Olá devs.

Hoje vou compartilhar como concluí um projeto de API REST que fornece um sistema de geração de número de cartão de crédito virtual.

A API gera números aleatórios para um pedido de novo cartão. Cada cartão gerado é associado a um email que identifica a pessoa que está utilizando.

São 2 endpoints. Um recebe o email da pessoa e retorna um objeto de resposta com o número do cartão de crédito. O outro endpoint lista, em ordem de criação, todos os cartões de crédito virtuais de um solicitante (passando seu email como parâmetro).

Este artigo considera que o leitor já está familiarizado com a linguagem C#, tem noções de Entity Framework. Irei comentar os arquivos e a lógica aplicada. Para ver os módulos no detalhe acesse Gilson401/modal_a: modalmais (github.com).

Lets Go!

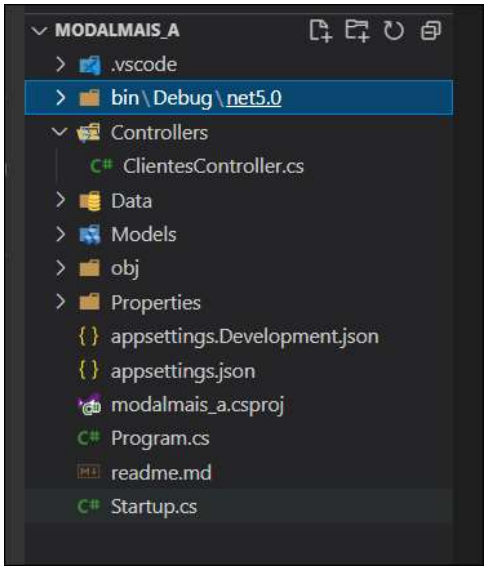
Inicialmente é necessário configurar o ambiente de desenvolvimento. O projeto foi feito com ajuda do VSCODE, então recomendo instalar as extensões necessárias no VSCODE para C#. A primeira de uma busca simples já atende.

Instale o Microsoft .NET CORE SDK mais recente e com suporte.

Com os itens instalados inicie um novo projeto com o comando dotnet new webapi -o e o nome do teu projeto. Este comando gera o projeto com base num boilerplate de aplicação web com servidor e sem front end, com conexão ao banco de dados Mcrosoft MYSQL.

O boierplate com Entity Framework já deixa tudo prontinho, com pasta com models, controladores de acesso , roteador para os endpoints etc. É uma espécie de “hello world” com previsão do tempo.

A estrutura das pastas fica conforme abaixo:



O boilerplate vem com a classe WeatherForecast.cs e o controller Controllers/WeatherForecastController.cs. Use-os como base o para se guiar, caso já esteja familiarizado, podem ser excluídos.

Em /Data fica o arquivo com o a definição do contexto do banco de dados. É a classe que diz com qual banco de dados está trabalhando. Aqui neste projeto estamos trabalhando com a dependência

Microsoft.EntityFrameworkCore.InMemory. Ela faz com que usemos um banco de dados na memória, dispensando a instalação de bancos pesados e ou caros em desenvolvimento.

Startup.cs , como sugere o nome, é script que dá o pontapé no programa.

appsettings.Development.json e appsettings.json são os arquivos de configuração. O primeiro é para o desenvolvimento. (mudando o nome intermediário para Production ou Staging geramos o arquivo para a ambiente indicada.) Neles podemos definir a conection string entre outras coisas.

Properties\launchSettings.json tem as configurações gerais de inicialização. Nele é informado qual o ambiente será usado. Neste arquivo temsoa s variáveis de ambiente.

Models\Clientes.cs. Em models ficam as definições dos objetos que estarão no banco de dados. Neste exemplo Creitei um modelo muito simples. Os registros terão Id, Email e nº de celular.

Data\DataContext.cs é o arquivo necessário para trabalhar com o EntityFramework.

Data > C# DataContext.cs

```
using Microsoft.EntityFrameworkCore;

using modalmais_a.Models;
namespace modalmais_a.Data
{
    public class DataContext : DbContext
    {
        //estou usando bd em memória.se fosse um banco mesmo ficaria aqui a string de conexão
        public DataContext(DbContextOptions<DataContext> options)
        : base(options)
        {

        }
        //Aqui são informados as tabelas que serão usada na api. No nosso exemplo apenas teremos a tabela de clientes
        public DbSet<Clientes> Clientes { get; set; }
    }

}
```

Controllers\ClientesController.cs é o nosso controlador onde definimos os endpoints.

Um endpoint para receber as requisições em que o cliente pede um nº de cartão virtual. É enviada uma requisição do tipo post com o email do cliente no body. Se passar pelo modelo entra no for que hera uma concatenação de 16 números aleatórios, adiciona ao “objeto” cliente, grava no banco de dados e responde a requisição com este objeto.

```
        [HttpPost]
        [Route("")]

        public async Task<ActionResult<Clientes>> Post(
        [FromServices] DataContext context,
        [FromBody] Clientes model)
        {
            if (ModelState.IsValid)
            {
                Random rnd = new Random();

                string randomBetween10And20 = "";
                for (int a = 0; a < 16; a = a + 1)
                {
                    randomBetween10And20 += rnd.Next(0, 9).ToString();
                }

                model.Cartao = randomBetween10And20;
                context.Clientes.Add(model);
                await context.SaveChangesAsync();
                return model;
            }
            else
            {
                return BadRequest(ModelState);
            }
        }
    }
```

O outro endpoint é para requisição GET, na qual deve ser enviada como param o email do cliente. O parâmetro email é usado para fazer o select no banco de dados retornando todos os registros para este determinado email e seus respectivos nº de cartão.

Jogamos os números num vetor e o inserimos num ExpandoObject , juntamente com o email, que será respondido para o requisitante.

```
19        [HttpGet("{email}")]
20        [Route("")]
21
22        public async Task<ActionResult<ExpandoObject>> Get([FromServices] DataContext context, string email)
23        {
24            var listDataBd = await context.Clientes
25            .AsNoTracking()
26            .Where(x => x.Email == email)
27            .Distinct()
28            .OrderBy(x => x.Id)
29            .ToListAsync();
30
31            dynamic ObjectToReturn = new ExpandoObject();
32            ObjectToReturn.email = email.ToString();
33            ObjectToReturn.listaDeCartoes = new List<string>();
34
35            listDataBd.ForEach(num => ObjectToReturn.listaDeCartoes.Add(num.Cartao));
36
37            return ObjectToReturn;
38        }
```

Basicamente é isso. Se o editor não apontar erros mande um dotnet run no terminal e seja feliz!

Obrigado e até a próxima!