

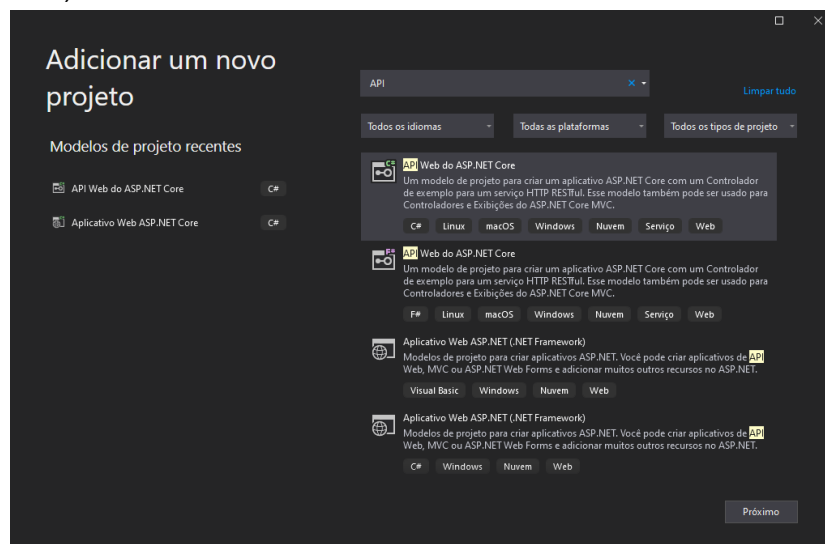
Criando uma API de Gerenciamento de Cartões em C# com .Net Core e Entity Framework

Neste tutorial, vamos aprender a criar uma **API REST** com o **Web API**, esse é um recurso do **.NET Framework** que permite criar de uma forma simples e serviços **REST**, serviços com essa arquitetura tem uma usabilidade mais ampla, um serviço **REST** usando o **WebAPI** não fica restrito a plataforma, podendo ser consumido por outros serviços.

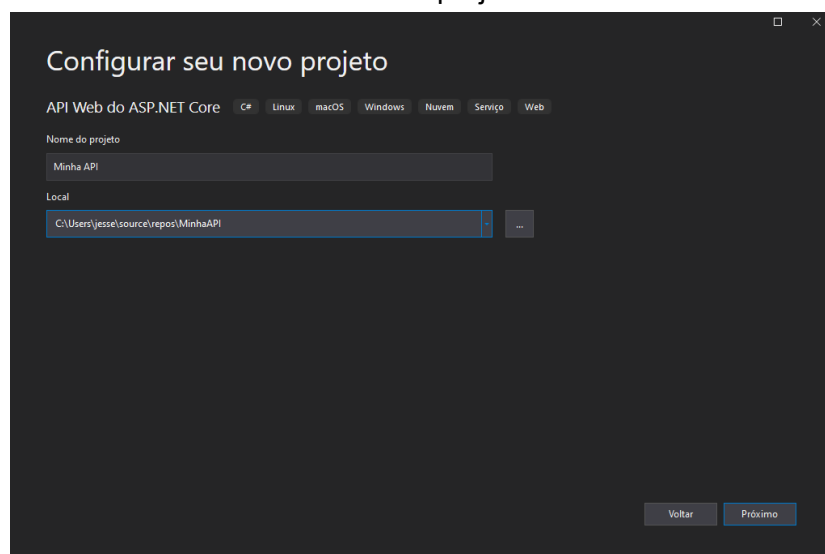
O que vamos utilizar:

- Visual Studio 2019 [Download](#)
- .NET Core e .NET Core [Download](#)

