

API Rest com .NET Core 5: Operações essenciais com verbos HTTP

1° criar um projeto '**API Web do ASP.NET Core**' com .NET Core 5.0;

Baixar **Postman**; baixar MySQL

Vamos criar uma API que vai fornecer nomes de filmes a quem realizar a requisição

Quando enviamos requisições para a API a fim de criarmos um novo recurso, é padrão explicitarmos um verbo HTTP para tal ação.

Nosso primeiro método (**AdicionaFilme**) vai postar/enviar (**HttpPost**) informações via Postman para o servidor da nossa API (localhost)

1° Criar o controller com suas palavras reservadas

Nosso método adiciona filmes na lista static do controller e mostra no console o título do envio HttpPost feito pelo Postman

```
[ApiController] // classe para criar a API
[Route("[controller]")] // define q a rota para acessar a API será sempre 'nome' + 'Controller', ou seja, FilmeController
public class FilmeController : ControllerBase
{
    private static List<Filme> filmes = new List<Filme>();

    // palavra reservada do método Rest, que significa CRIAR alguma coisa
    [HttpPost] // estou dizendo que a inf enviada pelo Programa Postman vai postar a info no server da minha API
    public void AdicionaFilme([FromBody] Filme filme) // recebe a inf enviada pelo postman pelo seu "Body"
    {
        filmes.Add(filme);
        Console.WriteLine(filme.Titulo);
    }
}
```

2° Criar a class Filme com suas propriedades

```

public class Filme
{
    1 referência
    public string Titulo { get; set; }
    0 referências
    public string Diretor { get; set; }
    0 referências
    public string Genero { get; set; }
    0 referências
    public int Duracao { get; set; }
}

```

3° configurar o Postman para enviar os dados sobre filmes via **JSON** e pelo seu **BODY**

4° ajustar o caminho de envio dos dados para **http://localhost:5000/filme**

The screenshot shows the Postman application interface. At the top, the URL bar is set to `http://localhost:5000/filme`. The request method is **POST**. The **Body** tab is selected, showing a JSON payload: `{ "titulo": "O senhor dos aneis", "diretor": "Peter Jackson", "genero": "Aventura", "duracao": 100 }`. The **Headers** tab shows 8 headers. The **Response** section at the bottom shows a status of **200 OK**, a time of **6 ms**, and a size of **92 B**. The **Body** tab of the response is selected, showing the JSON data.

Resultado de 3 envios:

C:\Users\user\Desktop\CSharp\FilmesAPI\bin\Debug\net5.0\FilmesAPI.exe

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\user\Desktop\CSharp\FilmesAPI
0 senhor dos aneis
0 senhor dos aneis
0 senhor dos aneis
```

VALIDANDO INFORMAÇÕES ENVIADAS VIA HTTPPOST PELO POSTMAN

Os campos são autoexplicativos. As ErrorMessage aparecem no próprio console do Postman

```
public class Filme
{
    [Required(ErrorMessage = "O título do filme é obrigatório.")]
    0 referências
    public string Titulo { get; set; }
    [Required(ErrorMessage = "O nome do diretor do filme é obrigatório.")]
    0 referências
    public string Diretor { get; set; }
    [Required(ErrorMessage = "Este campo é obrigatório.")]
    [StringLength(30, ErrorMessage = "O campo gênero tem um limite de 30 caracteres.")]
    0 referências
    public string Genero { get; set; }
    [Required(ErrorMessage = "Este campo é obrigatório.")]
    [Range(1, 180, ErrorMessage = "O filme deve ter a duração mínima de 1 minuto e máxima de 180 minutos.")]
    0 referências
    public int Duracao { get; set; }
}
```

RETORNANDO TODAS AS INFORMAÇÕES DA API

1º Criar o método **HttpGet** para capturar os dados da nossa API via Postman

```
// mais dinâmico retornar IEnumerable, pois amplia nosso leque de opções
[HttpGet]
0 referências
public IEnumerable<Filme> RecuperarFilmes()
{
    return filmes;
}
```

2° Vamos criar um ID manual para os filmes, por enquanto. O fiz na classe Filme

```
0 referências
public Filme()
{
    this.Id = _contId++;
}
```

3º criar a requisição GET no Postman. Basta adicionar nova guia e colar o http://...

GET

http://localhost:5000/filme

Send

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body

Cookies

Headers (4)

Test Results

200 OK 5 ms 452 B Save Response

Pretty

Raw

Preview

Visualize

JSON

