Andrius Dominiquini de Camargo

Teste técnico VaiVoa

28/05/2021

Neste artigo irei apresentar o desenvolvimento de um projeto de ApiREST utilizando .NET Core, que é uma plataforma open-source para desenvolvimento de aplicações. Também utilizarei o Entity Framework, um framework de mapeamento objeto-relacional cuja função é facilitar a interação do .NET Core com uma base de dados.

Considerando a instalação prévia dos devidos pacotes e ferramentas, iniciamos o desenvolvimento através de um template disponível no .NET Core, acessível pelo do Prompt de Comando (ou equivalente) através do uso do seguinte comando dentro do diretório onde deseja criar o projeto:

```
dotnet new mvc
```

Agora podemos acessar o projeto através de sua IDE ou editor de código favorito. O uso desse template facilita a inicialização e já prepara partes importantes do código como os arquivos Program.cs e Startup.cs , entre outros.

Primeiramente, é necessário alterar o arquivo CardsAPI.csproj para inserir as referências das ferramentas que serão utilizadas:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <RootNamespace>CardsAPI</RootNamespace>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="5.0.6"/>
    <PackageReference Include="Swashbuckle.AspNetCore" Version="5.6.3"/>
    <PackageReference Include="System.Data.SqlClient" Version="4.8.2"/>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
    Version="5.0.6"/>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Relational"
    Version="5.0.6"/>
  </ItemGroup>
</Project>
```

Seguimos para a definição do modelo, a parte do API responsável por definir as propriedades dos objetos que serão alocados para a base de dados. O uso do template já define um arquivo chamado ErrorViewModel.cs dentro da pasta Models. Iremos alterar o nome do arquivo para apenas Model.cs para facilitar a nomenclatura. Definiremos então a seguinte classe:

```
    using System;


namespace TodoApi.Models
{
    public record Card
    {
        public Guid Id { get; init; }

        public string Email { get; set; }

        public string NumCard { get; set; }

        public DateTimeOffset CreatedDate { get; set; }
```

```
    }
}
```

Aqui temos os atributos necessarios para o API que queremos criar : Id, Email, Numero do cartão (NumCard) e Data de criação do cartão (CreatedDate).

Agora, iremos criar o arquivo onde estará o código necessário para a geração dos numeros de cartão. Para isso criaremos uma pasta chamada Repositories com um arquivo chamado InMemCardsRepositories.cs. A estrutura desse arquivo será:

```
    namespace TodoApi.Repositories
{
    public class InMemCardsRepository
    {

        public static string CardNumGenerator(int length)
        {
            string cardnum = "";
            var rand= new Random(Guid.NewGuid().GetHashCode());
            while(cardnum.Length < length)
            {
                cardnum+=rand.Next();
            }
            return cardnum;
        }
```

Comentarei cada linha a seguir.

Desejamos um método chamado CardNumGenerator que recebe um atributo que define o tamanho da sequência numérica chamado length e retorna uma string com essa sequência gerada aleatoriamente. O mecanismo que gera a sequencia randomica é a classe Random fornecida pelo namespace System, que gera um numero pseudo-randomico. Assim, segue um loop do tipo while que concatena os resultados do métodos Random.Next() para montar a desejada sequência aleatória. No fim, retorna-se a string com essa sequência.

Uma vez definido o mecanismo de numeros aleatórios, precisamos definir a classe de contexto, que é a classe que irá representar as sessões feitas com o banco de dados. Nessa classe, o Entity Framework será bastante útil. Também utilizaremos o SQL Server. Nesse ponto, é necessário possuir tais ferramentas devidamente instaladas.

Criaremos uma pasta chamada Data com um arquivo CardsContext.cs com o seguinte código:

```
using Microsoft.EntityFrameworkCore;
using CardsAPI.Models;
using Microsoft.SqlServer.Server;

namespace CardsAPI.Data
{   //classe de contexto
    public class CardsContext : DbContext
    {
        //mapeamento de entidade p/ tabela
        public DbSet<Card> Card{get; set;}

        //provedor e string de conexao
        protected override void
        OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(@"Integrated Security=SSPI;
            Persist Security Info=False;Initial Catalog=DbCardApi;
            Data Source=DESKTOP-F0N0EQ6\SQLEXPRESS");
        }
```

```
    }
}
```

Nesse código eu montei uma classe criança chamada CardsContext que herda da classe padrão DbContext, simplificando a implementação da nossa API. Na definição dessa classe de contexto, utilizei o DbSet¡Card¿ para representar a coleção de todas entidades as quais serão do tipo Card, que definimos no modelo. Esse processo possibilitará realizar queries no database.

Por fim, é necessário realizar um override no método OnConfiguring para definir a string de conexão que possibilita o acesso ao banco de dados. Repare que a propriedade "Catalog=DbCardApi"define o database que será utilizado, porém ele nao precisa ter sido criado anteriormente. Utilizaremos um método a seguir para a criação deste a partir do código já feito. Tal formalismo é conhecido como Code-First.

Para criar o database no contexto do SQL Server e .NET Core com Entity Framework, após a definição da classe de contexto, basta chamar o seguinte código:

```
    dotnet ef migrations add "nome da migration"
```

Esse processo automatiza a geração do código necessário para acessar e modificar o database através de operações CRUD. O código será localizado em uma nova pasta chamada Migrations. Para atualizar a base de dados é necessário o comando:

```
    dotnet ef database update
```

Para facilitar o gerenciamentos dos bancos de dados, utilizei o programa Microsoft SQL Server Management Studio. Após a atualização do database pelo último comando citado, é possível ver no Server Management que tal banco de dados foi criado.

Agora, resta a parte final onde iremos definir os endpoints. Seguindo a metodologia escolhida iremos utilizar Controllers para mediar a interação server-client. Alteraremos o arquivo chamado HomeController.cs localizado na pasta Controllers. Alterei o nome para CardsController.cs para estar mais de acordo com nosso contexto. Alteraremos o código com o seguinte bloco:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using CardsAPI.Models;
using CardsAPI.Repositories;
using CardsAPI.Data;

namespace CardsAPI.Controllers
{
    [Route("cards")]
    [ApiController]
    public class CardsController: ControllerBase
    {

        private readonly InMemCardsRepository repository;

        public CardsController()
        {
            repository = new InMemCardsRepository();
        }

        //Receber email/gerar Cartão
        [HttpGet]
        public string CreateCard(string email)
        {
            var newCard = new Card();
            newCard.Email=email;
            newCard.CreatedDate=DateTimeOffset.UtcNow ;
```

```
            newCard.NumCard=InMemCardsRepository.CardNumGenerator(8);

            var db = new CardsContext();
            db.Add(newCard);
            db.SaveChanges();

            return newCard.NumCard;
        }

        [Route("mycards")]
        [HttpGet]
        public object GetMyCards(string email)
        {
            var db = new CardsContext();
            List<string> myCards = db.Card.Where(a => a.Email == email)
            .OrderBy(a => a.CreatedDate)
            .Select(a => a.NumCard).ToList();
            if (!myCards.Any())
            {
                string msg = "Usuário não encontrado";
            return msg;
            }

            return myCards;


        }
    }
}
```

Como é mais extensa do que as classes restantes, explicarei por partes. Primeiramente, definimos a classe CardsController que herda de ControllerBase, facilitando e simplificando a implementação do código. Definimos também os atributos [Route(”cards”)] e [ApiController], O primeiro define o prefixo de rota para todas as ações deste controlador, já o segundo fornece a classe alguns compportamentos uteis típicos de API. Esse tipo de definição de atributos é uma característica do namespace Microsoft.AspNetCore.Mvc.

Em seguida, é necessário criar uma instância somente de leitura da classe InMemCardsRepository, para possibilitar o acesso as funcionalidades definidas nela. A aprte correspondente do código é:

```
    private readonly InMemCardsRepository repository;

        public CardsController()
        {
            repository = new InMemCardsRepository();
        }
```

Agora será feito o primeiro endpoint, responsável por receber o email do usuário e fornecer um número de cartão. Para realizar a inclusão do novo cartão na base de dados é necessário instanciar a classe de contexto CardsContext() e usar os métodos .Add() e .SaveChanges. Aqui é utilizado também o método CardNumGenerator com atributo de tamanho da sequencia numérica definido para 8 (numero arbitrário). Também é realizada uma instância do modelo Card()

```
    public string CreateCard(string email)
        {
            var newCard = new Card();
            newCard.Email=email;
            newCard.CreatedDate=DateTimeOffset.UtcNow ;
            newCard.NumCard=InMemCardsRepository.CardNumGenerator(8);

            var db = new CardsContext();
```

```
        db.Add(newCard);
        db.SaveChanges();

        return newCard.NumCard;
    }
```

O próximo endpoint será o responsável por retornar ao cliente uma lista de todos seus cartões ordenados por data de criação. Para isso é definida uma nova rota e definida a classe GetMyCards(), que recebe como atributo o email do usuário. É feita uma instância da classe de contexto, e através dela será chamados os métodos Where() e OrderBy(), que permitem acessar o banco de dados, procurar as entidades que correspondem ao email do usuário e ordenar o retorno por ordem de data de criação, respectivamente.

```
    [Route("mycards")]
    [HttpGet]
    public object GetMyCards(string email)
    {
        var db = new CardsContext();
        List<string> myCards = db.Card.Where(a => a.Email == email).OrderBy(a => a.C
        .Select(a => a.NumCard).ToList();

        if (!myCards.Any())
        {
            string msg = "Usuário não encontrado";
        return msg;
        }

        return myCards;


    }
```

Feito isso, basta executar os comandos os comandos ”dotnet build”seguido de ”dotnet run”para possibilitar o acesso aos endpoints definidos na API. Com isso, concluo essa explicação.