Olá galera tudo bem?

Meu nome é Dyonata Machado e hoje eu vim aqui para ensinar vocês a criar uma API Web Rest com Entity Framework. E eu quero mostrar aqui que esta tarefa é muito fácil.

- ***API***

A API deverá gerar números aleatórios para o pedido de novo cartão virtual. Cada cartão gerado deve estar associado a um email para identificar a pessoa que está utilizando.

Essencialmente são 2 endpoints. Um receberá o email da pessoa e retornará um objeto de resposta com o número do cartão de crédito. E o outro endpoint deverá listar, em ordem de criação, todos os cartões de crédito virtuais de um solicitante (passando seu email como parâmetro).

- ***Configurações Iniciais***

Nosso primeiro passo é iniciar o Visual Studio 2019 e selecionar a opção “Criar Um Projeto”. Na lista de opções de projeto que aparece selecionamos a opção API Web do ASP.NET Core para C#. Abrirá a janela de configuração, onde podemos inserir o nome do projeto, a solução e o diretório do projeto. Clique em próximo e selecione a versão 5 do .NET no campo “Estrutura de Destino”, mantenha as demais configurações como padrão e clique em Criar.

O Visual Studio vai abrir o projeto. Na Janela “Gerenciador de Soluções” o .NET já criou alguns dos arquivos que vamos utilizar. Mas precisamos excluir a classe exemplo que o .NET cria e sua classe Controller. Para isso selecione a classe WeatherForecast e aperte Del para excluí-la. Faça o mesmo com a classe WeatherForecastController na pasta Controllers.

- ***Classe Modelo***

Depois disto iremos criar a classe modelo para o nosso banco de dados que vai relacionar-se com as colunas da tabela do banco de dados. Para manter mais organizado clique com o botão direito no arquivo que tem o nome do projeto que criamos (este é o nosso arquivo .csproj) e depois clique em Adicionar Nova Pasta. Dê a ela o nome Models. Em seguida clique com o botão direito na pasta Models e clique em Adicionar

Nova Classe. Abrirá uma nova Janela onde você irá escolher a opção classe e inserir o nome da nossa classe que se chamará Cards.

O .NET irá abrir a nossa classe. Iremos inserir apenas os atributos de que necessitamos. Não é necessário criar nenhum método pois esta classe serve apenas para se referenciar às tabelas do banco de dados. Portanto insira na classe os seguintes atributos (Id, Numero, DtCriacao e Email).

```
public class Cards

{
    public int Id { get; set; }
    public string Numero { get; set; }
    public DateTime DtCriacao { get; set; }
    public string Email { get; set; }
}
```

Basta inserir o código acima. O Id representará a chave primária da nossa tabela. Numero é o número do cartão virtual que iremos criar.

- ***Criando o Banco de Dados***

Depois disto, como estamos trabalhando com uma abordagem DataBase First criaremos nosso banco de dados com os mesmos atributos. Para isso abra o SQL Server. Clique em Nova Consulta e digite o comando: "*create database cartões;"* depois clique em executar. Depois digite o comando:

```
create table Cards(
Id int identity(1,1) primary key not null,
Numero varchar(4) not null,
DtCriacao DateTime,
Email varchar(40) not null
);
```

Note que estamos inserindo os mesmos tipos de dados que inserimos em nossa classe. Note o termo "identity". Ele configura nossa chave primária (isto é o Id) com o valor inicial 1 e com um auto incremento de 1 a cada nova instância. Todos os campos não podem ter valores nulos.

- ***Scaffold e String de Conexão***

Agora faremos uso da ferramenta Scaffold que irá configurar os verbos HTTP e as heranças DbContext que precisamos para controlar o

banco de dados. Basta clicar com o botão direito na pasta Controllers e Adicionar novo Controlador. Na janela que abre selecionamos a opção API e depois selecionamos a opção “Controlador API com ações, usando o Entity Framework” e clique em criar. Na janela que abrirá no campo Classe Modelo selecione a classe Card que criamos. No campo Classe de contexto de dados clique no sinal de + ao lado e depois clique em adicionar. Por último clique em criar.

