAVE DA ENTIDADE/TABELA

[Required]

public int Id { get; set; }

.

.

.

```

Executar as mudanças do código para a BD

Ferramentas-> Gerenciador de Pacotes NuGet->Console Gerenciador de Pacotes \\*\*

*Add-Migration CriandoTabelaDeFilmes* - Constrói a estrutura da tabela \\*\*

*Update-Database* - aplica as mudanças na base de dados MySql \\*\*

Com a estrutura do BD criado e com a conexão com BD configurada no código, aplicar as mudanças para que o código efetivamente utilize essa conexão, na classe FilmeController

```

public class FilmeController : ControllerBase

{

//LISTA DE FILMES (SUBSTITUÍDA PELA CONEXÃO COM BD)

//private static List<Filme> filmes = new List<Filme>();

//private static int id = 0;//VARIÁVEL QUE SERÁ ATRIBUÍDA AO CAMPO Id do objeto Filme

private FilmeContext \_context;//VARIÁVEL DE CONEXÃO COM BD

public FilmeController(FilmeContext context)

{

\_context = context;

}

//MÉTODO QUE ADICIONA UM OBJETO FILME À LISTA

[HttpPost] // DESIGNA QUE O MÉTODO ABAIXO INSERE INFORMAÇÕES NA APLICAÇÃO

public IActionResult AdicionaFilme([FromBody] Filme filme)

{ //[FromBody] DESGINA QUE O PARÂMETRO VIRÁ DO CORPO DA REQUISIÇÃO

//filme.Id = id++;// 0, 1, 2....

//filmes.Add(filme);

\_context.Filmes.Add(filme); //FILME ADICIONADO A BD

\_context.SaveChanges(); //MUDANÇAS SALVAS NA BD

return CreatedAtAction(nameof(RecuperaFilmePorId),

new { id = filme.Id },

filme);

//CreatedAtAction - MÉTODO PADRÃO REST - RETORNA O OBJETO ADICIONADO E O SEU CAMINHO

//nameof(RecuperaFilmePorId) new { id = filme.Id } - CAMINHO DO OBJETO CRIADO

//filme - OBJETO CRIADO

}

//MÉTODO QUE LISTA VÁRIOS FILMES DA APLICAÇÃO - PULANDO skip FILMES INICIAIS

//E MOSTRANDO OS PRÓXIMOS take FILMES

[HttpGet] //DESIGNA QUE O MÉTODO ABAIXO OBTEM INFORMAÇÕES DA APLICAÇÃO

public IEnumerable<Filme> RecuperaFilmes([FromQuery] int skip = 0, //SEM DEFINIR, skip É 0

[FromQuery] int take = 50) //SEM DEFINIR, take É 50

{

//return filmes.Skip(skip).Take(take);//LISTA DE FILMES

return \_context.Filmes.Skip(skip).Take(take);//LISTA DE FILMES

}

//MÉTODO QUE RETORNA O PRIMEIRO FILME ENCONTRADO, DADO SEU ID

[HttpGet("{id}")]//MÉTODO ABAIXO USA O VERBO GET, MAS COM ID, DIFERENTE DO ACIMA

public IActionResult RecuperaFilmePorId(int id)

{ //IActionResult - TIPO DE OBJETO QUE VEM DA INTERFACE ControllerBase

//SERVE PARA GERAR RETORNOS QUE SÃO RESULTADOS DE REQUISIÇÃO

//NESSE CASO, NotFound() E Ok() SÃO MÉTODOS COM RETORNO DO TIPO IActionResult

//var filme = filmes.FirstOrDefault(filme => filme.Id == id);

var filme = \_context.Filmes.FirstOrDefault(filme => filme.Id == id);

if (filme == null) return NotFound();//SE NÃO HOUVER RESULTADO - ERRO 404 - PADRÃO REST

return Ok(filme);//SE HOUVER RESULTADO - NORMAL - 200 OK

}

}

```

**### DTO - Data Transfer Object - Em Criar um Filme (adicionaFilme)**

Downloads necessários:

Ferramentas->Gerenciador de Pacotes Nuget->

AutoMapper (v. 12.0.0)

AutoMapper.Extensions.Microsoft.DependencyInjection (v.12.0.0)

Para que a classe Filmes fique menos exposta no contato com as requisições dos POSTs e GETs, é recomendável usar uma classe DTO, que vai intermediar a ligação entre a FilmeController e Filme, e vai, baseada no Model Filme, mostrar e fornecer informações para as requisições. Sendo assim, a nova classe FilmeDto fica em Data->Dtos-> Classe CreateFilmeDto

```

public class CreateFilmeDto//CLASSE QUE INTERMEDIA ENTRE Controll e Filme

{

[Required(ErrorMessage = "Título do filme obrigatório")]

[StringLength(50, ErrorMessage = "Título do filme não pode exceder 50 caractéres")]

public string Titulo { get; set; }

[Required(ErrorMessage = "Gênero do filme obrigatório")]

[StringLength(50, ErrorMessage = "Gênero do filme não pode exceder 50 caractéres")]

public string Genero { get; set; }

[Required(ErrorMessage = "Duração do filme obrigatória")]

[Range(50, 600, ErrorMessage = "Duração do filme precisa ser de 50 minutos a 10H")]

public int Duracao { get; set; }//DURAÇÃO EM MINUTOS

}

```

Para que seja possível usar essa classe DTO em FilmeController, baixar AutoMapper e AutoMapper.Extensions.Microsoft.DependencyInjection.

Em Program

```

//AUTOMAPPER PODE SER USADO EM TODA A APLICAÇÃO

builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());

Criar Profiles->FilmeProfile.cs, que vai ser a classe que vai possibilitar mapear um objeto CreateFilmeDto em Filme.

using FilmesApi.Data.Dtos;

using AutoMapper;

using FilmesApi.Models;

namespace FilmesApi.Profiles;

public class FilmeProfile : Profile

{

public FilmeProfile()

{

CreateMap<CreateFilmeDto, Filme>();

}

}

```

E as devidas mudanças são feitas em FilmeController, dado que agora a classe usada nos parâmetros de AdicionaFilme é CreateFilmeDto, e não Filme.