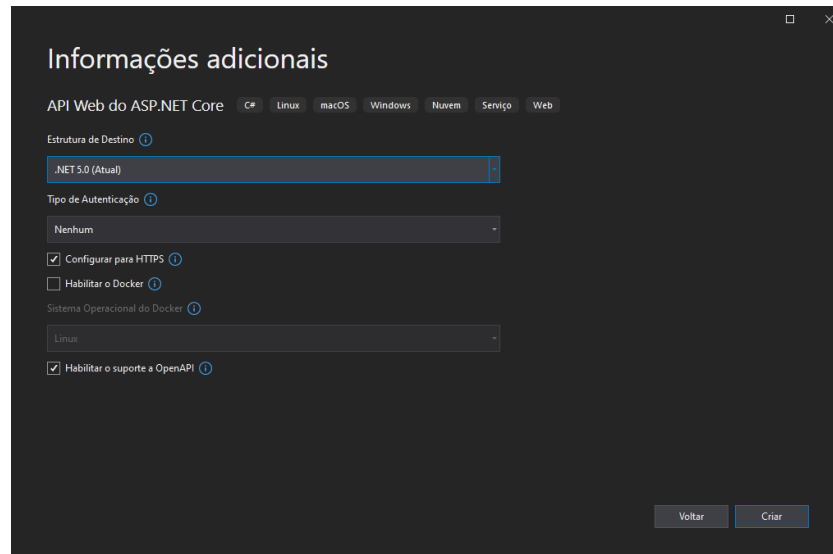
O primeiro passo é criar um novo projeto no Visual Studio selecionando o API Web do ASP .NET Core (em C#).



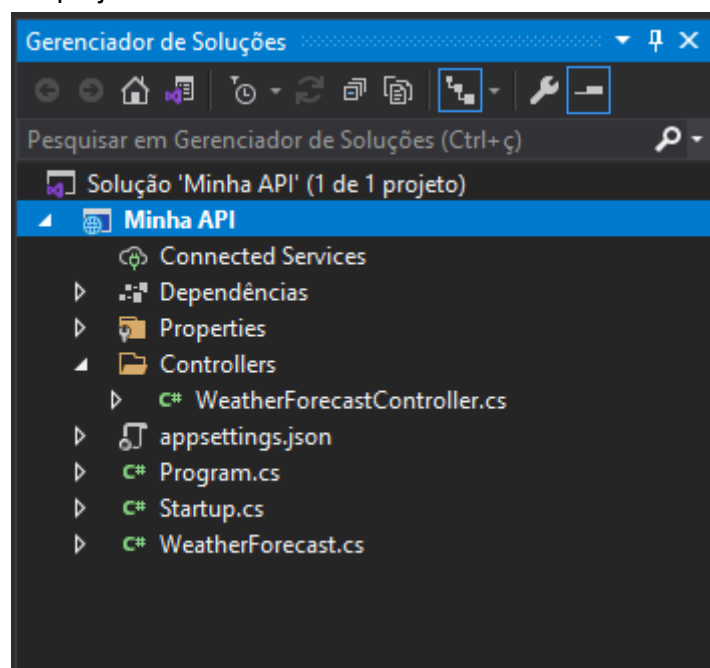
Depois é definido o nome do domínio e onde o projeto será salvo.



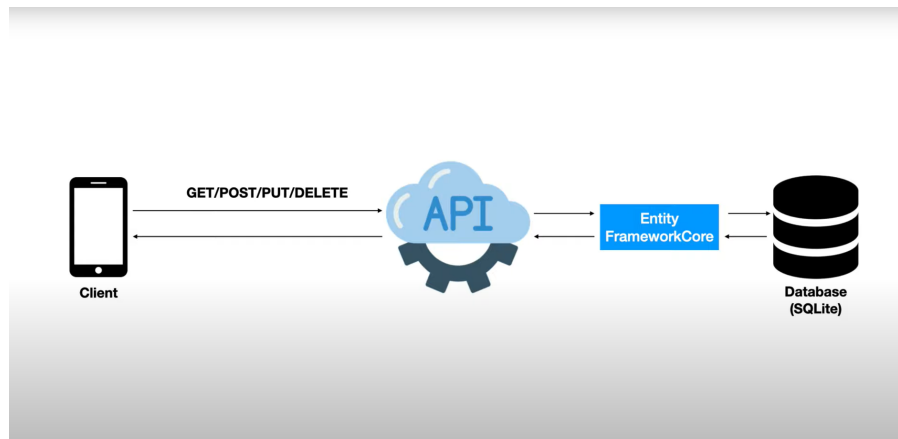
E para finalizar as configurações do projeto, basta deixar tudo como padrão, atentando se todos as caixas como a da imagem estão marcadas, em especial a opção “**habilitar o suporte a OpenAPI**”