```
1 []
2 {
3   "id": 0,
4   "titulo": "O senhor dos aneis 2",
5   "diretor": "Peter Jackson",
6   "genero": "Aventura",
7   "duracao": 100
8 },
9 {
```

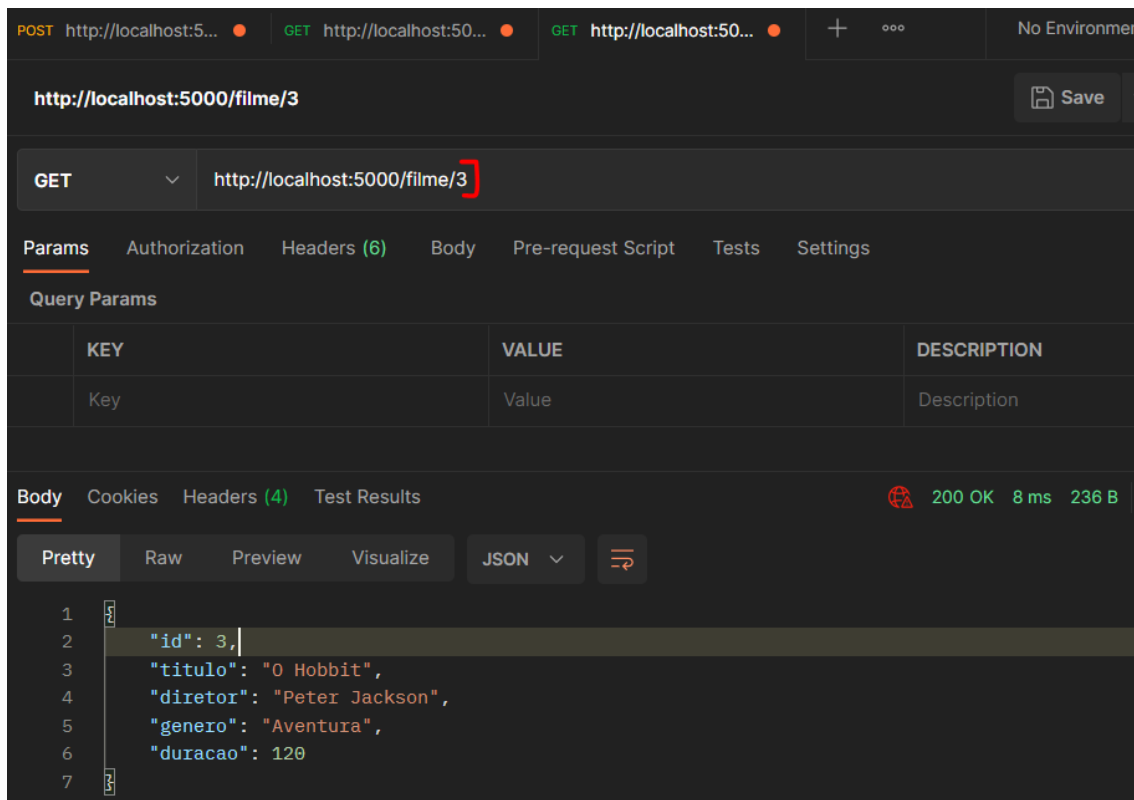
RETORNANDO INFORMAÇÕES DA API COM BASE EM 'Id' DO FILME

1° Vamos criar o método que recebe o Id e realizar a busca.

Note que ele recebe o parâmetro “{id}” em `HttpGet`.

```
[HttpGet("{id}")] // o id é passado pela url http://... do Postman para realizar a busca
0 referências
public Filme RecuperaFilmesPorId(int id)
{
    return filmes.FirstOrDefault(f => f.Id == id); //retorna o default em caso de não haver..
                                                //..na lista. Nesse caso, objetos, é null
}
```

2° Adicionar nova aba GET no Postman e testar a URL com o Id a ser buscado.



DEFININDO RETORNO DAS ROTAS/REQUISIÇÕES COM BASE NO PADRÃO REST

Vamos modificar os returns dos métodos, para que retornem códigos (404, 200, 201, etc) em todas as requisições, além de informações mais sólidas.

1° Modificar a Action/Método 'AdicionaFilme':

Note que todos os retornos foram mudados para **ActionResult**, por causa dos 'returns' específicos da REST API, que exigem isso.