```

private IMapper \_mapper;//ATRIBUTO DTO Filme

public FilmeController(FilmeContext context, IMapper mapper)

{

\_context = context;

\_mapper = mapper;

}

//MÉTODO QUE ADICIONA UM OBJETO FILME À LISTA

[HttpPost] // DESIGNA QUE O MÉTODO ABAIXO INSERE INFORMAÇÕES NA APLICAÇÃO

public IActionResult AdicionaFilme([FromBody] CreateFilmeDto filmeDto)

{ //[FromBody] DESGINA QUE O PARÂMETRO VIRÁ DO CORPO DA REQUISIÇÃO

//filme.Id = id++;// 0, 1, 2....

//filmes.Add(filme);

//Objeto filme recebe um objeto Filme a partir do mapeamento de filmeDTO

Filme filme = \_mapper.Map<Filme>(filmeDto);

\_context.Filmes.Add(filme); //FILME ADICIONADO A BD

\_context.SaveChanges(); //MUDANÇAS SALVAS NA BD

return CreatedAtAction(nameof(RecuperaFilmePorId),

new { id = filme.Id },

filme);

}

```

**### Update Filme (já com DTO) - PUT**

Atualizar as informações do objeto com o verbo Http PUT, que atualiza um objeto completo.

Data->Dtos-> Classe UpdateFilmeDto (com mesmo conteúdo de CreateFilmeDto)

Profile->FilmeProfile.cs -> Adicionar ao construtor o método que vai possibilitar o AutoMap a conveter UpdateFilmeDto para Filme

```

public class FilmeProfile : Profile

{

public FilmeProfile()

{

CreateMap<CreateFilmeDto, Filme>();

CreateMap<UpdateFilmeDto, Filme>();

}

}

```

NO POST



**### Update Filme (já com DTO) - PATCH**

Downloads necessários:

Ferramentas->Geranciador de Pacotes Nuget->

Microsoft.AspNetCore.Mvc.NewtonSoftJson (v.6.0.1)

Atualizar as informações do objeto com o verbo Http PATCH, que atualiza parcialmente um objeto.

Programs->

```

builder.Services.AddControllers().AddNewtonsoftJson();

Profiles-> FilmeProfile

public FilmeProfile()

{

CreateMap<CreateFilmeDto, Filme>();

CreateMap<UpdateFilmeDto, Filme>();

CreateMap<Filme, UpdateFilmeDto>();

}

```

FilmeController

```

//MÉTODO QUE ATUALIZA PARCIALMENTE UM FILME, DADO SEU ID, COM AS INFORMAÇÕES DO BODY

[HttpPatch("{id}")]//HttpPatch - DESIGNA UMA ATUALIZAÇÃO PARCIAL DO OBJETO

public IActionResult AtualizaFilmeParcial(int id, JsonPatchDocument<UpdateFilmeDto> patch)

{ //RECEBER NO PARÂMETRO UM patch DE ATUALIZAÇÃO

//SALVO O FILME PROCURADO EM filme

var filme = \_context.Filmes.FirstOrDefault(filme => filme.Id == id);

if (filme == null) return NotFound();

//VALIDAÇÃO DO PATCH RECEBIDO

//1º TORNAR O OBJETO Filme ENCONTRADO NO BANCO, UM UpdateFilmeDto - PARA PODER VERIFICAR SE É APLICÁVEL O PATCH

var filmeParaAtualizar = \_mapper.Map<UpdateFilmeDto>(filme); //\*ADICIONAR EM FilmeProfile CreateMap<Filme, UpdateFilmeDto>();

//TENTO APLICAR AS MUDANÇAS DO PATCH AO OBJETO filmeParaAtualizar

patch.ApplyTo(filmeParaAtualizar, ModelState);

//SE NÃO CONSEGUIR VALIDAR A ATUALIZAÇÃO COM O PATCH, O ERRO É INDICADO

if (!TryValidateModel(ModelState))

{

return ValidationProblem(ModelState);

}

//SE TUDO DER CERTO E O FILME FOR ATUÁLIZAVEL, CONSIDERANDO AS MUDANÇAS INDICADAS

\_mapper.Map(filmeParaAtualizar, filme);//filmeParaAtualizar É MAPEADO/ATUALIZADO PARA filme - MUDANÇA NO BANCO

\_context.SaveChanges();

return NoContent(); //RETORNO REST PARA UPDATE (204 NoContent)

}

```

NO POSTMAN - ATUALIZAR O TÍTULO E DURAÇÃO



**### DELETE FILME**

```

[HttpDelete("{id}")]//DESIGNA QUE A FUNÇÃO ABAIXO UTILIZARÁ DELEÇÃO

public IActionResult DeletaFilmes(int id)

{

var filme = \_context.Filmes.FirstOrDefault(filme => filme.Id == id);

if (filme == null) return NotFound();