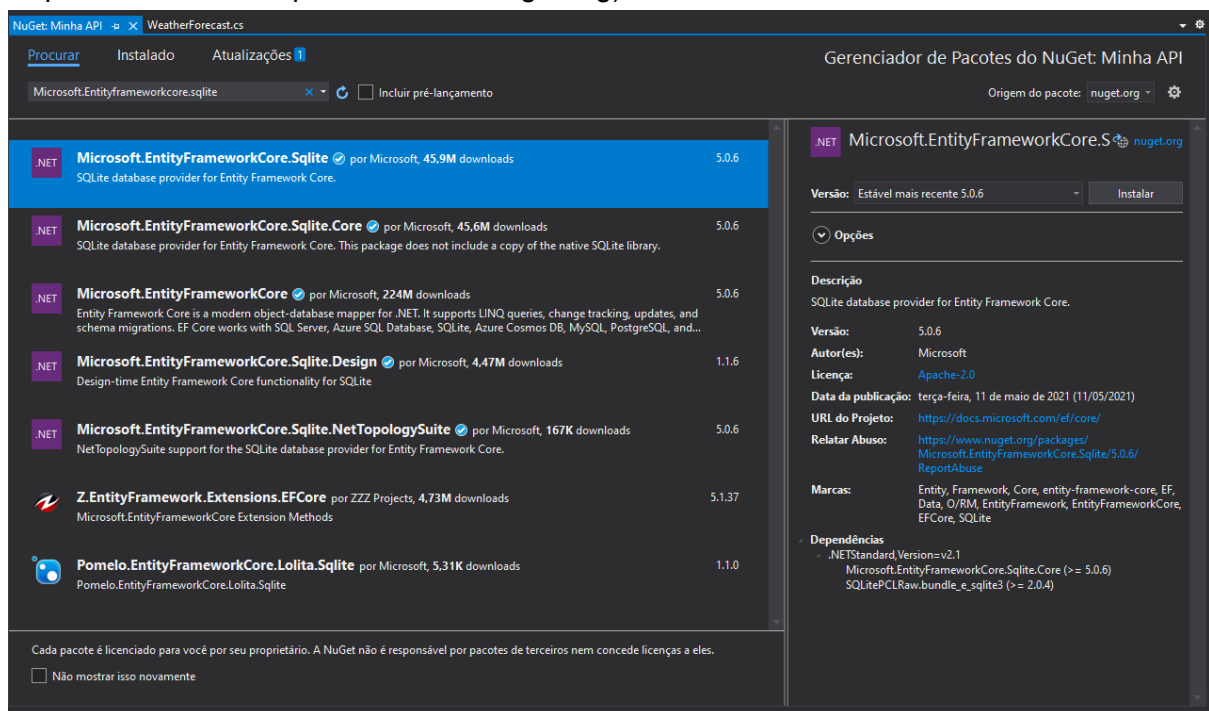
Depois de carregar a interface de desenvolvimento, no canto superior direito existe um explorador de arquivos chamado “Gerenciador de Soluções”. Nele precisamos remover dois arquivos, o **WeatherForecast.cs** na raiz e o arquivo **Controllers/WeatherForecastController.cs**. São arquivos de exemplo e não são necessários no nosso projeto.



