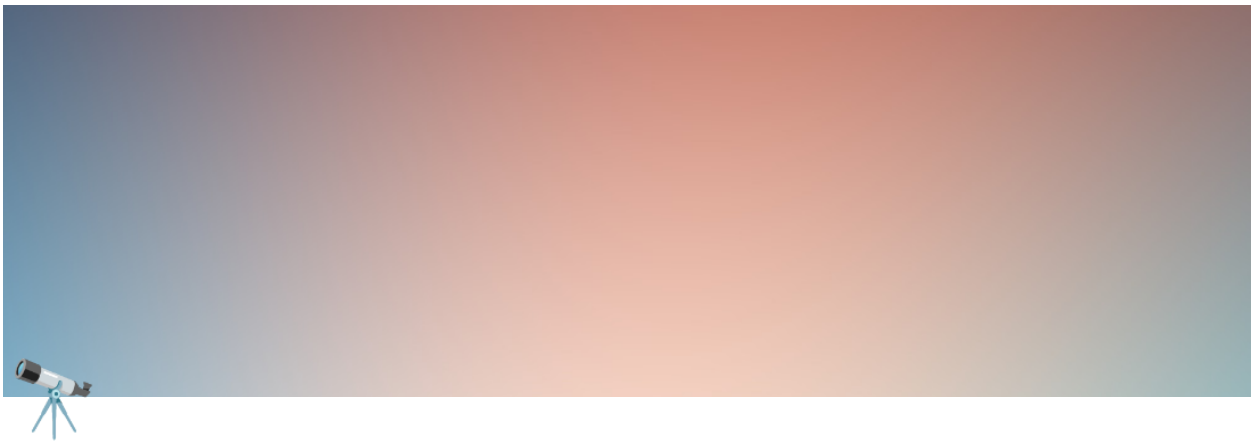

API-REST

Passo a passo do módulo 3 do curso de API's rest do professor Macoratti

Ver meu projeto no github:

> https://github.com/MatheusdeOliveiraCorrea/APIRest-CursoMacoratti

1. **Criar um projeto do tipo web api c#**
2. **Excluir entidades desnecessárias e criar uma pasta Models com as classes Categoria.cs e Produtos.cs**
3. **No nuGet instalar os pacotes:**
    a. Pomelo.EntityFrameworkCore.MySql
    b. Microsoft.EntityFrameworkCore.Design
4. **Instalar a ferramenta Entity Framework Core .Net command-line Tools. Primeiramente, verifique se já não está instalada com o comando:**
    a. **dotnet ef**
5. **Após este comando deve aparecer uma mensagem da ferramenta, caso contrário, siga a instalação:**
    a. dotnet tool install --global dotnet-ef
    b. dotnet tool update --global dotnet-ef

O que é a classe de contexto?

É uma classe muito importante, herda do DbContext (do EntityFrameworkCore) essa classe que define o **mapeamento entre as entidades e as tabelas**

DbContext- Representa uma sessão com o banco de dados, é a ponte entre as entidades e o banco

DbSet<T>- Representa uma coleção de entidades no contexto que podem ser consultadas no bando de dados

6. **Criar uma classe de contexto chamada APICatalogoContext.cs na pasta Context**
7. **Definir a string de conexão no arquivo appsettings.json:**

```
{
  "ConnectionStrings": {
    "ConexaoPadrao": "stringconexão"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

8. **Pegar a string de conexão na classe Program**

```
//linha 1
string stringDeConexaoBD = builder.Configuration.GetConnectionString("ConexaoPadrao");

//linha 2
builder.Services.AddDbContext<APICatalogoContext>
        (options => options.UseMySql(stringDeConexaoBD, ServerVersion.AutoDetect(stringDeConexaoBD)));
```

9. **Antes de fazer o migrations nós precisamos definir o relacionamento entre as tabelas** **Produtos** e **Categorias**. Uma categoria vai ter uma coleção de produtos e um produto vai ter somente uma categoria.
10. **Para fazer isso, definimos na classe Categoria a propriedade:**

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

E na classe **Produto**, as propriedades:

```
public int CategoriaId { get; set; }
public Categoria? Categoria { get; set; }
```

11. **======== Fazendo o Migrations ========**

O que é o migrations no EF core ?

É uma maneira de atualizar as tabelas do banco de dados para mantê-lo sempre em sincronia com o modelo de dados do aplicativo.

IMPORTANTÍSSIMO: Sempre que você alterar as classes do modelo domínio, precisará executar o Migrations para manter o banco de dados atualizado.

Comandos do EF Core Tools:

- **dotnet ef:**

  Verifica se o EF Core Tools está instalado (se não estiver, volte ao passo de número 5)

- **dotnet ef migrations add 'nome':**

  Sendo 'nome' uma variável a escolha do usuário, que pode ser declarada com as aspas simples ou não, representa o nome do script de migração (com os métodos Up() e Down())

- **dotnet ef migrations remove 'nome':**

  Remove o script criado pelo comando migrations add, se nescessário ..

- **dotnet ef database update**

  Por fim, aplica as mudanças que você configurou.

12. **Passo 12, saber o que é DataAnnotations**

DataAnnotations, Para que servem ?

As convenções do **Entity Framework** core ditam que deve-se usar o maior valor possível para guardar uma coluna no banco de dados se o tamanho não estiver especificado, como por exemplo, uma entidade definida com o tipo string vai ser guardada no banco de dados como longtext que é um valor muito maior do que o que precisamos.

Para solucionar este problema, podemos sobrescrever essas convenções podemos usar DataAnnotations para:

- Definir regras de validação para o modelo
- Definir como os dados devem ser exibidos na interface em aplicações ASP.NET CORE MVC
- Especificar o relacionamento entre as entidades do modelo
- **Sobrescrever as convenções padrão do Entity Framework Core**

Estes atributos estão disponíveis nos namespaces:

- System.ComponentModel.DataAnnotations
- System.ComponentModel.DataAnnotations.Schema

Após definir os valores para o tamanho da string e dos decimal, faça uma nova migration e execute-a.

13. **Popular as Tabelas do banco de dados**

Criar uma migração vazia usando instruções de insert into para incluir dados nas tabelas

Começaremos pela tabela **Categorias**
->mb.Sql("Insert into Categorias(Nome, ImagemUrl) values('Bebidas', 'bebidas.jpg')");
E um código semelhante para a tabela Produtos ...

14. Criar um Controller para Produtos (vazio)

Na pasta controller, criar um arquivo do tipo controller (que herde a classe ControllerBase e tem uma rota e o atributo ApiController.

Em seguida vamos usar Injeção de Dependência para a classe de contexto (classe que faz o mapeamento do bando de dados) pois precisaremos acessar o banco de dados.