```
// palavra reservada do método Rest, que significa CRIAR alguma coisa
[HttpPost] // estou dizendo que a inf enviada pelo Programa Postman vai postar a info no server da minha API
public IActionResult AdicionaFilme([FromBody] Filme filme) // recebe a inf enviada pelo postman pelo seu "Body"
{
    filmes.Add(filme);
    return CreatedAtAction(nameof(RecuperaFilmesPorId), new { Id = filme.Id }, filme); //Especifica qual foi a action..
    //..que criou esse recurso: a primeira action recupera o recurso que foi adicionado pelo seu Id, que é..
    //..instanciado em 'new {Id = ...}'. Por fim, o último parâmetro é o objeto adicionado em si.
}
```

Agora é possível ver o caminho de criação da requisição **POST**:

The screenshot shows the Postman interface for a POST request to `http://localhost:5000/filme`. The response is a 201 Created status with a response time of 4 ms and a body size of 285 B. The 'Headers' tab is selected, showing the following headers:

KEY	VALUE
Date	Thu, 05 Aug 2021 21:51:04 GMT
Content-Type	application/json; charset=utf-8
Server	Kestrel
Transfer-Encoding	chunked
Location	https://localhost:5001/Filme/2

2° Modificar a Action 'RecuperarFilmes':

```
[HttpGet]
0 referências
public IActionResult RecuperarFilmes()
{
    return Ok(filmes);
}
```

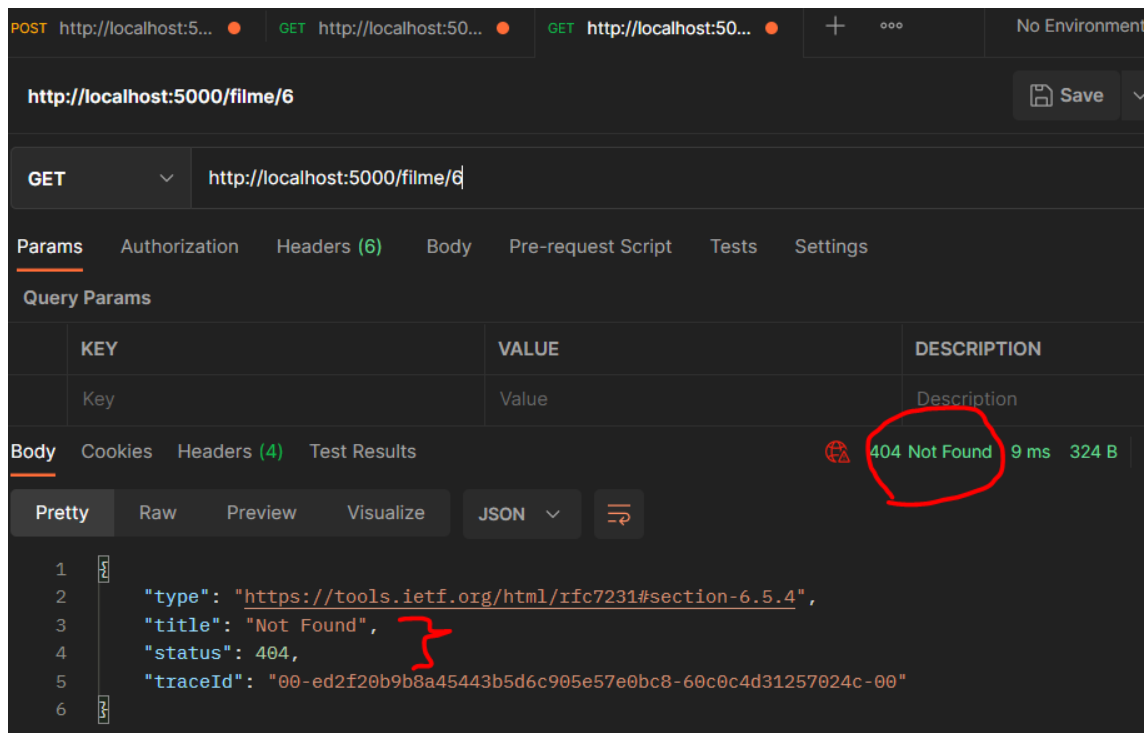
3° Modificar a Action 'RecuperarFilmePorId':