Por trás da API vamos usar EntityFrameworkCore para interagir com o banco de dados SQLite, a EntityFramework é um mapa de objeto relacional que permite trabalhar com o banco de dados usando o objeto .NET de instância SQL.



Agora vamos adicionar o SQLite, em gerenciador de soluções clique com o botão direito no projeto e vá até **Gerenciar Pacotes do NuGet**, em procurar digite **Microsoft.EntityFrameworkCore.Sqlite** e instale. (caso não apareça, verifique se a origem do pacote no canto superior direito é nuget.org)



Usando o Framework core o acesso ao data é realizado usando um model. O model é a definição dos atributos que aquela classe vai possuir, definindo como ela vai ser gravada no banco e suas regras. Vamos criar a pasta Models e depois adicionar a classe Card.

No Model possui os seguintes atributos:

- > Card o Id é o identificador único de cada objeto
- > CardNumber é o número de cartão de crédito
- > Email é o email do portador do cartão que o identifica

```

1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel.DataAnnotations.Schema;
4  using System.ComponentModel.DataAnnotations;
5  using System.Linq;
6  using System.Threading.Tasks;
7
8  namespace CardGenerator.Models
9  {
10     public class Card
11     {
12         [Key()]
13         public int Id { get; set; }
14         public string CardNumber { get; set; }
15         public string Email { get; set; }
16     }
17 }

```

Entity Framework também necessita de um objeto de contexto, o objeto de contexto permite requisitar e salvar data. No mesmo diretório de Card crie outra classe chamada CardContext que herda da classe DbContext, no construtor os parâmetros são o DbContextOptions<CardContext> e options é a configuração do context que é injetada através de injeção de dependência em :base(options).

Também vamos expor a propriedade DbSet que representa a coleção de cartões.

O Database.EnsureCreated garante que a database será criada quando usado o contexto.

```

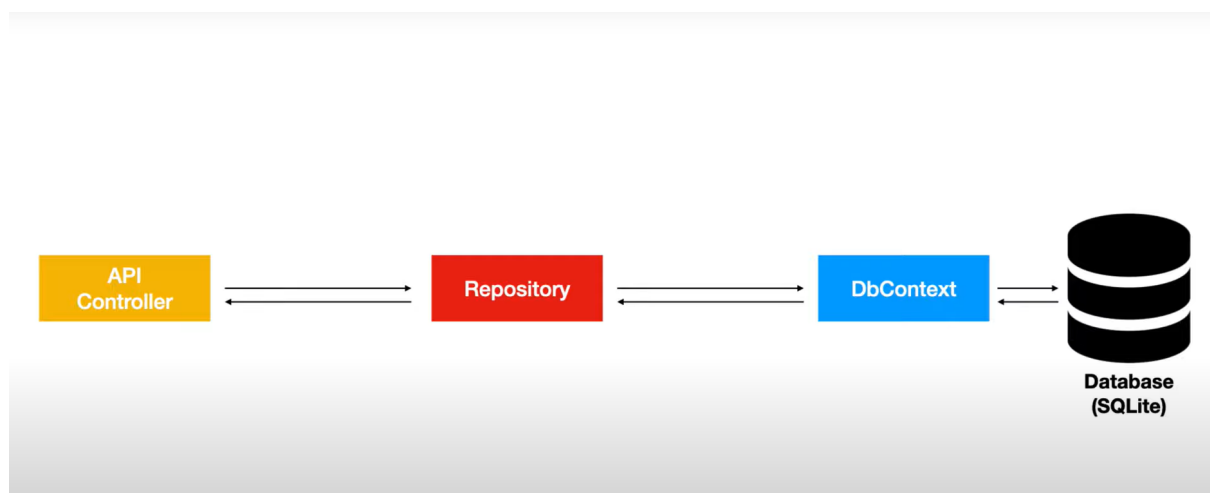
1  using Microsoft.EntityFrameworkCore;
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Threading.Tasks;
6
7  namespace CardGenerator.Models
8  {
9      public class CardContext : DbContext
10     {
11         public CardContext(DbContextOptions<CardContext> options)
12             : base(options)
13         {
14             Database.EnsureCreated();
15         }
16         public DbSet<Card> Cards { get; set; }
17     }
18 }