\_context.Remove(filme);//DELETA O FILME

\_context.SaveChanges();

return NoContent();

}

```

**### LEITURA DE DADOS - COM DTO**

Data->Dtos->ReadFilmeDTO

```

public class ReadFilmeDto

{

public string Titulo { get; set; }

public string Genero { get; set; }

public int Duracao { get; set; }

public DateTime HoraDaConsulta { get; set; } = DateTime.Now;//VARIÁVEL EXCLUSIVA DO ReadDto

//Hora que o filme foi consultado

}

```

FilmeController-> Ambos métodos de leitura de dados modificados para retornar DTOs

```

public IEnumerable<ReadFilmeDto> RecuperaFilmes([FromQuery] int skip = 0, //SEM DEFINIR, skip É 0

[FromQuery] int take = 50) //SEM DEFINIR, take É 50

{

//return filmes.Skip(skip).Take(take);//LISTA DE FILMES

//return \_context.Filmes.Skip(skip).Take(take);//LISTA DE FILMES

//RETORNO É UMA MAPPER DA LISTA DTO DO TIPO ReadFilmeDto

return \_mapper.Map<List<ReadFilmeDto>>

(\_context.Filmes.Skip(skip).Take(take));

}

//MÉTODO QUE RETORNA O PRIMEIRO FILME ENCONTRADO, DADO SEU ID

[HttpGet("{id}")]//MÉTODO ABAIXO USA O VERBO GET, MAS COM ID, DIFERENTE DO ACIMA

public IActionResult RecuperaFilmePorId(int id)

{ //IActionResult - TIPO DE OBJETO QUE VEM DA INTERFACE ControllerBase

//SERVE PARA GERAR RETORNOS QUE SÃO RESULTADOS DE REQUISIÇÃO

//NESSE CASO, NotFound() E Ok() SÃO MÉTODOS COM RETORNO DO TIPO IActionResult

//var filme = filmes.FirstOrDefault(filme => filme.Id == id);

var filme = \_context.Filmes.FirstOrDefault(filme => filme.Id == id);

if (filme == null) return NotFound();//SE NÃO HOUVER RESULTADO - ERRO 404 - PADRÃO REST

var filmeDto = \_mapper.Map<ReadFilmeDto>(filme);//O RETORNO É UM Dto

return Ok(filmeDto);//SE HOUVER RESULTADO - NORMAL - 200 OK

}

```

FilmeProfile->Adicionar

```

CreateMap<Filme, ReadFilmeDto>();

```

**### DOCUMENTAÇÃO DA APLICAÇÃO**

A documentação, pelo swagger, pode ser feita através dos próprios métodos

FilmeController->

```

//DOCUMENTAÇÃO SWAGGER

/// <summary>

/// Adiciona um filme à base de dados.

/// </summary>

/// <param name="filmeDto">DTO usado pelo mapper para efetivar as mudanças na base de dados</param>

/// <returns>IActionResult</returns>

/// <response code="201">Em caso de inserção bem sucedida</response>

[HttpPost]

[ProducesResponseType(StatusCodes.Status201Created)]

public IActionResult AdicionaFilme([FromBody] CreateFilmeDto filmeDto)

{ //[FromBody] DESGINA QUE O PARÂMETRO VIRÁ DO CORPO DA REQUISIÇÃO

//filme.Id = id++;// 0, 1, 2....

//filmes.Add(filme);

//Objeto filme recebe um objeto Filme a partir do mapeamento de filmeDTO

Filme filme = \_mapper.Map<Filme>(filmeDto);

\_context.Filmes.Add(filme); //FILME ADICIONADO A BD

\_context.SaveChanges(); //MUDANÇAS SALVAS NA BD

return CreatedAtAction(nameof(RecuperaFilmePorId),

new { id = filme.Id },

filme);

//CreatedAtAction - MÉTODO PADRÃO REST - RETORNA O OBJETO ADICIONADO E O SEU CAMINHO

//nameof(RecuperaFilmePorId) new { id = filme.Id } - CAMINHO DO OBJETO CRIADO

//filme - OBJETO CRIADO

}

Program->

builder.Services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new Microsoft.OpenApi.Models.OpenApiInfo { Title = "FilmesApi", Version = "v1" });

var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";

var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);

c.IncludeXmlComments(xmlPath);

});