```
[HttpGet("{id}")] // o id é passado pela url http://... do Postman para realizar a busca
1 referência
public IActionResult RecuperaFilmesPorId(int id)
{
    var filme = filmes.FirstOrDefault(f => f.Id == id);
    if (filme != null)
    {
        return Ok(filme); // padrão de retorno quando dá certo a requisição GET. também retorna o objeto pesquisado
    }
    return NotFound(); //padrão em caso do objeto não ser encontrado.
}
```

Observe os 'returns' do método em ação no Postman, em caso de requisição ok e notFound:

The screenshot shows the Postman interface for a GET request to `http://localhost:5000/filme/2`. The status bar at the bottom indicates a successful response with `200 OK`, which is circled in red. The response body is displayed in JSON format:

```
1 {
2   "id": 2,
3   "titulo": "O Hobbit 2",
4   "diretor": "Peter Jackson",
5   "genero": "Aventura",
6   "duracao": 120
7 }
```

CONECTANDO A API AO BD MySQL

Primeiro é necessário baixar os frameworks necessários para rodar a app:

Em ferramentas > gerenciador do pac. Nuget > Gerenciar pacotes do Nuget

Em Procurar: **Microsoft.EntityFrameworkCore** (obrigatório)

Microsoft.EntityFrameworkCore.Tools (obrigatório)

MySQL.EntityFrameworkCore (para permitir conexão com o MySQL)

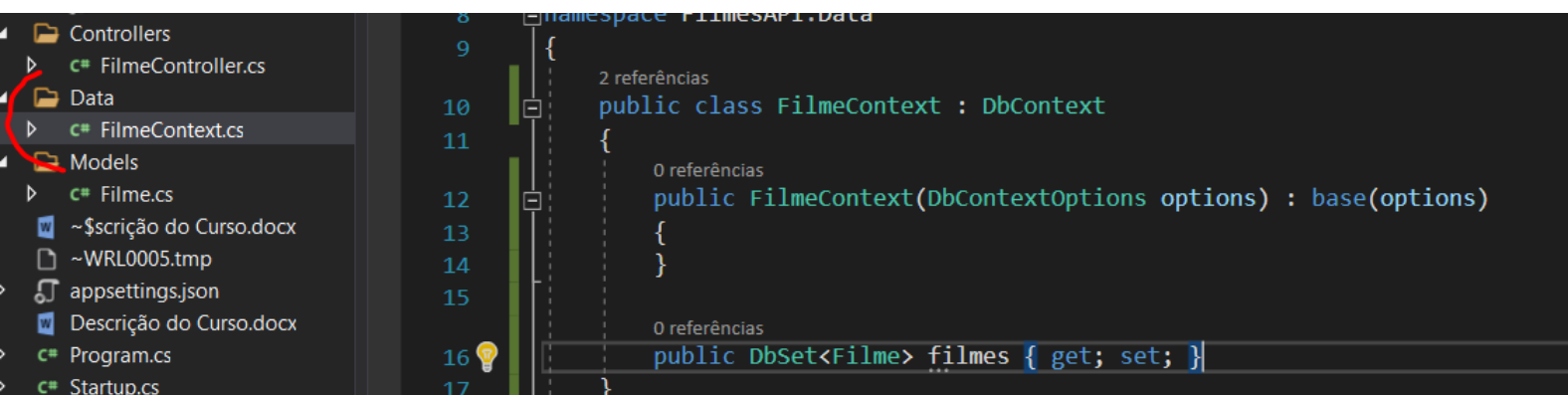
Instalar um por um. No caso de SQL, seria necessário instalar o

Microsoft.EntityFrameworkCore.SqlServer

Vamos conectar agora:

1° Criar uma pasta no projeto (nome pode ser Data), criar a class **Context**, a qual vai estabelecer a conexão com **DbContext**

Construtor obrigatório. Basta usar Ctrl + . na classe e gerar automaticamente.



2° Criar a **StringConnection** em **appsettings.json**:

```
"ConnectionStrings": {  
  "FilmeConnection": "server=localhost;database=filmeDb;user=root;password=passBDmysql"  
}
```

O atributo que recebe a string de conexão pode ter o nome que quisermos. O padrão e o mais simples é sempre: **local do BD; Nome do BD; Nome do USER; Senha do BD.**

3° Criar o Service que vai realizar a conexão na classe **Startup.cs** e realizar a conexão via código:

com SQL server seria **"UseSqlServer(ConnectionString)"**