```

Depois de criado o contexto do Card, nós precisamos registrar para injeção de dependência atualizando o `ConfigureServices` no arquivo `startup.cs`. Usaremos o **AddDbContext** para registrar o **CardContext** com o `asp.net`, também vamos prover uma conexão com o **SQLite** database sendo um simples arquivo no projeto.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<CardContext>(o => o.UseSqlite("Data source=cards.db"));
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "CardGenerator", Version = "v1" });
    });
}
```

Com o contexto configurado, podemos adicionar o repositório. O repositório é uma camada entre a nossa aplicação e a camada de acesso de dados, que no nosso caso é o `CardContext`, adicionar o repositório é uma boa prática de programação pois cria uma camada de abstração entre o nosso código e a camada de acesso aos arquivos.



Vamos criar uma nova pasta chamada `Repositories` e criar uma nova interface nela, chamada `ICardRepository`, esta interface descreve as operações do database. Que são:

- > Listar todos os cartões
- > Listar um único por email
- > Lista um único cartão por id
- > Criar um novo Card
- > Atualizar um Card
- > Deletar o Card.

```

1  using CardGenerator.Models;
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Threading.Tasks;
6
7  namespace CardGenerator.Repositories
8  {
9      4 referências
10     public interface ICardRepository
11     {
12         2 referências
13         Task<IEnumerable<Card>> Get();
14
15         1 referência
16         Task<IEnumerable<Card>> Get(String email);
17
18         2 referências
19         Task<Card> Get(int id);
20
21         2 referências
22         Task<Card> Create(Card card);
23
24         2 referências
25         Task Update(Card card);
26
27         2 referências
28         Task Delete(int id);
29     }
30 }

```

Depois criamos uma classe CardRepository para implementar a interface ICardRepository. O repository vai requisitar o database usando o CardContext, injetamos o context através do construtor. No Created vamos usar o Add Method do DbSet para adicionar a nova instância de Card, para salvar as mudanças usamos o método SaveChangesAsync, não esquecendo de tornar async os métodos pois é utilizado o await para aguardar o database. No delete Usamos o FindAsync para buscar pelo Id, depois o .Remove com o Card a ser deletado e SaveChangesAsync para salvar. Temos também dois métodos para retornar uma lista de Cards, a primeira retorna todos os Cards cadastrados no banco usando o ToListAsync, já o segundo recebe também o email para filtrar por uma lista específica. Existe um ultimo find que busca um específico por id e por último o método Update que usa o EntityState.Modified para editar e o SaveChangesAsync para salvar a mudança.