Para habilitar a exibição de documentação no swagger

Clique duplo no projeto->

<PropertyGroup>

<TargetFramework>net6.0</TargetFramework>

<Nullable>enable</Nullable>

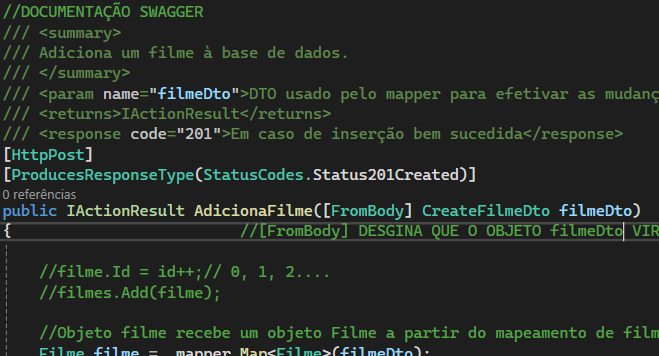
<ImplicitUsings>enable</ImplicitUsings>

<GenerateDocumentationFile>true</GenerateDocumentationFile>

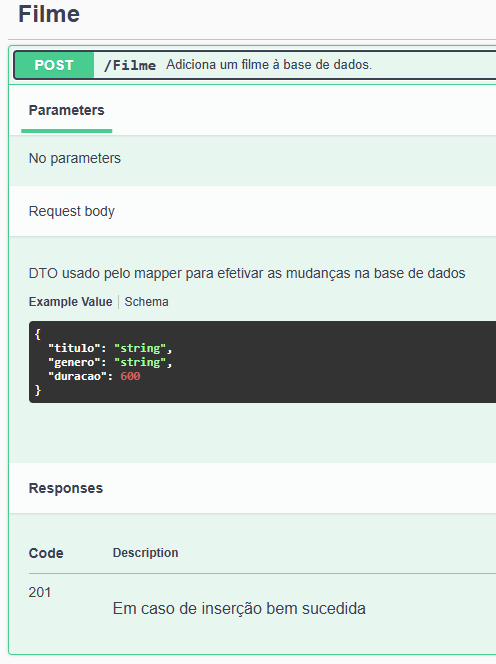
</PropertyGroup>

```

Ao final, isso



É apresentado dessa forma pelo swagger



**CURSO .NET6: CRIANDO UMA WEB API - PARTE 2**

**ENTIDADE CINEMA**

Models-> Cinema.cs

public class Cinema

{

[Key]

[Required]

public int Id { get; set; }

[Required(ErrorMessage = "Campo nome é obrigatório")]

public string Nome { get; set; }

}

Data->Dtos->

CreateCinemaDto - Nome (required)

public class CreateCinemaDto

{

[Required(ErrorMessage = "Campo nome é obrigatório")]

public string Nome { get; set; }

}

ReadCinemaDto - Id e Nome

public class ReadCinemaDto

{

public int Id { get; set; }

public string Nome { get; set; }

}

UpdateCinemaDto - Nome (required)

public class UpdateCinemaDto

{

[Required(ErrorMessage = "Campo nome é obrigatório")]

public string Nome { get; set; }

}

FilmeContext -> adicionar propriedade de conexão com tabela Cinemas

public DbSet<Cinema> Cinemas{ get; set; }

Profiles-> criar classe CinemaProfile

using AutoMapper;

using FilmesApi.Data.Dtos;

using FilmesApi.Models;

namespace FilmesApi.Profiles

{

public class CinemaProfile : Profile

{

public CinemaProfile()

{

CreateMap<CreateCinemaDto, Cinema>();

CreateMap<Cinema, ReadCinemaDto>();

CreateMap<UpdateCinemaDto, Cinema>();

}

}

}

CinemaController

using AutoMapper;

using FilmesApi.Data.Dtos;

using FilmesApi.Data;

using FilmesApi.Models;

using Microsoft.AspNetCore.Mvc;

namespace FilmesApi.Controllers

{

[ApiController]

[Route("[controller]")]

public class CinemaController : ControllerBase

{

//PROPRIEDADES-ATRIBUTOS

private FilmeContext \_context;

private IMapper \_mapper;

//CONSTRUTOR

public CinemaController(FilmeContext context, IMapper mapper)

{

\_context = context;

\_mapper = mapper;

}

//DEMAIS MÉTODOS

//MÉTODO QUE ADICIONA UM CINEMA AO BANCO

[HttpPost]

public IActionResult AdicionaCinema([FromBody] CreateCinemaDto cinemaDto)

{

Cinema cinema = \_mapper.Map<Cinema>(cinemaDto);

\_context.Cinemas.Add(cinema);

\_context.SaveChanges();

return CreatedAtAction(

nameof(RecuperaCinemasPorId), new { Id = cinema.Id }, cinemaDto);

}

//MÉTODO QUE MOSTRA TODOS OS CINEMAS DA APLICAÇÃO

[HttpGet]

public IEnumerable<ReadCinemaDto> RecuperaCinemasDto()

{

return \_mapper.Map<List<ReadCinemaDto>>(

\_context.Cinemas.ToList());

}

//MÉTODO QUE RETORNA UM CINEMA DADO SEU ID

[HttpGet("{id}")]

public IActionResult RecuperaCinemasPorId(int id)

{

Cinema cinema = \_context.Cinemas.FirstOrDefault(cinema => cinema.Id == id);

if (cinema != null)

{

ReadCinemaDto cinemaDto = \_mapper.Map<ReadCinemaDto>(cinema);

return Ok(cinemaDto);

}

return NotFound();

}

//MÉTODO QUE ATUALIZA UM OBJETO CINEMA POR COMPLETO

//NÃO HÁ UPDATE COM VERBO PATCH, CONSIDERANDO QUE CINEMA SÓ TEM UM CAMPO

//MODIFICÁVEL - NOME

[HttpPut("{id}")]

public IActionResult AtualizaCinema(int id, [FromBody] UpdateCinemaDto cinemaDto)

{

Cinema cinema = \_context.Cinemas.FirstOrDefault(cinema => cinema.Id == id);

if (cinema == null) return NotFound();

\_mapper.Map(cinemaDto, cinema);//REALIZAÇÃO DO UPDATE NA BASE

\_context.SaveChanges();

return NoContent();

}

//MÉTODO QUE DELETA UM CINEMA, DADO SEU ID

[HttpDelete("{id}")]

public IActionResult DeleteCinema(int id)

{

Cinema cinema = \_context.Cinemas.FirstOrDefault(cinema => cinema.Id == id);

if (cinema == null) return NotFound();

\_context.Remove(cinema);

\_context.SaveChanges();

return NoContent();

}

}

}

**ENTIDADE ENDEREÇO**

Model

public class Endereco

{

[Key]

[Required]

public int Id { get; set; }

[Required]

public string Logradouro { get; set; }

[Required]

public int Numero { get; set; }

}

DTOs

CreateEnderecoDto

public class CreateEnderecoDto

{

[Required]

public string Logradouro { get; set; }

[Required]