O Scaffold irá configurar nossas classes de contexto, que são aquelas que fazem a conexão com o banco de dados, irá configurar também nossos verbos HTTP e irá configurar nossa string de conexão com o banco de dados. Eventualmente o Scaffold talvez não consiga fazer a instalação de algum pacote. Então teremos que instalar o pacote que ele informará direto na biblioteca NuGet e depois repetir o processo. Outro problema que pode acontecer é com relação a string de conexão que precisaremos configurar de forma mais precisa no arquivo appsentings.json. Basta configurar desta forma:

```
  "ConnectionStrings": {

    "VirtualCardsContext": "Server=DESKTOP-FBG2D38\\SQLEXPRESS;
Database=cartoes; Integrated Security=true;"
  }
```

- *GET*

E a partir deste momento já temos efetivamente uma API Web configurada na classe CardsControllers. Porém ela não está adequada para nossa regra de negócios. Portanto vamos escrevê-la de acordo com o solicitado.

Mantemos esta parte do código:

```
[Route("api/[controller]")]
    [ApiController]
    public class CardsController : ControllerBase
    {
        private readonly VirtualCardsContext _context;

        public CardsController(VirtualCardsContext context)
        {
            _context = context;
        }
```

Depois iremos escrever o método GET que necessitamos. Para isto inserimos estes códigos:

```
        [HttpGet("get-by-email")]
        [ProducesResponseType(StatusCodes.Status200OK)]
        [ProducesResponseType(StatusCodes.Status404NotFound)]
```

A primeira linha determina qual verbo HTTP estamos escrevendo. A informação que fica entre parênteses é a nossa URL. Depois nas linhas seguintes informamos o tipo de resposta que o front-end pode esperar de nossa API. Neste caso são duas respostas. Código 200 quando conseguimos encontrar o recurso no banco de dados e 404 quando não encontrado.

Depois criamos a seguinte classe:

```
public async Task<ActionResult<Cards>> GetCardsByEmail(string email)
        {

}
```

O termo async nos informa que esta API é assíncrona. Task é um tipo de operação assíncrona que pode nos retornar um valor. O termo “GetCardsByEmail” é o nome do nosso método e ele recebe como parâmetro o e-mail que será usado para criar o cartão virtual. Dentro do método iremos inserir uma condicional para verificar se o usuário digitou um valor nulo ou apenas espaços em branco pois estes não podem gerar um novo cartão.

```
if (string.IsNullOrWhiteSpace(email))
            {
                return NotFound();
            }
```

Note que utilizei um método que já consta nas bibliotecas do .NET e passei a string digitada como parâmetro. Se o usuário digitar um valor nulo ou em branco este método vai retornar o valor booleano *true* e então

retornará *NotFound()* que é o método que usamos para devolver a resposta 404 do protocolo HTTP.

Caso o usuário tenha digitado um valor correto para email entraremos neste código:

```
var cards = await _context.Cards
    .Where(x => x.Email == email)
    .OrderBy(x => x.DtCriacao)
    .ToListAsync();
```

Onde criamos uma variável cards para avaliar se o e-mail digitado está associado a algum cartão. Fazemos isto usando a tarefa *await* que aguarda o retorno da lista dos elementos conforme as condições que usaremos. Neste caso temos expressões lambdas para representar as condições. Note que as condições são expressas em métodos que lembram as tarefas de manipulação em linguagem SQL como *Where* e *Order By*. No método *Where* buscamos os e-mails que estão cadastrados na coluna Email do banco de dados com a mesma string que o usuário forneceu. No método OrderBy estamos ordenando a lista conforme a Data de Criação que por default é ascendente, o que se enquadra com nossa regra de negócio. E por fim o método *ToListAsync* é uma chamada para listar todos os elementos que satisfazem as condições que inserimos em Where em API's assíncronas. Todos os resultados que satisfazem as condições serão guardados na variável *cards*.

Depois disto ainda faremos uma verificação em *cards* para verificar se o resultado é nulo, isto é, se não foi encontrado nenhum resultado para a pesquisa *Where* que fizemos. Se o resultado for nulo, retornaremos 404 com o método *NotFound*.

```
if (cards == null)
{
    return NotFound();
}
```

Mas caso sejam encontrados valores, retornaremos os dados dos cartões encontrados com o comando *return base.Ok(cards);*

```
[HttpGet("get-by-email")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<ActionResult<Cards>> GetCardsByEmail(string email)
{
    if (string.IsNullOrWhiteSpace(email))
    {
        return NotFound();
    }

    var cards = await _context.Cards
        .Where(x => x.Email == email)
        .OrderBy(x => x.DtCriacao)
        .ToListAsync();

    if (cards == null)
    {
        return NotFound();
    }

    return base.Ok(cards);
}
```

E esse é nosso método GetCardsByEmail que corresponde ao verbo HTTP GET.

