Criação de uma API WEB utilizando C# com .NET, EF Core 7.0 e SQL Server:

Antes de iniciar, verifique se o dotnet está instalado no computador. Para isso, abra o terminal e passe o seguinte comando:

```
dotnet --version
```

- Caso o terminal não retorne um valor ou retorne uma versão diferente da 7 (a ser utilizada nesta atividade), instale o dotnet 7 pelo site da Microsoft.

Com o dotnet 7 instalado, crie o projeto através do terminal, no local onde desejar, utilizando o seguinte comando:

```
dotnet new web -o <Nome da Aplicação> -f net7.0
```

- dotnet = Informa que será através dele que será criada a aplicação.
- new = Inicializa uma nova aplicação.
- web = Informa que será uma aplicação voltada para a web (API).
- -o = Informa que o próximo parâmetro será o nome da aplicação.

- <Nome da Aplicação>, no exemplo utilizaremos Customer**API**.
- -f = Informa qual será a versão do dotnet utilizada.

Entre na pasta principal do projeto:

```
cd <Nome da Aplicação>
```

Instale os pacotes necessários para o desenvolvimento da API:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet add package Microsoft.EntityFrameworkCore --version 7.
```

Abra o projeto:

```
start <Nome da Aplicação>.csproj
```

Agora, com o projeto aberto, adicione uma pasta com o nome **Models**. É nesta pasta onde serão criados os modelos das tabelas que você deseja trabalhar.

Dentro desta pasta, crie uma classe com o nome da tabela que você deseja criar, no caso, será Customer**.cs**.

Adicione as colunas desejadas, já dando a elas o seu tipo e nome. O resultado final de uma tabela fictícia e simples de Customers ficará da seguinte forma:

```
namespace CustomerAPI.Models
{
    public class Customer
    {
        public int Id { get; set; }
        public string FullName { get; set; }
        public string Email { get; set; }
        public string Telephone { get; set; }
```

```
        public string Cpf { get; set; }
    }
}Para trabalhar com mais tabelas, apenas continue criando nov
```

Agora, crie uma nova pasta, chamada Data e, dentro dela, crie uma classe chamada **AppDbContext.cs**. Está classe servirá para representar o banco de dados em memória, ou seja, diremos que a classe X de nosso projeto representa a tabela X no banco de dados, criando assim uma ponte entre esses ambientes.

A primeira coisa que deve ser feira é herdar DbContext do pacote do EF Core instalado anteriormente. Isso é feito adicionando < **: DbContext>** após o nome da classe.

Após adicionada a herança, será feito o mapeamento em si, adicionando as tabelas da seguinte forma:

```
public DbSet<Customer> Customer { get; set; }
```

Em resumo, você está dizendo para o EF core que há uma tabela chamada Customers em seu banco de dados e esta será utilizada/modificada através de seu projeto.

Após isso, instanciaremos o método *OnConfiguring()*, pelo qual nos conectaremos ao banco de dados em si, de forma simplificada.

Como parâmetro, passamos *DbContextOptionsBuilder optionsBuilder* e, dentro do método, *optionsBuilder.UseSqlServer(connectionString:"<string para conexão com seu banco de dados>")*

- A *"conection string"*, tanto para conexão com sql server, quanto para outras plataformas de dados, pode ser encontrada no site: https://www.connectionstrings.com/sql-server/
- Não esqueça de adicionar os *"using"* necessários no topo da classe, um para utilizar o EF Core e outro para para referenciar a classe CustomersAPI criada anteriormente.

O resultado final será este:

```
using Microsoft.EntityFrameworkCore;
using CustomerAPI.Models;
```

```
namespace CustomerAPI.Data
{
    public class AppDbContext: DbContext
    {
        public DbSet<Customer> Customer { get; set; }

        protected override void OnConfiguring(DbContextOption
            => optionsBuilder.UseSqlServer(connectionString:
    }
}
```

- Os dados a serem passados para a *“conection string”* podem ser encontrados em qualquer ambiente integrado para o gerenciamento de qualquer infraestrutura de SQL. Um exemplo é o SSMS - SQL Server Management Studio.
- Caso ainda não haja um banco criado com o nome que for passado em *“Database=”* em sua *“conection string”*, ele será criado automaticamente, através das *“Migrations”*, que serão abordadas em sequência. Caso contrário, utilizará o banco existente. Neste momento, criaremos um chamado **CustomerAPI**.

Após esta configuração inicial, rode o seguinte comando no terminal, para criar um arquivo de *“Migrations”*:

```
dotnet ef migrations add <Nome da migration>
```

- **Migration** é, em resumo, um arquivo gerado pelo dotnet em conjunto com o EF Core, de forma inteligente e prática, para manipulação do banco de dados do projeto. Assim, caso você altere os arquivos relacionados ao banco de dados, basta gerar uma nova *migration* e seu banco será atualizado de forma automática. Assim, economiza-se tempo e cria-se um histórico de alterações feitas no banco.

Após criado o arquivo, atualize seu banco de dados com ele, através do seguinte comando:

```
dotnet ef database update
```

- Agora, através de um gerenciador como o SSMS, confira se o banco e sua respectiva tabela foram criados. Caso negativo, resolva os problemas necessários antes de seguir.

Vá até a classe Program.cs e adicione algumas informações, para que fique da seguinte forma:

```
using CustomerAPI.Data;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddDbContext<AppDbContext>();

var app = builder.Build();

app.MapControllers();

app.Run();
```

- A primeira *“using CustomerAPI.Data”* importa o *namespace* relativo a própria aplicação, para que o Program.cs possa trabalhar com o projeto relativo a ele.
- Em seguida criamos uma variável que será um objeto chamado *builder* e atribuímos a ela o método *CreateBuilder* da classe *WebApplication*. Esse objeto criado é utilizado para configurar e inicializar a aplicação.
- A terceira linha adiciona serviços relacionados a métodos HTTP ao objeto anteriormente criado. Já a quarta linha adiciona o serviço de conexão criado pela classe *AppDbContext***.**
- Em seguida, a quinta linha cria uma variável app (que conterá as regras da aplicação), atribuindo a ela o builder estruturado anteriormente, passando ainda o método *Build()*, para construir/estruturar a aplicação.
- A sexta linha passa a informação de que será utilizado controllers na aplicação, devendo então serem mapeados.

- Por fim, a última linha inicia de fato a execução da API e aguarda as solicitações HTTP.

Agora, crie uma pasta chamada **Controllers** e, dentro dela, uma classe chamada **CustomerController.cs** (ou <NomeDaSuaTabela>Controller.cs).

- Este arquivo será responsável por seus endpoints, ou seja, será a parte principal da API, uma vez que são os endpoints que recebem, manipulam e retornam os dados do banco de dados.
- São, em resumo, o CRUD - Create, Read, Update e Delete (Post, Get, Put, Delete).

Após criado, herde "*Controller"* do EF Core em sua classe e adicione o respectivo using no topo do arquivo.

Defina então a classe como uma [ApiController] e uma adicione um atributo de prefixo de rota (que irá compor sua rota para os endpoints desta classe), adicionando estas informações logo antes da criação da classe. Após estas configurações iniciais, o código deverá ficar assim:

```
using Microsoft.AspNetCore.Mvc;

namespace CustomerAPI.Controllers
{
    [ApiController]
    [Route(template:"customer")]
    public class CustomerController : Controller
    {

    }
}
```

Agora, injetaremos a classe AppDbContext dentro da classe Controllers e, para isso, primeiramente adicione uma variável chamada *_context,* do tipo *AppDbContext*, como privada e apenas para leitura.

Abaixo, adicione um construtor da classe, recebendo *context* como parâmetro e atribua seu valor a variável recém criada.

Neste momento, o código deverá estar da seguinte maneira:

```
using CustomerAPI.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace CustomerAPI.Controllers
{
    [ApiController]
    [Route(template:"customer")]
    public class CustomerController : Controller
    {
        private readonly AppDbContext _context;
        public CustomerController(AppDbContext context)
        {
            _context = context;
        }
    }
}
```

Passaremos agora para a criação do primeiro e mais simples, o GET, que servirá para retornar todos os dados da tabela.

- Primeiramente, mas de forma opcional, deixe explicito o método que será utilizado, no caso, [HttpGet]. Após isso, crie o método Get, da seguinte forma:

```
using CustomerAPI.Data;
using Microsoft.AspNetCore.Mvc;

namespace CustomerAPI.Controllers
{
    [ApiController]
    [Route(template:"customer")]
    public class CustomerController : Controller
    {
        [HttpGet]
        public async Task<IActionResult> Get(
            [FromServices] AppDbContext context)
```

```
        {
            var todo = await context
                .Customer
                .AsNoTracking()
                .ToListAsync();

            return Ok(todo);
        }
    }
}
```

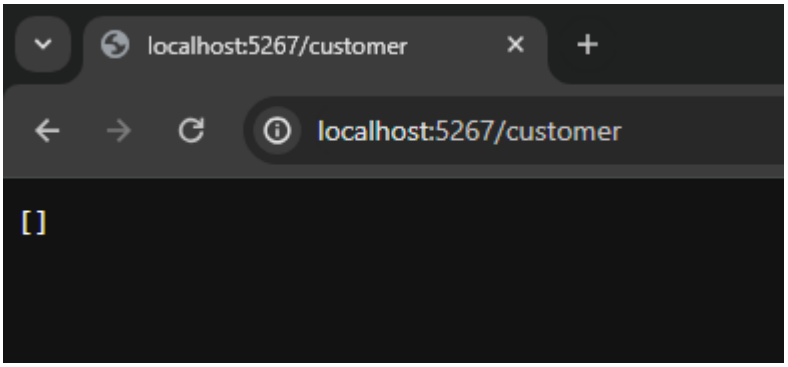
- Primeiramente, definimos que o método será publico e assíncrono, passamos então “*Task<IActionResult>”* que é uma representação de uma operação assíncrona que retorna um resultado de uma ação, no caso, o get. Por fim, definimos o nome do método como Get().
- Como parâmetro do método passamos *“[FromServices] AppDbContext context”* que indica que o sistema de injeção de dependências será fornecido pela classe AppDbContext.