```

1 using CardGenerator.Models;
2 using Microsoft.EntityFrameworkCore;
3 using System;
4 using System.Collections.Generic;
5 using System.Linq;
6 using System.Threading.Tasks;
7
8 namespace CardGenerator.Repositories
9 {
10     public class CardRepository : ICardRepository
11     {
12         private readonly CardContext _context;
13
14         public CardRepository(CardContext context)
15         {
16             _context = context;
17         }
18
19         public async Task<Card> Create(Card card)
20         {
21             _context.Cards.Add(card);
22             await _context.SaveChangesAsync();
23
24             return card;
25         }
26
27         public async Task Delete(int id)
28         {
29             var cardToDelete = await _context.Cards.FindAsync(id);
30             _context.Cards.Remove(cardToDelete);
31             await _context.SaveChangesAsync();
32         }
33
34         public async Task<IEnumerable<Card>> Get()
35         {
36             return await _context.Cards.ToListAsync();
37         }
38
39         public async Task<IEnumerable<Card>> Get(string email)
40         {
41             return await _context.Cards.ToListAsync();
42         }
43
44         public async Task<Card> Get(int id)
45         {
46             return await _context.Cards.FindAsync(id);
47         }
48
49         public async Task Update(Card card)
50         {
51             _context.Entry(card).State = EntityState.Modified;
52             await _context.SaveChangesAsync();
53         }
54     }
55 }
56

```

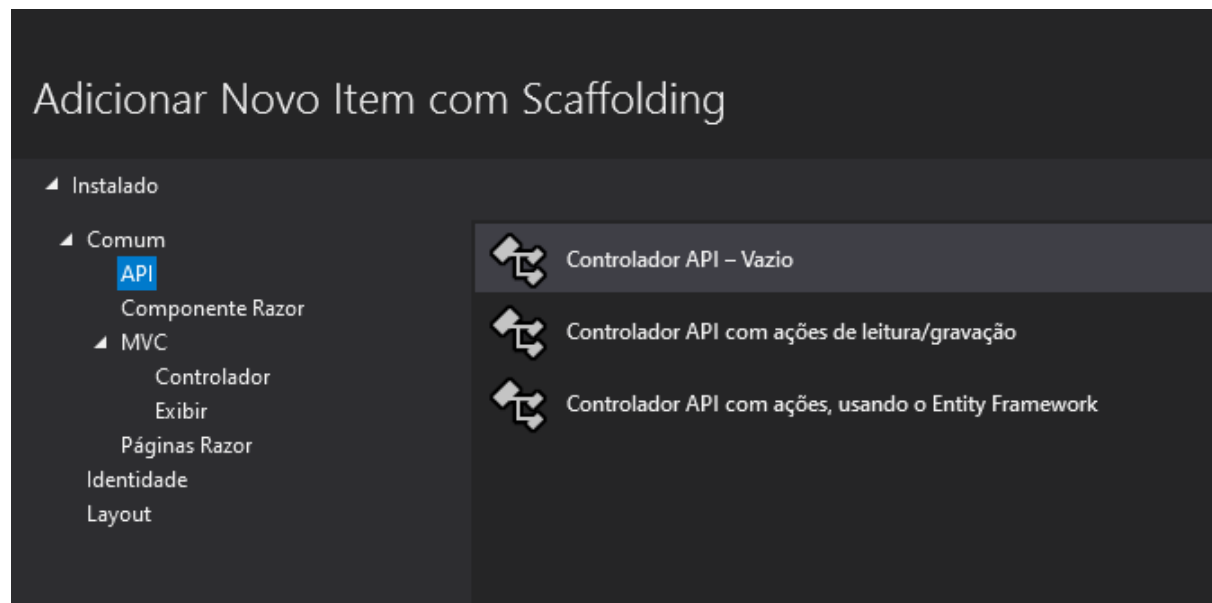
Agora precisamos fazer uma última atualização no arquivo Startup.cs adicionando a ConfigureServices o AddScoped que vai registrar uma instancia do CardRepository, isso garante também que apenas uma instância será criada para um dado http request.

```

0 referências
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<ICardRepository, CardRepository>();
    services.AddDbContext<CardContext>(o => o.UseSqlite("Data source=cards.db"));
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "CardGenerator", Version = "v1" });
    });
}

```

Agora vamos criar uma API controller. Uma API controller é a classe responsável por administrar as requisições para um endpoint. Click com o botão direito do mouse em controller no gerenciador de soluções, vá até adicionar > controlador. Click em API no menu esquerdo e selecione o Controlador API-Vazio e adicione.



Nomeie a classe de CardsController.

Primeiro podemos ver o Route que por padrão vem definido api/controller, que é o path que o controller vai administrar. No nosso caso, api/cards.

O controller herda de ControllerBase que provê muitas propriedades e methods úteis. O controller necessita de uma instância do CardRepository para interagir com o database, por isso injetamos o book repositório no construtor.