```
public void ConfigureServices(IServiceCollection services)  
{  
    var connectionString = Configuration.GetConnectionString("FilmeConnection");  
    services.AddDbContext<FilmeContext>(option =>  
        option.UseMySQL(connectionString) //estabelece a conexão com MySQL, da mesma forma que em SQL  
    );  
}
```

GERANDO A 1ª MIGRATION

1° No console do gerenciador de pacotes NuGet: **Add-Migration 'DarNomeAEla'** >

chegar se está tudo correto > **Update-Database**

2° Entrar no MySQL e checar se o BD e suas tabelas foram criadas:

Query 1

```

1  show databases;
2
3  • use filmedb; -- diz para usa esse BD --
4
5  • show tables;
6
7  • desc filmes; -- pede para descrever as colunas da tabela --

```

Result Grid

	Field	Type	Null	Key	Default	Extra
►	Id	int(11)	NO	PRI	NULL	auto_increment
	Título	text	NO		NULL	
	Diretor	text	NO		NULL	
	Genero	varchar(30)	NO		NULL	
	Duracao	int(11)	NO		NULL	

ESCREVENDO E LENDO DADOS NO BD

1º Vamos acessar a tabela de filmes pela instanciação da classe **FilmeContext** em **FilmeController**

```

public class FilmeController : ControllerBase
{
    private FilmeContext _context;

    0 referências
    public FilmeController(FilmeContext context)
    {
        this._context = context;
    }
}

```

2º Agora devemos realizar as substituições da **List<Filme>** que estava sendo usada nos métodos/actions pelo campo de acesso ao BD.

```

[HttpPost] // estou dizendo que a inf enviada pelo Programa Postman vai postar a info no server da minha API
0 referências
public IActionResult AdicionaFilme([FromBody] Filme filme) // recebe a inf enviada pelo postman pelo seu "Body"
{
    _context.Filmes.Add(filme);
    _context.SaveChanges();
    return CreatedAtAction(nameof(RecuperaFilmesPorId), new { Id = filme.Id }, filme); //Especifica qual foi a action..
    //..que criou esse recurso: a primeira action recupera o recurso que foi adicionado pelo seu Id, que é..
    //..instanciado em 'new {Id = ...}'. Por fim, o último parâmetro é o objeto adicionado em si.
}

```

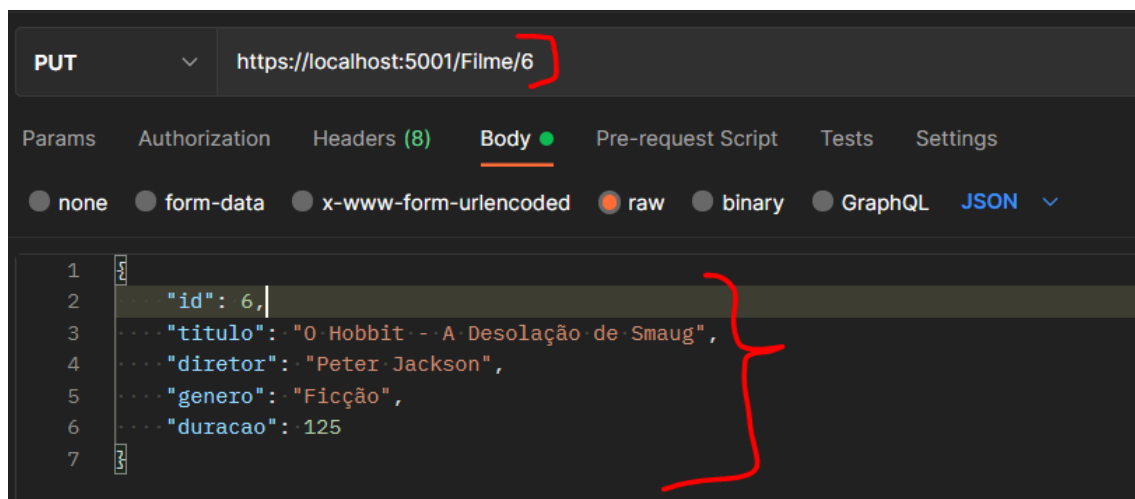
```
[HttpGet]
0 referências
public IEnumerable<Filme> RecuperarFilmes() //podemos voltar ao retorno IEnumerable. Aqui o IAction não é necessário
{
    return _context.Filmes.ToList();
}
```

ATUALIZANDO E DELETANDO DADOS NO BD