```
using APICatalogo.Context;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

//namespace sem chaves, .NET6
namespace APICatalogo.Controllers;

[Route("[controller]")] //atributo que define a rota
[ApiController] //atributo que define uma controller
public class ProdutosController : ControllerBase
{
    private readonly APICatalogoContext _context;

    public ProdutosController(APICatalogoContext context)
    {
        _context = context;
    }
}
```

15. Criar método Get para ProdutoController

Vamos começar vendo diretamente o código, em seguida seus porquês:

```
[HttpGet]
public ActionResult<IEnumerable<Produto>> Get()
{
    var produtos = _context.Produtos.ToList();
    if (produtos is null) return NotFound("Produtos não encontrados");
    return produtos;
}
```

A coisa mais interessante a se notar é o tipo do retorno do método, observe a última linha return que está retornando uma lista e a linha do meio return que está retornando NotFound(string).

Então o método está retornando dois tipos de coisas(objetos, formalmente) diferentes. Isso é possível pq o ActionResult definido na linha da declaração do método está aceitando retornar também um IEnumerable.

Além disso, há uma verificação para ver se a lista de produtos não está null, caso esteja null, retorna um ActionResult NotFound(”Produtos não encontrados”). Por fim, o atributo [HttpGet] informa qual o verbo HTTP o método responde.

16. Criar método para pegar pelo ID:

```
[HttpGet("{id:int}")]
    public ActionResult<Produto> Get(int id)
    {
        var produto = _context.Produtos.FirstOrDefault(p => p.ProdutoId == id);
        if (produto is null) return NotFound("Produto com ID: " + id + " não encontrado, por favor, tente outro ID.");
```

```
        return produto;
    }
```

Nada muito diferente do método acima, desta vez a localização do produto está sendo feito por uma expressão lambda (p ⇒ p.ProdutoId == id) ou de forma verbal “Ache o número que foi passado no parametro do método que seja IGUAL ao número que foi definido na coluna ProdutoId no banco de dados “

17. Criar um método para criar um novo produto

    Para fazer o código desse método foi preciso alterar o método anterior. Cronologicamente não foi foi a primeira coisa que eu fiz, então vou mostrar como escrevi o código passo a passo para a explicação ficar clara e bem definida

    Começaremos pelo básico

```
    [HttpPost]
    public ActionResult Post(Produto produto)
    {
        if (produto is null) return BadRequest();
    }
```

A declaração do método como HttpPost (ou seja, irá criar/ adicionar alguma coisa) que retorna até agora somente um BadRequest() caso produto seja nulo. Mas de onde vem esse produto? Esse produto vem do body da requisição, em versões mais antigas era necessário especificar isso na declaração do método com [FromBody]:

```
    [HttpPost]
    public ActionResult Post([FromBody] Produto produto){return BadRequest();}
```

Entretanto não é mais necessário nas versões mais recentes, não vamos utilizar.

O outro passo é diretamente do context na tabela Produtos adicionar na memória e persistir no banco de dados o que foi recebido no parametro (body) :

```
[HttpPost]
    public ActionResult Post(Produto produto)
    {
        if (produto is null) return BadRequest();
        _context.Produtos.Add(produto);
        _context.SaveChanges();
    }
```

Importante notar que sem o método SaveChanges(); as mudanças são apenas locais em memória e **não serão salvas no banco de dados**, então é bastante importante não esquecer dele.

E por fim, vamos adicionar uma linha que retorna um CreatedAtRouteResult();

```
[HttpPost]
    public ActionResult Post(Produto produto)
    {
        if (produto is null) return BadRequest();
```

```
        _context.Produtos.Add(produto);
        _context.SaveChanges();
        return new CreatedAtRouteResult("ObterProduto",
            new { id = produto.ProdutoId }, produto);
    }
```

Para entendermos melhor, vamos dar um nome ao método do passo anterior:

```
[HttpGet("{id:int}", Name = "ObterProduto")]
    public ActionResult<Produto> Get(int id)
    {
        var produto = _context.Produtos.FirstOrDefault(p => p.ProdutoId == id);
        if (produto is null) return NotFound("Produto com ID: " + id + " não encontrado, por favor, tente outro ID.");
        return produto;
    }
```

Observe o atributo Name na primeira linha. Foi a única modificação feita neste método.

De volta ao código do método que estamos trabalhando, uma visão mais de perto:

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

Como parametro deste construtor estamos passando o nome que definimos ao método que obtém o produto pelo ID, em seguida passamos o id e o produto. Então essa linha vai chamar o método “ObterProduto” que é o método de obter o id e pegar o id que está sendo passado dentro do new.