public int Numero { get; set; }

}

ReadEnderecoDto

public class ReadEnderecoDto

{

public int Id { get; set; }

public string Logradouro { get; set; }

public int Numero { get; set; }

}

UpdateEnderecoDto

public class UpdateEnderecoDto

{

[Required]

public string Logradouro { get; set; }

[Required]

public int Numero { get; set; }

}

FilmeContext-> Enderecos

public DbSet<Endereco> Enderecos{ get; set; }

EnderecoProfile

using AutoMapper;

using FilmesApi.Data.Dtos;

using FilmesApi.Models;

namespace FilmesApi.Profiles

{

public class EnderecoProfile : Profile

{

public EnderecoProfile()

{

CreateMap<CreateEnderecoDto, Endereco>();

CreateMap<Endereco, ReadEnderecoDto>();

CreateMap<UpdateEnderecoDto, Endereco>();

}

}

}

EnderecoController

using AutoMapper;

using FilmesApi.Data;

using FilmesApi.Data.Dtos;

using FilmesApi.Models;

using Microsoft.AspNetCore.Mvc;

namespace FilmesApi.Controllers

{

public class EnderecoController : ControllerBase

{

//PROPRIEDADES - ATRIBUTOS

private FilmeContext \_context;

private IMapper \_mapper;

//CONSTRUTOR

public EnderecoController(FilmeContext context, IMapper mapper)

{

\_context = context;

\_mapper = mapper;

}

//DEMAIS MÉTODOS

//MÉTODO QUE ADICIONA UM ENDEREÇO A BD

[HttpPost]

public IActionResult adicionaEndereco([FromBody] CreateEnderecoDto enderecoDto)

{

Endereco endereco = \_mapper.Map<Endereco>(enderecoDto);

\_context.Enderecos.Add(endereco);

\_context.SaveChanges();

return CreatedAtAction(

nameof(RecuperaEnderecosPorId), new { Id = endereco.Id }, endereco);

}

//MÉTODO QUE MOSTRA TODOS OS ENDEREÇOS CADASTRADOS

[HttpGet]

public IEnumerable<ReadEnderecoDto> RecuperaEnderecos()

{

return \_mapper.Map<List<ReadEnderecoDto>>(\_context.Enderecos);

}

//MÉTODO QUE MOSTRA UM ENDEREÇO, DADO SEU ID

[HttpGet("{id}")]

public IActionResult RecuperaEnderecosPorId(int id)

{

Endereco endereco = \_context.Enderecos.FirstOrDefault(endereco => endereco.Id == id);

if(endereco != null)

{

ReadEnderecoDto enderecoDto = \_mapper.Map<ReadEnderecoDto>(endereco);

return Ok(enderecoDto);

}

return NotFound();

}

//MÉTODO QUE ATUALIZA TODO UM ENDEREÇO, DADO SEU ID

[HttpPut("{id}")]

public IActionResult AtualizaEndereco(int id, [FromBody] UpdateEnderecoDto enderecoDto)

{

Endereco endereco = \_context.Enderecos.FirstOrDefault(endereco => endereco.Id == id);

if(endereco == null)

{

return NotFound();

}

\_mapper.Map(enderecoDto, endereco);//ATUALIZA NO BD

\_context.SaveChanges();

return NoContent();

}

//MÉTODO QUE DELETA UM ENDEREÇO, DADO SEU ID

[HttpDelete("{id}")]

public IActionResult DeletaEndereco(int id)

{

Endereco endereco = \_context.Enderecos.FirstOrDefault(endereco => endereco.Id == id);

if (endereco == null)

{

return NotFound();

}

\_context.Remove(endereco);

\_context.SaveChanges();

return NoContent();

}

}

}

**RELAÇÃO ENTRE ENDIDADES CINEMA <-> ENDERECO**

Downloads necessários

Ferramentas->Gerenciador de Pacotes Nuget->Gerenciar pacotes para solução->

*Microsoft.EntityFramework.Proxies*

**RELAÇÃO 1:1 Cinema<->Endereço**

(Cinema não existe sem endereço / Endereço existe sem Cinema)

Mudar models Cinema e Endereço

Model Cinema

//PROPRIEDADES RELAÇÃO 1:1 CINEMA <-> ENDERECO

//TABELA CINEMA TEM DUAS COLUNAS - UMA SENDO O ID DO ENDEREÇO E OUTRA SENDO O ENDEREÇO EM SI

public int EnderecoId { get; set; }

public virtual Endereco Endereco { get; set; }

Model Endereco

//PROPRIEDADE RELAÇÃO 1:1 CINEMA<->ENDEREÇO

public virtual Cinema Cinema { get; set; }

\*Lembrar: a entidade com a chave estrangeira - guarda a propriedade ...Id {get;set;}

\*Cinema contém uma chave estrangeira para Endereço, logo, Cinema fica com EnderecoId;

Mudar CreateCinemaDto

public int EnderecoId { get; set; }

Mudar ReadCinemaDto

public ReadEnderecoDto ReadEnderecoDto { get; set; }

Executar as mudanças do código para a BD

Ferramentas-> Gerenciador de Pacotes NuGet->Console Gerenciador de Pacotes \\*\*

*Add-Migration Relacao-Cinema-Endereco* - Constrói a estrutura da tabela \\*\*

*Update-Database* - aplica as mudanças na base de dados MySql \\*\*

