

## Como criar uma API REST com C# e .NET Core

Para criar o projeto, usando o Visual Studio Code, temos que usar a linha de comando NET CLI. Com o .NET core 5.0.3 instalado no nosso computador podemos usar o terminal de comando NET CLI para criar o projeto. Digite “dotnet --version” para verificar se o ambiente está apto para programar.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos os direitos reservados.

Experimente a nova plataforma cruzada PowerShell https://aka.ms/pscore6

PS C:\Users\Vitor> dotnet --version
5.0.300
PS C:\Users\Vitor>
```

Agora temos que criar a pasta para a API. Para isso, no terminal, digite os seguintes comandos.

```
PS C:\Users\Vitor> md src

Diretório: C:\Users\Vitor

Mode                LastWriteTime         Length Name
----                -
d-----         28/05/2021   17:30             src
```

“md src” para criarmos uma pasta com o nome “src”.

```
PS C:\Users\Vitor> cd src
PS C:\Users\Vitor\src>
```

“cd src” para acessar a pasta “src” criada no tópico anterior.

```

PS C:\Users\Vitor\src> md GeradorDeCartaoVirtual

Diretório: C:\Users\Vitor\src

Mode                LastWriteTime         Length Name
----                -
d-----         28/05/2021    17:40             GeradorDeCartaoVirtual

PS C:\Users\Vitor\src> cd GeradorDeCartaoVirtual

```

“md GeradorDeCartaoVirtual” para criar uma pasta e “cd GeradorDeCartaoVirtual” para acessá-la.

```

PS C:\Users\Vitor\src\GeradorDeCartaoVirtual> dotnet new webapi
O modelo "ASP.NET Core Web API" foi criado com êxito.

Processando ações pós-criação...
Executando 'dotnet restore' em C:\Users\Vitor\src\GeradorDeCartaoVirtual\GeradorDeCartaoVirtual.csproj...
Determinando os projetos a serem restaurados...
C:\Users\Vitor\src\GeradorDeCartaoVirtual\GeradorDeCartaoVirtual.csproj restaurado (em 1,01 sec).
A restauração foi bem-sucedida.

PS C:\Users\Vitor\src\GeradorDeCartaoVirtual> 

```

Para criar um projeto com template API digite “dotnet new webapi”.

```

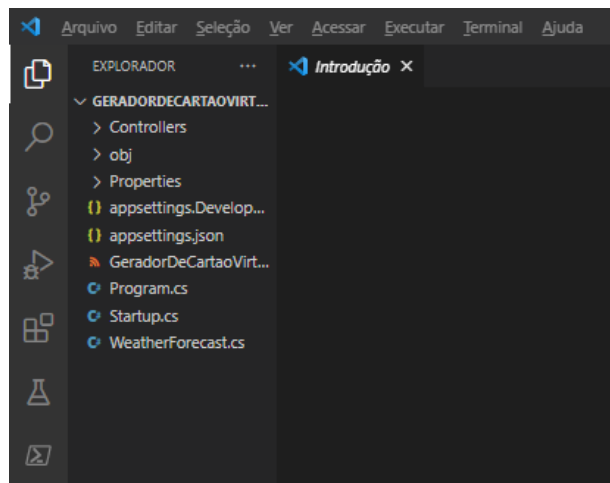
PS C:\Users\Vitor\src\GeradorDeCartaoVirtual> dir