- ***POST***

Agora faremos a configuração do nosso POST que será usado para gerar novos cartões virtuais ao receber a informação de e-mail do usuário. Para isto iniciaremos o código da seguinte forma:

```
[HttpPost("post-cards")]
[ProducesResponseType(StatusCodes.Status201Created)]
```

Onde *HttpPost* define o verbo HTTP e cria a rota *post-cards.* E *ProducesResponseType* define o tipo de retorno que o front-end receberá deste método.

Como a API será consumida primeiramente pelo front-end e para este método o front-end não precisa ter acesso a configurar todos os atributos do objeto que for instanciado no banco de dados, iremos criar uma nova classe chamada CardsDTO que eu inseri numa pasta chamada

DTO que permitirá acesso do front-end apenas para a configuração do e-mail para geração do cartão. Veja ela abaixo:

```
namespace VirtualCards.DTO
{
    public class CardsDTO
    {
        public string Email { get; set; }
    }
}
```

Agora voltaremos para nossa classe CardsControllers onde configuraremos nosso método Post.

```
        public async Task<ActionResult<Cards>> PostCards(CardsDTO cards)
        {

}
```

Note que é quase idêntico ao início do método GET que implementei anteriormente. Temos apenas a mudança do nome para *PostCards* e o parâmetro *cards* que o método recebe desta vez vem da classe CardsDTO que criamos.

Agora dentro do método iremos primeiramente gerar o número aleatório para o cartão conforme nossa regra de negócio. Eu escolhi trabalhar com números de 4 dígitos mas isto pode ser alterado facilmente. Para gerar o número aleatório segue a criação de uma instância Random. Também criarei uma nova instância de cartão da classe *Cards* para gerar o cartão no banco de dados.

```
Random numero = new Random();
Cards card = new Cards();
```

Agora iremos configurar os atributos do cartão gerado:

```
            card.DtCriacao = DateTime.Now;
            card.Email = cards.Email;
            card.Numero = numero.Next(1000, 9999).ToString();
```

Note que o atributo *DtCriacao* recebe a data atual. O atributo *Email* recebe a string de e-mail informada pelo usuário como parâmetro do método. E o atributo *Numero* recebe um valor aleatório gerado pelo método Next que depois é convertido para string.

Nosso método Post está praticamente pronto. Como ele faz alterações no banco de dados precisamos adicioná-lo ao banco e depois salvar as alterações. O legal é que a classe DbContext que o Scaffold configurou a herança automaticamente tem métodos prontos para fazer as duas atividades conforme o código abaixo:

```
            _context.Cards.Add(card);
            await _context.SaveChangesAsync();
```

Na primeira linha o cartão é adicionado ao banco e na segunda as mudanças são confirmadas. Agora basta dar o retorno ao usuário com o código a seguir:

```
return CreatedAtAction("PostCards", new { Id = card.Id }, card.Numero);
```

CreatedAtAction nos retornará o código HTTP 201, o Id do cartão no banco de dados e o número de cartão conforme nossa regra de negócios.

Segue o código completo do método Post que criamos:

```
        [HttpPost("post-cards")]
        [ProducesResponseType(StatusCodes.Status201Created)]
        public async Task<ActionResult<Cards>> PostCards(CardsDTO cards)
        {
            Random numero = new Random();
            Cards card = new Cards();

            card.DtCriacao = DateTime.Now;
            card.Email = cards.Email;
            card.Numero = numero.Next(1000, 9999).ToString();

            _context.Cards.Add(card);
            await _context.SaveChangesAsync();

            return CreatedAtAction("PostCards", new { Id = card.Id },
card.Numero);


        }
```

Agora sim nossa API Web está preparada para ser tratada pelo front-end para ser apresentada ao usuário com um visual mais elegante porém antes precisamos testar o funcionamento. Para isto nesta versão o .NET já configura o aplicativo Swagger para testarmos nossa API. Basta colocar o projeto em execução no botão IIS Express. Ele vai abrir uma página do Swagger com a lista de verbos HTTP que criamos onde podemos fazer os testes de funcionamento inserindo os parâmetros necessários.