Mudar Program - linha 10

builder.Services.AddDbContext<FilmeContext>(opts =>

opts.UseLazyLoadingProxies().UseMySql(connectionString,

ServerVersion.AutoDetect(connectionString)));

Até o momento, sempre que um cinema é adicionado, ele não tem endereço, isso porque mesmo configurando tudo acima, ainda falta configurar o CinemaProfile, dado que sem essa configuração, a função do AutoMap não consegue relacionar o Cinema com o Endereço

Mudar CinemProfile->

public CinemaProfile()

{

CreateMap<CreateCinemaDto, Cinema>();

//USADO NO RecuperaCinemas - CONFIGURAR COMO O AUTOMAPPER FUNCIONA

CreateMap<Cinema, ReadCinemaDto>().ForMember(cinemaDto => cinemaDto.ReadEnderecoDto,

opt => opt.MapFrom(cinema => cinema.Endereco));

//ForMember(cinemaDto - PARA O MEMBRO DO DESTINO, QUE É DO TIPO ReadCinemaDto

//=> cinemaDto.ReadEnderecoDto - ACESSANDO O CAMPO ReadEnderecoDto, QUE É UM CAMPO DESSE OBJETO

//opt => opt.MapFrom(cinema => cinema.Endereco - QUERO PEGAR, DA ORIGEM, O CAMPO Endereco

CreateMap<UpdateCinemaDto, Cinema>();

DESSA FORMA, AS ENTIDADES CINEMA E ENDERECO FICAM RELACIONADAS CORRETAMENTE, CONSIDERANDO QUE:

-RELACIONAMENTO 1:1;

-O ENDEREÇO JÁ EXISTE ANTES DE INSTANCIAR O CINEMA;

-A RELAÇÃO SE DÁ NO MOMENTO DA INSERÇÃO DO CINEMA, QUE NO CAMPO EnderecoId GUARDA UMA REFERÊNCIA AO CAMPO Id da ENTIDADE ENDEREÇO.

**ENTIDADE SESSAO**

**RELACIONANDO ENTIDADES SESSAO<->FILME 1:N**

\*Uma sessão tem um filme. Um filme pode estar ao mesmo tempo em diversas sessões

\*Não existe sessão sem filme. Existe filme sem sessão.

Model Sessao

public class Sessao

{

[Key]

[Required]

public int Id { get; set; }

//RELAÇÃO SESSÃO<->FILME

[Required]

public int FilmeId { get; set; }

public virtual Filme Filme { get; set; }

}

DTOs

CreateSessaoDto

public class CreateSessaoDto

{

public int FilmeId { get; set; }

}

ReadSessaoDto

public class ReadSessaoDto

{

public int Id { get; set; }

}

SessaoProfile

public class SessaoProfile : Profile

{

public SessaoProfile()

{

CreateMap<CreateSessaoDto, Sessao>();

CreateMap<Sessao, ReadSessaoDto>();

}

}

FilmeContext->Adicionar

public DbSet<Sessao> Sessoes{ get; set; }

SessaoController

namespace FilmesApi.Controllers

{

[ApiController]

[Route("[controller]")]

public class SessaoController : ControllerBase

{

//PROPRIEDADES

private FilmeContext \_context;

private IMapper \_mapper;

//CONSTRUTOR

public SessaoController(FilmeContext context, IMapper mapper) //ctor

{

\_context = context;

\_mapper = mapper;

}

//MÉTODO QUE ADICIONA UMA SESSÃO AO SISTEMA

[HttpPost]

public IActionResult AdicionaSessao(CreateSessaoDto dto)

{

Sessao sessao = \_mapper.Map<Sessao>(dto);

\_context.Sessoes.Add(sessao);

\_context.SaveChanges();

return CreatedAtAction(nameof(RecuperaSessoesPorId), new { Id = sessao.Id }, sessao);

}

//MÉTODO QUE MOSTRA AS SESSÕES DO SISTEMA

[HttpGet]

public IEnumerable<ReadSessaoDto> RecuperaSessoes()

{

return \_mapper.Map<List<ReadSessaoDto>>(\_context.Sessoes.ToList());

}

//MÉTODO QUE MOSTRA UMA SESSÃO, DADO SEU ID

public IActionResult RecuperaSessoesPorId(int id)

{

Sessao sessao = \_context.Sessoes.FirstOrDefault(sessao => sessao.Id == id);

if(sessao != null)

{

ReadSessaoDto sessaoDto = \_mapper.Map<ReadSessaoDto>(sessao);

return Ok(sessaoDto);

}

return NotFound();

}

}

}

Mudança Model Filme

//RELAÇÃO SESSÃO<->FILME

//FILME PODE ESTAR AO MESMO TEMPO EM 1 OU MUITAS SESSÕES

//COLEÇÃO DE SESSÕES

public virtual ICollection<Sessao> Sessoes { get; set; }

Executar as mudanças do código para a BD

Ferramentas-> Gerenciador de Pacotes NuGet->Console Gerenciador de Pacotes \\*\*

*Add-Migration Relacao-Sessao-Filme* - Constrói a estrutura da tabela \\*\*

*Update-Database* - aplica as mudanças na base de dados MySql \\*\*

Até aqui, as entidades estão relacionadas corretamente, mas ainda faltam algumas melhorias, como:

Mostrar na consulta de sessões, qual filme dessa sessão;

Mostar na consulta de filmes, em quais sessões cada filme está;

**RELACIONANDO ENTIDADE SESSAO<->CINEMA 1:N**

Um cinema passa várias sessões. Cada sessão só acontece em um cinema.