Diretório: C:\Users\Vitor\src\GeradorDeCartaoVirtual

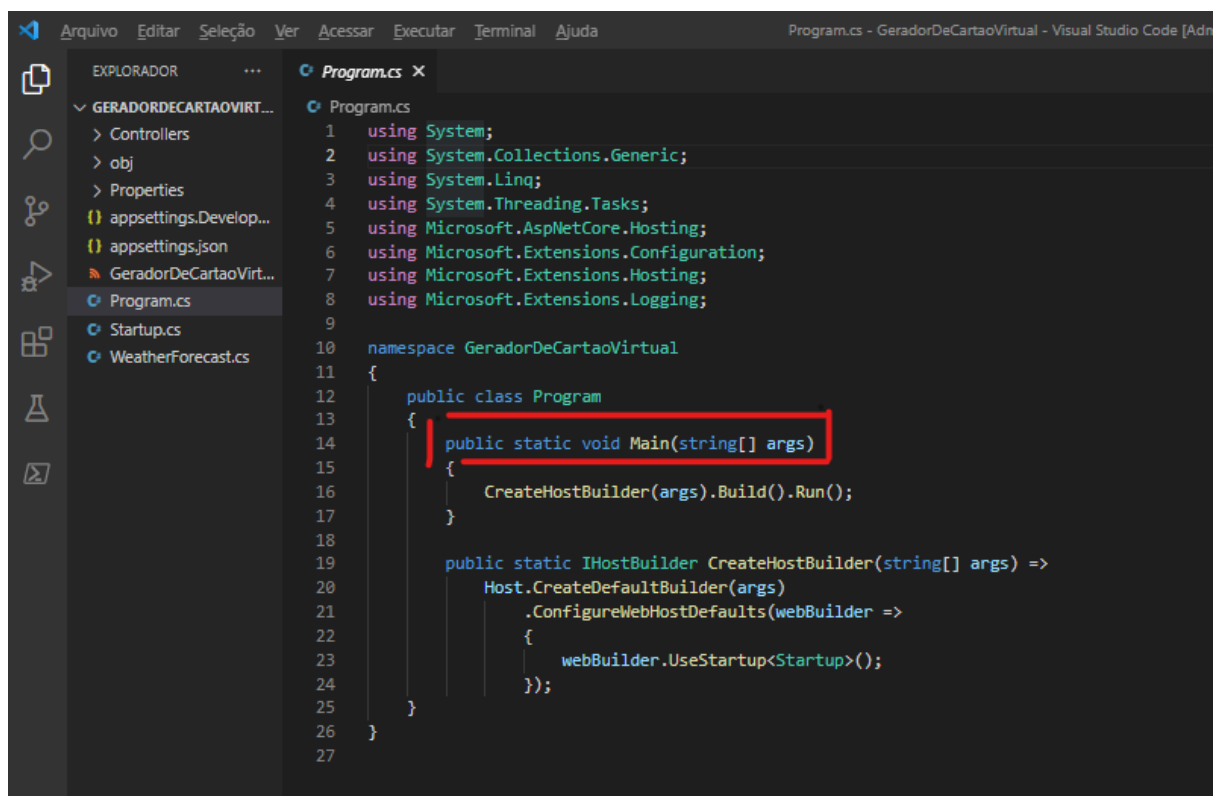
Mode                LastWriteTime         Length Name
----                -
d-----         28/05/2021    17:47             Controllers
d-----         28/05/2021    17:47             obj
d-----         28/05/2021    17:47             Properties
-a----         28/05/2021    17:47           162 appsettings.Development.json
-a----         28/05/2021    17:47           192 appsettings.json
-a----         28/05/2021    17:47           249 GeradorDeCartaoVirtual.csproj
-a----         28/05/2021    17:47           730 Program.cs
-a----         28/05/2021    17:47          1840 Startup.cs
-a----         28/05/2021    17:47           318 WeatherForecast.cs

```

Para ver toda a estrutura criada da pasta, digite “dir”.



Para abrir o diretório com vs code, digite “code” no seu terminal.



Abrindo a classe “Program.cs”, vemos que é nela que há o método Main, ou seja, o método principal. Ele cria o host da Web, por padrão, é a porta 5000 para Http e 5001 para Https.

## CRIANDO NOSSAS PASTAS DE ACESSO

Essa pasta vai conter nossas classes que vão representar nosso gerador de cartão de crédito virtual, além de interfaces de serviços. Digite os comandos abaixo:

“md PastaAcesso”, “cd PastaAcesso”, “md Models” e “cd Models”.

```
PS C:\Users\Vitor\src\GeradorDeCartaoVirtual> md PastaAcesso

Mode                LastWriteTime         Length Name
----                -
d-----          28/05/2021   18:13             PastaAcesso