- Após isso, criamos uma variável chamada *customers*, que aguarda receber do banco de dados toda a informação da tabela Customer.
- A respeito da chamada, iniciamos o pedido com *await*, para que a execução do código somente prossiga após receber o conteúdo deste pedido. Em seguida chamamos *context* para receber as informações do banco de dados, seguido de *.Customer*, ou seja, o nome da tabela a qual quero buscar as informações. Após isso, passamos *.AsNoTracking()* para interromper a busca após receber os valores e, por fim, *.ToListAsync()* para converter o resultado para uma lista.
- Encerramos o bloco retornando *Ok(customer)*, que retorna o status HTTP 200, contendo o conteúdo da tabela, em formado de lista JSON.

Tendo finalizado o primeiro endpoint (GET), já é possível realizar um teste da API. Para isso, inicialize a aplicação pelo terminal, através do seguinte comando:

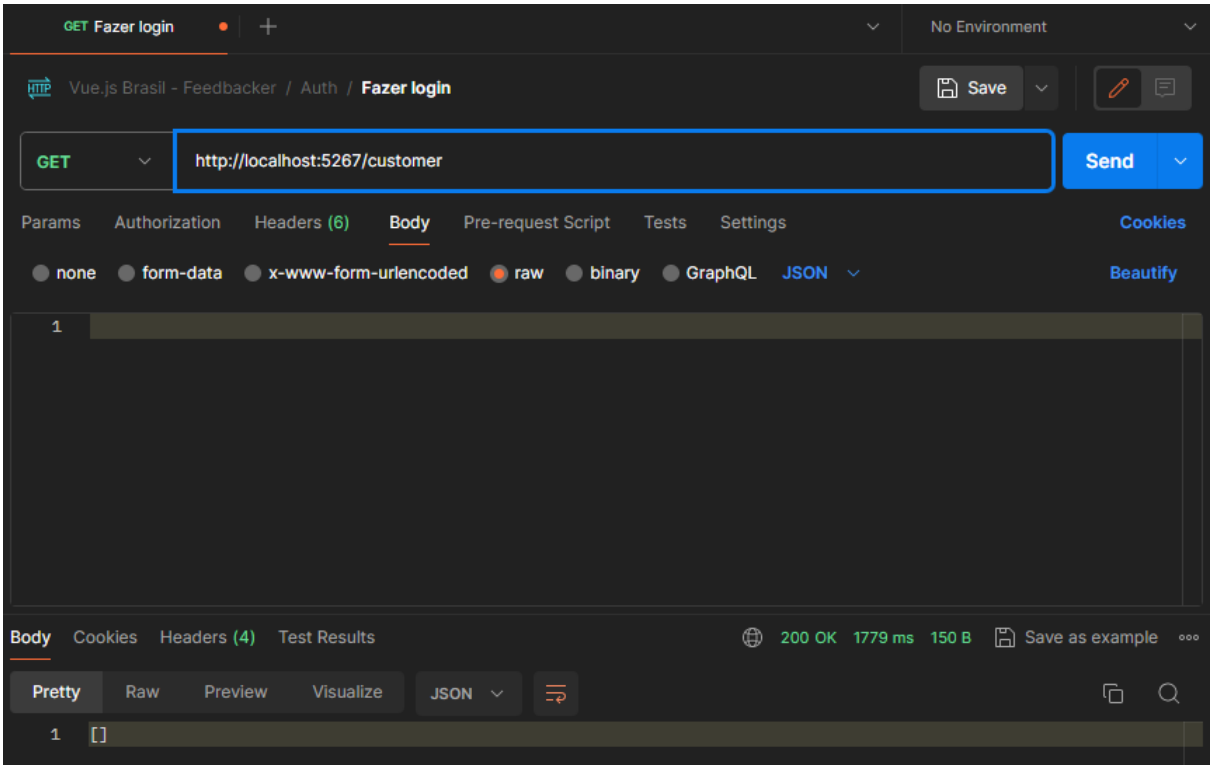
```
dotnet watch run
```

Seu navegador irá abrir na seguinte rota: *http://localhost:5267.*

Adicione /customer ao final dela (rota definida como padrão em sua classe *CustomerController*) e teste. O resultado esperado será algo similar a isso:



- Como o resultado da busca é uma lista de objetos, mas ainda não há nenhum dado na tabela, o retorno, por óbvio, será uma lista vazia.
- A partir deste momento, utilizaremos o Postman para realizar os testes, uma vez que posteriormente precisaremos adicionar dados no banco, mas sem um “frontend”. No Postman, a pesquisa e resultado deverão ficar da seguinte forma:



Agora, adicionaremos na classe CustomerController o código para buscar dados no banco de dados, através de um ID. Ainda será um método GET, mas agora

passando um parâmetro. O código deverá ficar da seguinte forma:

```
using CustomerAPI.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace CustomerAPI.Controllers
{
    [ApiController]
    [Route(template:"customer")]
    public class CustomerController : Controller
    {
        private readonly AppDbContext _context;
        public CustomerController(AppDbContext context)
        {
            _context = context;
        }

        [HttpGet]
        public async Task<IActionResult> Get()
        {
            var customer = await _context
                .Customer
                .AsNoTracking()
                .ToListAsync();

            return Ok(customer);
        }

        [HttpGet("{id}")]
        public async Task<IActionResult> GetById(
            [FromServices] AppDbContext context,
            [FromRoute] int id)
        {
            var customer = await context
                .Customer
                .AsNoTracking()
                .FirstOrDefaultAsync(x => x.Id == id);
```

```
            return customer == null ? NotFound() : Ok(custome
        }
    }
}
```

- A primeira mudança está já na definição do método, que agora passa a receber um parâmetro, que será o id. Esse parâmetro virá através da URL. Ou seja, agora a busca será feita através da mesma rota anterior + /id, ficando assim: http://localhost:5267/customer/1.
- O nome do método também muda, passando a especificar que será uma chamada através de um id: GetById.
- Como parâmetro, agora recebemos além do contexto, também [FromRoute] int id, que indica que será recebido pela rota (URL), um id do tipo int.
- Já dentro do método, continuamos criando uma variável chamada todo, mas agora não atribuímos a ela toda a lista, mas sim, somente o primeiro item da tabela que contenha o ID especificado na rota, utilizando o método FirstOrDefaultAsync que retornará o primeiro resultado encontrado para a expressão ou uma exceção. E como parâmetro, passamos uma simples função lamba: x => x.Id == id.
- Por fim, retornamos o item referente ao ID informado através do método *Ok(todo)* e, caso este não exista, retornamos *NotFound()*.

Para continuarmos e criarmos o método POST para finalmente passar a incluir dados na tabela, precisamos antes criar uma pasta chamada **ViewModels** e, dentro dela, criamos uma classe chamada **CreateCustomerViewModel**.

Esta classe define os dados que queremos manipular, assim como seus formato e outras regras, para que possamos utilizar nos métodos POST e PUT que serão criados posteriormente.

Vamos criar uma variável para cada valor que esperamos receber e, antes de cada uma, definiremos sua obrigatoriedade, através da tag [Required]. O código ficará da seguinte maneira:

```
using System.ComponentModel.DataAnnotations;
```

```
namespace CustomerAPI.ViewModels
{
    public class CreateCustomerViewModel
    {
        [Required]
        public string FullName { get; set; }
        [Required]
        public string Email { get; set; }
        [Required]
        public string Telephone { get; set; }
        [Required]
        public string Cpf { get; set; }
    }
}
```

Criado o ViewModel, volte para a sua classe de Controllers e importe a classe criada, logo no inicio, juntamente com os outros using (using CustomerAPI.ViewModels). Aproveite e também adicione o modelo da tabela (using CustomerAPI.Models).

Agora, adicione o método POST em seu código, que deverá ficar da seguinte forma:

```
using CustomerAPI.Data;
using CustomerAPI.Models;
using CustomerAPI.ViewModels;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace CustomerAPI.Controllers
{
    [ApiController]
    [Route(template:"customer")]
    public class CustomerController : Controller
    {
        private readonly AppDbContext _context;
        public CustomerController(AppDbContext context)
        {
            _context = context;
```

```
        }

        [HttpGet]
        public async Task<IActionResult> Get()
        {
            var customer = await _context
                .Customer
                .AsNoTracking()
                .ToListAsync();

            return Ok(customer);
        }

        [HttpGet("{id}")]
        public async Task<IActionResult> GetById(
            [FromServices] AppDbContext context,
            [FromRoute] int id)
        {
            var customer = await context
                .Customer
                .AsNoTracking()
                .FirstOrDefaultAsync(x => x.Id == id);

            return customer == null ? NotFound() : Ok(custome
        }

        public async Task<IActionResult> PostAsync(
            [FromServices] AppDbContext context,
            [FromBody] CreateCustomerViewModel model)
        {
            if (!ModelState.IsValid)
                return BadRequest();

            var customer = new Customer
            {
                FullName = model.FullName,
                                Email = model.Email,
                                Telephone = model.Telephone,
```