```
[Route("api/[controller]")]
[ApiController]
1 referência
public class CardsController : ControllerBase
{
    private readonly ICardRepository _cardRepository;

    0 referências
    public CardsController(ICardRepository cardRepository)
    {
        _cardRepository = cardRepository;
    }
}
```


Agora vamos gerenciar um específico http request, que são as actions, nós vamos criar uma para cada endpoint que quisermos administrar.

Acima de cada endpoint colocamos o atributo do método de requisição que vamos usar para o asp.net entender o seu tipo. Como [HttpGet]. [HttpPost] e etc.

No primeiro método temos o GetCardsByEmail que usa o IEnumerable<Cards> por retornar uma lista de cartões cadastrado com o email. Ele usa o _cardRepository.Get para pegar a lista e depois filtra pelo email recebido na requisição. E retorna a lista filtrada.

```
[HttpGet("user/{email}")]
0 referências
public async Task<IEnumerable<Card>> GetCardsByEmail(String email)
{
    var cards = await _cardRepository.Get();
    var filter = from c in cards
                 where c.Email == email
                 select c;
    return filter;
}
```

O segundo é PostCard que recebe um email e retorna o número de cartão de crédito gerado. Ele usa um objeto Random para gerar a segunda metade do cartão aleatoriamente, pois a primeira metade são informações fixas do banco, cria um objeto Card e adiciona nele o número do cartão e o email que foi recebido e cria no database o Card, retornando ao usuário o número do cartão criado.

```
[HttpPost("{email}")]
0 referências
public async Task<ActionResult<String>> PostCard(string email)
{
    Card card = new Card();
    Random rd = new Random();

    int rand_num = rd.Next(11111111, 99999999);
    card.CardNumber = "41005678" + rand_num;
    card.Email = email;

    var newCard = await _cardRepository.Create(card);
    return "Your new credit card is: " + newCard.CardNumber;
}
```

O terceiro é o PutCard que edita um cartão, recebendo do body o id do Card e os dados a serem alterados. Ele verifica se o card existe e chama o _cardRepository.Update mandando o card para alterar o Card, não é retornado conteúdo.

```
[HttpPut]
0 referências
public async Task<ActionResult> PutCard(int id, [FromBody] Card card)
{
    if (id != card.Id)
    {
        return BadRequest();
    }

    await _cardRepository.Update(card);
    return NoContent();
}
```

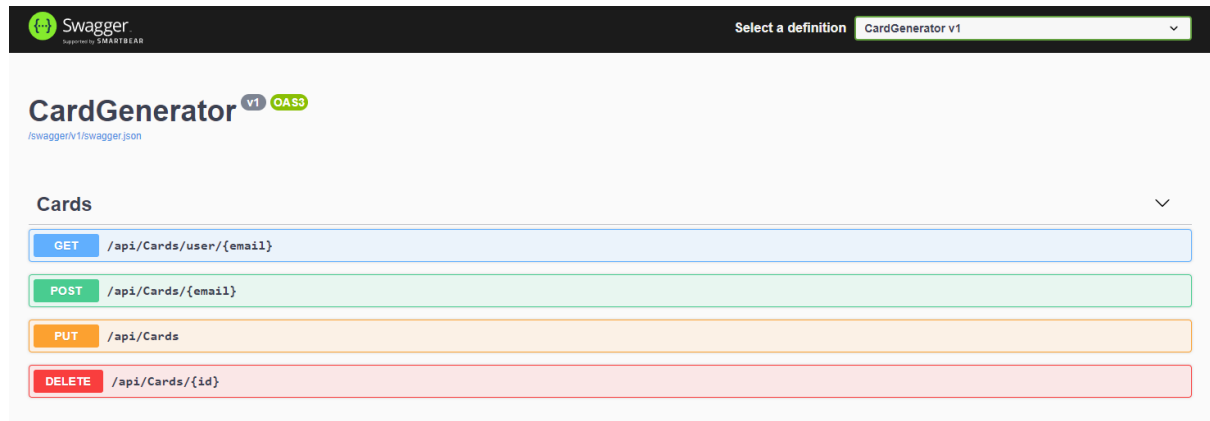
Por último temos o Delete, que deleta um Card recebendo o id do Card a ser deletado. Primeiro ele busca o cartão com o _cardRepository.Get mandando o id e verifica se não é um objeto nulo de retorno, caso exista ele chama o _cardRepository.Delete mandando o Id e deleta o Card, sem conteúdo de retorno.