Mudar Model Sessao -> Adicionar

//RELAÇÃO CINEMA<->SESSAO

[Required]

public int CinemaId { get; set; }

public virtual Cinema Cinema { get; set; }

Mudar Model Cinema -> Adicionar

//RELAÇÃO 1:N CINEMA<-> SESSÃO

public virtual ICollection<Sessao> Sessoes { get; set; }

Executar as mudanças do código para a BD

Ferramentas-> Gerenciador de Pacotes NuGet->Console Gerenciador de Pacotes

*Add-Migration Relacao-Cinema-Sessao*- Constrói a estrutura da tabela

*Update-Database* - aplica as mudanças na base de dados MySql

Problema: A chave estrangeira CinemaId, em Sessao, começa com 0 por padrão, enquanto a chave primária Id em Cinema começa com 1, o que impede o funcionamento da chave estrangeira.

Para resolver:

1º - Remover a migration no console Nuget

Remove-Migration

2º - Alterar a tabela Sessoes e remover a coluna CinemaId - no MySql Workbench

alter table sessoes drop column CinemaId;

3º - Permitir que, na tabela Sessoes, a coluna CinemaId possar, ser nula, retirando o “[Required]” e tornando o atribtuo *nullable*

//RELAÇÃO CINEMA<->SESSAO

public int? CinemaId { get; set; }

public virtual Cinema Cinema { get; set; }

Executar as mudanças do código para a BD

Ferramentas-> Gerenciador de Pacotes NuGet->Console Gerenciador de Pacotes

*Add-Migration Relacao-Cinema-Sessao*- Constrói a estrutura da tabela

*Update-Database* - aplica as mudanças na base de dados MySql

Até aqui, tudo funciona, mas a consulta dos filmes gera erro, porque em FilmeController, o método RecuperaFilmes, no seu retorno, por usar

\_context.Filmes retorna um Queryble, que não pode ser convertido pelo AutoMapper.

Isso se resolve colocando .ToList() ao final dele.

**MELHORIAS**

Ao consultar filmes, mostrar as sessões relacionadas

ReadFilmeDto->Adicionar

public ICollection<ReadSessaoDto> Sessoes{ get; set; }

Ao consultar cinemas, mostrar as sessões relacionadas

ReadCinemaDto->Adicionar

public ICollection<ReadSessaoDto> Sessoes{ get; set; }

Configurar o AutoMapper para conseguir mapear, em Filmes, as Sessões relacionadas

FilmeProfile

CreateMap<Filme, ReadFilmeDto>()

.ForMember(filmeDto => filmeDto.Sessoes,

opt => opt.MapFrom(filme => filme.Sessoes));

//ForMember(filmeDto - PARA O MEMBRO DO DESTINO, QUE É DO TIPO ReadFilmeDto

//=> cinemaDto.Sessoes - ACESSANDO O CAMPO Sessoes, QUE É UM CAMPO DESSE OBJETO

//opt => opt.MapFrom(filme => filme.Sessoes - QUERO PEGAR, DA ORIGEM, O CAMPO Sessoes

Configurar o AutoMapper para conseguir mapear, em Cinemas, as Sessões relacionadas

CinemaProfile

//USADO NO CinemaController->RecuperaCinemas - CONFIGURAR COMO O AUTOMAPPER FUNCIONA

CreateMap<Cinema, ReadCinemaDto>()

.ForMember(cinemaDto => cinemaDto.Endereco,

opt => opt.MapFrom(cinema => cinema.Endereco))

.ForMember(cinemaDto => cinemaDto.Sessoes,

opt => opt.MapFrom(cinema => cinema.Sessoes));

//ForMember(cinemaDto - PARA O MEMBRO DO DESTINO, QUE É DO TIPO ReadCinemaDto

//=> cinemaDto.Endereco - ACESSANDO O CAMPO Endereco, QUE É UM CAMPO DESSE OBJETO

//opt => opt.MapFrom(cinema => cinema.Endereco - QUERO PEGAR, DA ORIGEM, O CAMPO Endereco

**RELACIONANDO ENTIDADE FILME<->CINEMA N:N**

Relacionamentos N:N são mapeados com uma tabela separada para eles. No nosso sistema, ao invés de criar uma tabela com essas informações, usaremos a tabela Sessao, que já guarda as informações que relacionam Filme e Cinema.

Para que a entidade Sessão funcione corretamente, nesse caso, ela passar por algumas mudanças

Drop database no WorkBench para evitar conflitos

Tornar o campo Model Sessao->FilmeId nullable

//RELAÇÃO SESSÃO<->FILME

public int? FilmeId { get; set; }

Executar as mudanças do código para a BD

Ferramentas-> Gerenciador de Pacotes NuGet->Console Gerenciador de Pacotes

*Add-Migration “FilmeId nulo”*

*Update-Database*

Retirar do model Sessao o campo Id

Construir, em FilmeContext, o código que vai relacionar os campos FilmeId e CinemaId como chave primária de Sessao e vai definir como a tabela Sessao se relaciona com Filme e Cinema

//DEMAIS MÉTODOS

//MÉTODO QUE CRIA A CHAVE PRIMÁRIA DE SESSOES - COMPOSTA por FilmeId + CinemaId