```
                                Cpf = model.Cpf,
            };

            try
            {
                await context.Customer.AddAsync(customer);
                await context.SaveChangesAsync();
                return Created($"customer/{customer.Id}", cus
            }
            catch (Exception e)
            {
                return BadRequest(e);
            }
        }
    }
}
```

- Novamente antes do método, informe agora ao EF Core que ele será um POST utilizando [HttpPost].
- Nomeie o método como Post e como parâmetro passe novamente o context. Agora, vamos receber também informações, desta vez para inserção de dados, que virá como JSON e as pegaremos através da ViewModel criada anteriormente.
- Desta vez, primeiro faremos uma verificação para saber se as informações recebidas batem com as informações necessárias e, caso não, será retornado um HTTP BadRequest().
- Caso as informações estejam corretas, criaremos um novo objeto com as informações vindas do ViewModel e armazenaremos elas em uma variável customer. Um detalhe aqui é que podemos também, caso necessário, passar valores não vindos do frontend, como hora de criação e/ou valores que inicias obrigatórios (done = false em uma lista de tarefas, por exemplo).
- Na sequência, abrimos um bloco Try/ Catch para, caso ocorra uma exceção, haja um retorno StatusCode(500, "Internal Server Error"), que informará ao usuário que houve uma falha no servidor, mas sem expor possíveis dados sensíveis.

- Caso esteja tudo certo com o valor recebido, o valor será enviado ao banco de dados, as informações serão salvas e então será retornado do HTTP o código 201 (Created), juntamente da URI que aponta para o objeto criado e inserido na tabela.

Vamos agora criar um método para editar os dados de uma tabela, que será através do PUT. Para isso, deixe o código da seguinte maneira:

```
using CustomerAPI.Data;
using CustomerAPI.Models;
using CustomerAPI.ViewModels;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace CustomerAPI.Controllers
{
    [ApiController]
    [Route(template:"customer")]
    public class CustomerController : Controller
    {
        private readonly AppDbContext _context;
        public CustomerController(AppDbContext context)
        {
            _context = context;
        }

        [HttpGet]
        public async Task<IActionResult> Get()
        {
            var customer = await _context
                .Customer
                .AsNoTracking()
                .ToListAsync();

            return Ok(customer);
        }

        [HttpGet("{id}")]
```

```
        public async Task<IActionResult> GetById(
            [FromServices] AppDbContext context,
            [FromRoute] int id)
        {
            var customer = await context
                .Customer
                .AsNoTracking()
                .FirstOrDefaultAsync(x => x.Id == id);

            return customer == null ? NotFound() : Ok(custome
        }

        [HttpPost]
        public async Task<IActionResult> Post(
            [FromServices] AppDbContext context,
            [FromBody] CreateCustomerViewModel model)
        {
            if (!ModelState.IsValid)
                return BadRequest();

            var customer = new Customer
            {
                FullName = model.FullName,
                Email = model.Email,
                Telephone = model.Telephone,
                Cpf = model.Cpf,
            };

            try
            {
                await context.Customer.AddAsync(customer);
                await context.SaveChangesAsync();
                return Created($"customer/{customer.Id}", cus
            }
            catch (Exception e)
            {
                return BadRequest(e);
            }
```

```
        }

        [HttpPut("{id}")]
        public async Task<IActionResult> Put(
            [FromServices] AppDbContext context,
            [FromBody] CreateCustomerViewModel model,
            [FromRoute] int id)
        {
            if (!ModelState.IsValid)
                return BadRequest();

            var customer = await context.Customer.FirstOrDefa

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

            try
            {
                customer.Email = model.Email;
                customer.Telephone = model.Telephone;

                context.Customer.Update(customer);
                await context.SaveChangesAsync();
                return Ok(customer);
            }
            catch (Exception e)
            {
                return BadRequest(e);
            }
        }
    }
}
```

- Novamente, antes da criação do método, diga ao EF Code qual será, e no caso, é o PUT: [HttpPut("{id}")]. Veja que há um ID. Ele funciona da mesma forma que o método GetById(), uma vez que vamos buscar o objeto da tabela pelo Id do mesmo, via URI.

- Como parâmetro, agora passamos o contexto para conexão, a ViewModel para receber as informações e o ID vindo da URI.
- Antes de editarmos um objeto, precisamos encontrar ele. E fazemos isso exatamente como no método GetById(). Verificamos se os dados são válidos, criamos a variável customer e nela armazenamos o resultado da busca. Caso nenhum objeto for encontrado, o sistema retornara o método HTTP 404 NotFound().
- Encontrado o objeto referente ao ID, o sistema tentará atualizar os dados no banco, através do Médoto Update(), que recebe por parâmetro os valores da ViewModel armazenados na variável customer.
- Após isso, chamamos o médoto SaveChangesAsync(), que irá salvar e finalizar a conexão de atualização. Retornamos então um HTTP Ok(customer) para informar que a atualização foi concluída.
- Caso houver falha na atualização, o retorno será um StatusCode(500, "Internal Server Error"), da mesma forma e motivo que no método Post();

Por fim, adicionamos o método DELETE ao código, que chega a sua forma final:

```
using CustomerAPI.Data;
using CustomerAPI.Models;
using CustomerAPI.ViewModels;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace CustomerAPI.Controllers
{
    [ApiController]
    [Route(template:"customer")]
    public class CustomerController : Controller
    {
        private readonly AppDbContext _context;
        public CustomerController(AppDbContext context)
        {
            _context = context;
        }
```

```
        [HttpGet]
        public async Task<IActionResult> Get()
        {
            var customer = await _context
                .Customer
                .AsNoTracking()
                .ToListAsync();

            return Ok(customer);
        }

        [HttpGet("{id}")]
        public async Task<IActionResult> GetById(
            [FromServices] AppDbContext context,
            [FromRoute] int id)
        {
            var customer = await context
                .Customer
                .AsNoTracking()
                .FirstOrDefaultAsync(x => x.Id == id);

            return customer == null ? NotFound() : Ok(custome
        }

        [HttpPost]
        public async Task<IActionResult> Post(
            [FromServices] AppDbContext context,
            [FromBody] CreateCustomerViewModel model)
        {
            if (!ModelState.IsValid)
                return BadRequest();

            var customer = new Customer
            {
                FullName = model.FullName,
                Email = model.Email,
                Telephone = model.Telephone,
                Cpf = model.Cpf,
```

```
            };

            try
            {
                await context.Customer.AddAsync(customer);
                await context.SaveChangesAsync();
                return Created($"customer/{customer.Id}", cus
            }
            catch (Exception)
            {
                return StatusCode(500, "Internal Server Error
            }
        }

        [HttpPut("{id}")]
        public async Task<IActionResult> Put(
            [FromServices] AppDbContext context,
            [FromBody] CreateCustomerViewModel model,
            [FromRoute] int id)
        {
            if (!ModelState.IsValid)
                return BadRequest();

            var customer = await context.Customer.FirstOrDefa

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

            try
            {
                customer.Email = model.Email;
                customer.Telephone = model.Telephone;

                context.Customer.Update(customer);
                await context.SaveChangesAsync();
                return Ok(customer);
            }
            catch (Exception)
```

```
            {
                return StatusCode(500, "Internal Server Error
            }
        }

        [HttpDelete("{id}")]
        public async Task<IActionResult> Delete(
            [FromServices] AppDbContext context,
            [FromRoute] int id)
        {
            var customer = await context.Customer.FirstOrDefa

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

            try
            {
                context.Customer.Remove(customer);
                await context.SaveChangesAsync();

                return Ok();
            }
            catch (Exception e)
            {
                return StatusCode(500, "Internal Server Error
            }
        }
    }
}
```

- Como de praxe, informamos qual será o método, desta vez [HttpDelete("{id}")], já informando também que haverá um ID a ser recebido pela URI.
- Criamos o método Delete() e passamos como parâmetro o contexto e o ID a ser recebido pela rota.
- Armazenamos na variável customer novamente o primeiro resultado encontrado pelo ID informado e retornamos NotFound() caso nenhum resultado seja encontrado.

- Encontrado o resultado, entramos no bloco try/ catch, o qual chama o método Remove(), passando o objeto recuperado como parâmetro. Após isso, as alterações são salvas e a coexão encerrada pelo método SaveChangesAsync(). Por fim, retornando um status Ok(), informando o sucesso.
- Caso haja algum problema na deleção, retornando um StatusCode 500, como nos métodos anteriores.