```
[HttpDelete("{id}")]
0 referências
public async Task<ActionResult> Delete(int id)
{
    var cardToDelete = await _cardRepository.Get(id);

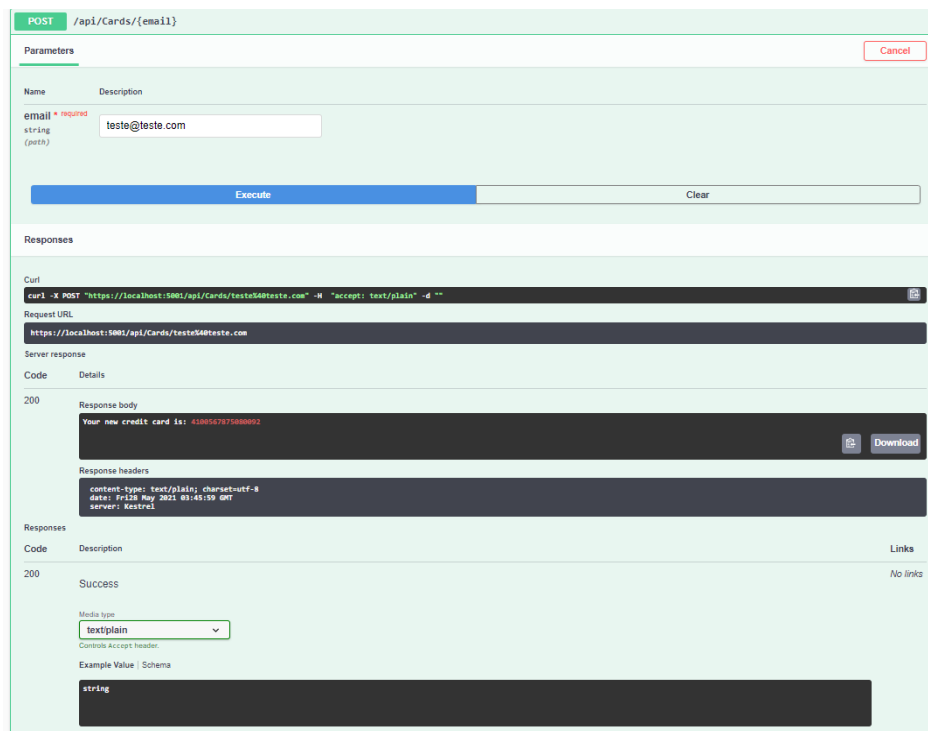
    if (cardToDelete == null)
    {
        return NotFound();
    }

    await _cardRepository.Delete(cardToDelete.Id);
    return NoContent();
}
```

Pronto, já possuímos a API pronta para ser compilada. Como selecionamos no início, nossa API possui suporte ao OPEN API que nos dá suporte ao Swagger com uma ferramenta que gera a documentação automática para nossa API com uma página de teste no browser. Compile e rode o projeto que a página do swagger vai ser aberta automaticamente com os endpoint disponíveis para ser testado.



Pronto, seu API está pronta e disponível para uso. Os dados ficam persistidos em um arquivo .db no projeto e mesmo fechando o projeto e rodando novamente se mantém.



Este tutorial foi escrito por [Jessé Monteiro Ferreira](#) inspirado nas seguintes fontes:

[How to create a Web API with ASP.NET Core | C# tutorial for beginners](#)

[Criando e consumindo uma aplicação ASP.NET Web API](#)

[Tutorial: criar uma API Web com o ASP.NET Core](#)