//E QUE RELACIONA A TABELA SESSOES COM FILME E CINEMA (AGORA QUE ELA É A TABLE QUE GUARDA A

//RELAÇÃO N:N ENTRE FILME E CINEMA

protected override void OnModelCreating(ModelBuilder builder)

{

//CADA SESSÃO TEM COMO CHAVE PRIMÁRIA FilmeId+CinemaId

builder.Entity<Sessao>().HasKey(sessao => new { sessao.FilmeId, sessao.CinemaId });

//COMO A SESSÃO SE RELACIONA COM CINEMA?

builder.Entity<Sessao>().HasOne(sessao => sessao.Cinema) //1 SESSÃO -> 1 CINEMA

.WithMany(cinema => cinema.Sessoes) //1 CINEMA -> 1 OU MUITAS SESSÕES

.HasForeignKey(sessao => sessao.CinemaId); //CHAVE ESTRANGEIRA: SESSAO->CinemaId PARA CHAVE PRIMÁRIA DE Cinema

//COMO A SESSÃO SE RELACIONA COM FILME?

builder.Entity<Sessao>().HasOne(sessao => sessao.Filme) //1 SESSÃO -> 1 FILME

.WithMany(filme => filme.Sessoes) //1 FILME -> 1 OU MUITAS SESSÕES

.HasForeignKey(sessao => sessao.FilmeId); //CHAVE ESTRANGEIRA: SESSAO->FilmeId PARA CHAVE PRIMÁRIA DE Filme

}

FEITAS ESSAS MUDANÇAS, COMO NÃO HÁ MAIS O CAMPO Id em Sessao, mudar também

SessaoController -> método RecuperaSessoesPorId

//MÉTODO QUE MOSTRA UMA SESSÃO, DADO SEU ID

[HttpGet("{filmeId}/{cinemaId}")]

public IActionResult RecuperaSessoesPorId(int filmeId, int cinemaId)

{

Sessao sessao = \_context.Sessoes.FirstOrDefault(sessao => sessao.FilmeId == filmeId &&

sessao.CinemaId == cinemaId);

if(sessao != null)

{

ReadSessaoDto sessaoDto = \_mapper.Map<ReadSessaoDto>(sessao);

return Ok(sessaoDto);

}

return NotFound();

}

SessaoControler-> Método AdicionaSessao

//MÉTODO QUE ADICIONA UMA SESSÃO AO SISTEMA

[HttpPost]

public IActionResult AdicionaSessao(CreateSessaoDto dto)

{

Sessao sessao = \_mapper.Map<Sessao>(dto);

\_context.Sessoes.Add(sessao);

\_context.SaveChanges();

return CreatedAtAction(nameof(RecuperaSessoesPorId), new { filmeId = sessao.FilmeId,

cinemaId = sessao.CinemaId}

, sessao);

}

ReadSessaoDto

public class ReadSessaoDto

{

public int FilmeId { get; set; }

public int CinemaId { get; set; }

}

Executar as mudanças do código para a BD

Ferramentas-> Gerenciador de Pacotes NuGet->Console Gerenciador de Pacotes

*Add-Migration “Cinema e Filme”*

*Update-Database*

Feitas essas mudanças, o sistema agora cadastra cada sessão como sendo uma maneira de identificar unicamente um filme associado a uma sessão.

**TRATANDO DELEÇÕES DE ENTIDADES RELACIONADAS**

Por padrão, usando o Entity Framework, sempre que uma entidade é deletada, as entidades a qual ela está ligada também são, para que a base de dados não fique com informações faltantes. Isso pode ser ruim, dado que, a deleção do endereço leva a deleção do cinema, que leva a deleção da sessão (efeito cascata). Isso pode ser tratado mudando o comportamento no FilmeContext, adicionando esse comportamento ao método OnModelCreating

builder.Entity<Endereco>().HasOne(endereco => endereco.Cinema) //1 ENDERECO -> 1 CINEMA

.WithOne(cinema => cinema.Endereco) //1 CINEMA -> 1 ENDERECO

.OnDelete(DeleteBehavior.Restrict); //DELEÇÃO RESTRITA

//NÃO DELETA SE HOUVER CHAVES PRIMÁRIAS NA TUPLA

**UTILIZANDO SQL NOS MÉTODOS DO CONTROLLER**

Os métodos das classes controllers podem ser modificados para usar comandos SQL no próprio código. Por exemplo, para o método CinemaController->RecuperaCinemas, mudá-lo para que, se for passado um parâmetro enderecoId na URL da requisição, o método retorna somente o cinema que tem esse enderecoId (assim como foi feito com skip e take em FilmeController->RecuperaFilmes).

//MÉTODO QUE

//MOSTRA TODOS OS CINEMAS DA APLICAÇÃO (SEM PARÂMETROS NA REQUISIÇÃO)

//MOSTRA OS CINEMAS CUJO ENDEREÇO SEJA enderecoId

[HttpGet]

public IEnumerable<ReadCinemaDto> RecuperaCinemas([FromQuery] int? enderecoId = null)

{

//SEM PARÂMETROS NA URL - RETORNA TODOS CINEMAS

if(enderecoId == null)

{

return \_mapper.Map<List<ReadCinemaDto>>(

\_context.Cinemas.ToList());

}

//COM PARÂMETRO - RETORNA CINEMAS CUJO ENDEREÇO SEJA enderecoId

return \_mapper.Map<List<ReadCinemaDto>>

(\_context.Cinemas.FromSqlRaw($"SELECT Id, Nome, EnderecoId FROM cinemas where cinemas.EnderecoId = {enderecoId}").ToList());

}