```
[HttpPut("{id}")] // "PUT" é a requisição utilizada para atualizar
0 referências
public IActionResult AtualizaFilme(int id, [FromBody] Filme filmeNovo) //estamos recebendo como..
{ //..parâmetros o id para achar o objeto e o novo objeto
    var filme = _context.Filmes.Where(f => f.Id == id).FirstOrDefault();
    if (filme == null)
    {
        return NotFound();
    }
    filme.Titulo = filmeNovo.Titulo;
    filme.Diretor = filmeNovo.Diretor;
    filme.Genero = filmeNovo.Genero;
    filme.Duracao = filmeNovo.Duracao;
    _context.SaveChanges();
    return NoContent(); //boa prática retornar 'semConteúdo' para updates
}
```

Veja no Postman como a requisição deve ser feita:

Colocamos o ID do objeto a ser mudado na URL e os novos dados.



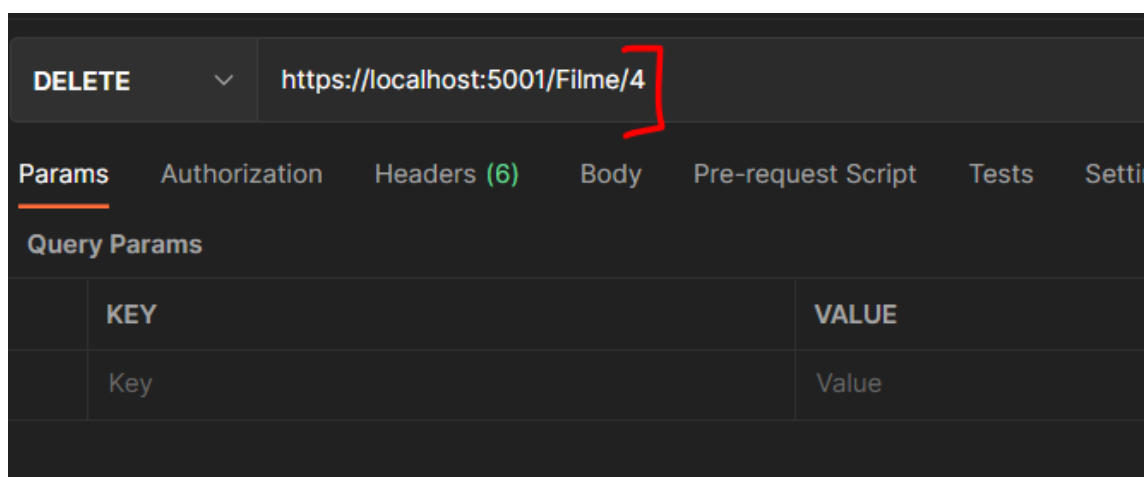
```

[HttpDelete("{id}")] //requisição para deletar por ID
0 referências
public IActionResult ExcluirFilme(int id)
{
    var filme = _context.Filmes.Where(f => f.Id == id).FirstOrDefault();
    if (filme == null)
    {
        return NotFound();
    }
    _context.Filmes.Remove(filme);
    _context.SaveChanges();
    return NoContent(); //boa prática retornar 'semConteúdo' para deletes
}

```

Veja como enviar a requisição DELETE pelo Postman:

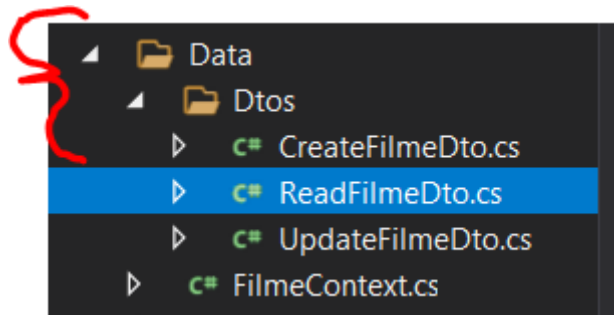
Colocar o ID do objeto a ser deletado na URL.



MELHORANDO O CÓDIGO COM DTOS

Vamos usar **Data Transfer Object (DTO)**, que é um padrão de projeto que visa criar classes que vão instanciar os objetos que transferem dados entre as partes do nosso projeto.

1º Criar o diretório **Dtos** dentro de **Data**. Cada tipo de operação vai iniciar o nome da classe DTO, como criar objetos no BD: **"Create"+Nome da Classe+"Dto"**



2º na classe DTO, passar apenas os atributos que poderão ser acessados:

Note que tiramos a prop ID, pois nessa requisição POST ele não deve ser acessado pelo usuário.

```
public class CreateFilmeDto
{
    public string Titulo { get; set; }
    [Required(ErrorMessage = "O nome do diretor do filme é obrigatório.")]
    public string Diretor { get; set; }
    [Required(ErrorMessage = "Este campo é obrigatório.")]
    [StringLength(30, ErrorMessage = "O campo gênero tem um limite de 30 caracteres.")]
    public string Genero { get; set; }
    [Required(ErrorMessage = "Este campo é obrigatório.")]
    [Range(1, 180, ErrorMessage = "O filme deve ter a duração mínima de 1 minuto e máxima de 180 minutos.")]
    public int Duracao { get; set; }
}
```

3º Trocar o tipo do objeto no respectivo método/Action:

```
// palavra reservada do método Rest, que significa CRIAR alguma coisa
[HttpPost] // estou dizendo que a inf enviada pelo Programa Postman vai postar a info no server da minha API
public IActionResult AdicionaFilme([FromBody] CreateFilmeDto filmeDto)
{
    Filme filme = new Filme //o objetivo da classe Dto é limitar e controlar os dados que podem ser enviados..
    { //..pelo usuário via requisição POST
        Diretor = filmeDto.Diretor,
        Genero = filmeDto.Genero,
        Titulo = filmeDto.Titulo,
        Duracao = filmeDto.Duracao
    };
    _context.Filmes.Add(filme);
    _context.SaveChanges();
    return CreatedAtAction(nameof(RecuperaFilmesPorId), new { Id = filme.Id }, filme);
}
```

4º Criar os acessos DTO restantes:

Vamos criar um acesso para retornar mais dados do que a classe tem de propriedades:

Note a eficiência desse padrão de projeto.

```
2 referências
public class ReadFilmeDto
{
    [Key]
    [Required]
    1 referência
    public int Id { get; set; }
    1 referência
    public string Titulo { get; set; }
    [Required(ErrorMessage = "O nome do diretor do filme é obrigatório.")]
    1 referência
    public string Diretor { get; set; }
    [Required(ErrorMessage = "Este campo é obrigatório.")]
    [StringLength(30, ErrorMessage = "O campo gênero tem um limite de 30 caracteres.")]
    1 referência
    public string Genero { get; set; }
    [Required(ErrorMessage = "Este campo é obrigatório.")]
    [Range(1, 180, ErrorMessage = "O filme deve ter a duração mínima de 1 minuto e máxima de 180 minutos.")]
    1 referência
    public int Duracao { get; set; }
    1 referência
    public DateTime DataConsulta { get; set; } //vamos retornar a data da consulta a usuário.
}
```

Veja como fica a mudança no método:

```
[HttpGet("{id}")] // o id é passado pela url http://... do Postman para realizar a busca
1 referência
public IActionResult RecuperaFilmesPorId(int id)
{
    var filme = _context.Filmes.Where(f => f.Id == id).FirstOrDefault();
    if (filme != null)
    {
        ReadFilmeDto filmeDto = new ReadFilmeDto //agora estamos usando o padrão DTO para retornar..
        { //..informações extras sem mexer na classe principal, Filme.
            Titulo = filme.Titulo,
            Genero = filme.Genero,
            Id = filme.Id,
            Diretor = filme.Diretor,
            Duracao = filme.Duracao,
            DataConsulta = DateTime.Now
        };
        return Ok(filmeDto); // padrão de retorno quando dá certo a requisição GET. também retorna o objeto pesquisado
    }
    return NotFound(); //padrão em caso do objeto não ser encontrado.
}
```

MELHORANDO O CÓDIGO COM AUTOMAPPER

É um pacote que é utilizado para auxiliar na **conversão de tipos dados**, evitando a conversão manual de propriedade por propriedade, o que aumenta o número de linhas do código.

1° Baixar o framework **AutoMapper.Extensions.Microsoft.DependencyInjection** pelo NuGet.

2° Adicionar o service na classe Startup:

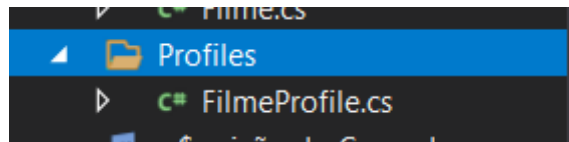
```
services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies()); // estamos dizendo que vamos..  
//..utilizar o domínio da nossa aplicação com o AutoMapper.
```

3° Configurar quais as classes/tipos de objetos que deverão ser implicitamente convertidos pelo AutoMapper. São os chamados Perfis/Profiles.

4° Criar o diretório Profiles no projeto, onde serão armazenados os tipos de conversões entre classes.

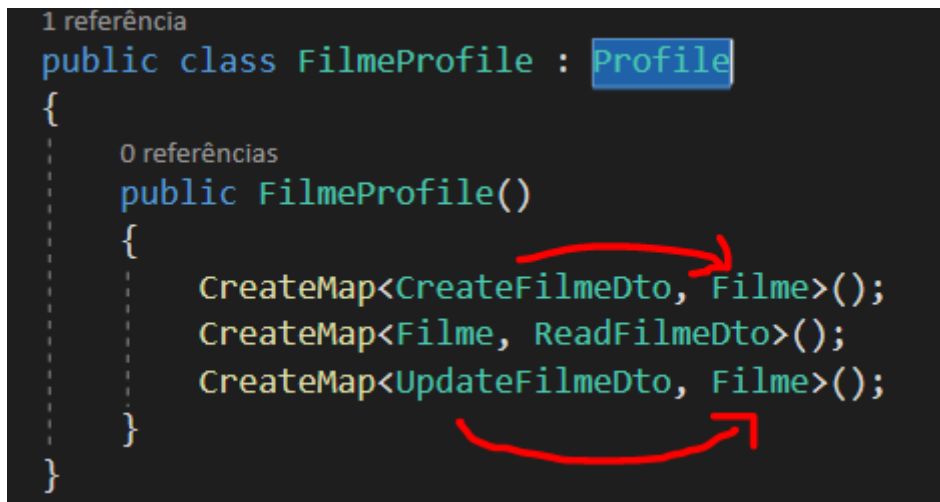
As conversões serão entre objetos de tipo Filme, somente.

5° Criar a classe Nome+"Profile".



Note que devemos adicionar quais tipos serão convertidos para quais:

```
1 referência  
public class FilmeProfile : Profile  
{  
    0 referências  
    public FilmeProfile()  
    {  
        CreateMap<CreateFilmeDto, Filme>();  
        CreateMap<Filme, ReadFilmeDto>();  
        CreateMap<UpdateFilmeDto, Filme>();  
    }  
}
```



Agora vamos usar a nova classe criada, instanciando uma propriedade do tipo **IMapper** na classe que usará as conversões:

```
public class FilmeController : ControllerBase
{
    private FilmeContext _context;
    private IMapper _mapper;

    //O referências
    public FilmeController(FilmeContext context, IMapper mapper) //preenchido por injeção de dependência
    {
        this._context = context;
        this._mapper = mapper;
    }
}
```

Hora de realizar as conversões. Usaremos dois tipos:

1° Converter de um tipo para o outro:

```
// palavra reservada do método Rest, que significa CRIAR alguma coisa
[HttpPost] // estou dizendo que a inf enviada pelo Programa Postman vai postar a info no server da minha API
//O referências
public IActionResult AdicionaFilme([FromBody] CreateFilmeDto filmeDto)
{
    Filme filme = _mapper.Map<Filme>(filmeDto); //estou convertendo filmeDto para a classe Filme e..
    //..armazenando na variável filme.
    _context.Filmes.Add(filme);
    _context.SaveChanges();
    return CreatedAtAction(nameof(RecuperaFilmesPorId), new { Id = filme.Id }, filme);
}
```

2° Jogar os dados de uma variável de um tipo para uma variável de outro.

```
[HttpPut("{id}")] //"PUT" é a requisição utilizada para atualizar
//O referências
public IActionResult AtualizaFilme(int id, [FromBody] UpdateFilmeDto filmeDto) //estamos recenbendo como..
{ //..parâmetros o id para achar o objeto e o novo objeto
    Filme filme = _context.Filmes.Where(f => f.Id == id).FirstOrDefault();
    if (filme == null)
    {
        return NotFound();
    }
    _mapper.Map(filmeDto, filme); //dessa vez estamos jogando as informações de filmeDto para filme e..
    //..salvando no BD um objeto do tipo Filme.
    _context.SaveChanges();
    return NoContent(); //boa prática retornar 'semConteúdo' para updates
}
```

Pronto. Está tudo mais legível e com boas práticas de desenvolvimento e funcionando perfeitamente.

FIM DO CURSO