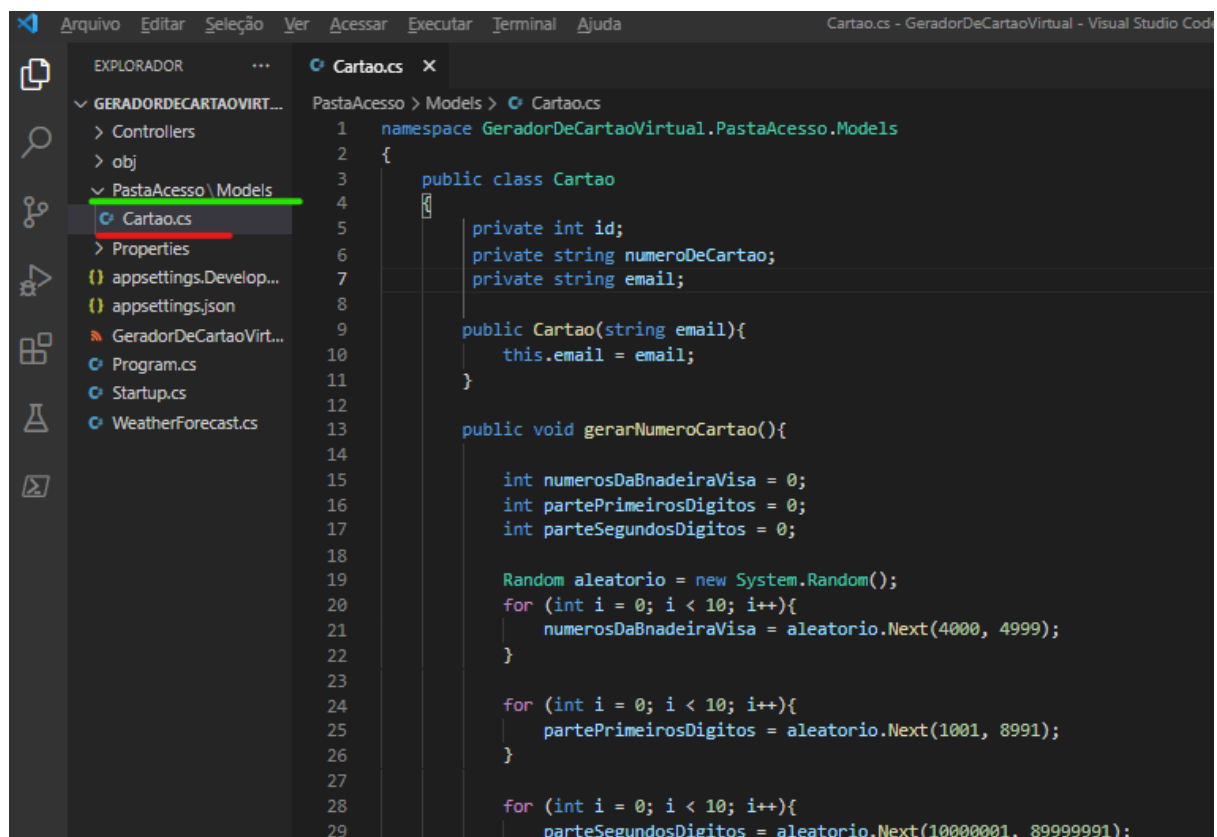
PS C:\Users\Vitor\src\GeradorDeCartaoVirtual> cd PastaAcesso
PS C:\Users\Vitor\src\GeradorDeCartaoVirtual\PastaAcesso> md Models

Mode                LastWriteTime         Length Name
----                -
d-----          28/05/2021   18:14             Models

PS C:\Users\Vitor\src\GeradorDeCartaoVirtual\PastaAcesso> cd Models
```

Todos os comandos devem ser digitados no seu terminal.

Dentro da pasta “Models” vamos criar uma classe chamada “Cartao”. Nela terá apenas os métodos “gerarCartao”, “MostrarCartaoGerado” e os getters e setters do Email e do Id.



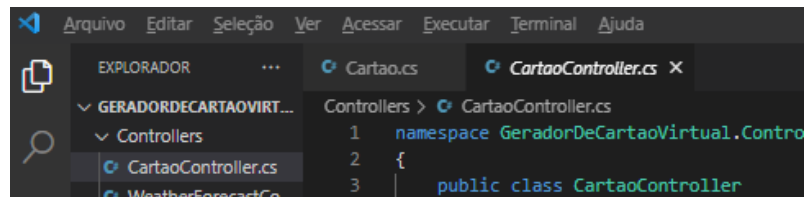
```
Arquivo  Editar  Seleção  Ver  Acessar  Executar  Terminal  Ajuda  Cartao.cs - GeradorDeCartaoVirtual - Visual Studio Code

EXPLORADOR  ...  Cartao.cs X
  GERADORDECARTAOVIRT...
    > Controllers
    > obj
    PastaAcesso > Models
      Cartao.cs
      > Properties
    appsettings.Develop...
    appsettings.json
    GeradorDeCartaoVirt...
    Program.cs
    Startup.cs
    WeatherForecast.cs

1  namespace GeradorDeCartaoVirtual.PastaAcesso.Models
2  {
3      public class Cartao
4      {
5          private int id;
6          private string numeroDeCartao;
7          private string email;
8
9          public Cartao(string email){
10             this.email = email;
11         }
12
13         public void gerarNumeroCartao(){
14
15             int numerosDaBnadeiraVisa = 0;
16             int partePrimeirosDigitos = 0;
17             int parteSegundosDigitos = 0;
18
19             Random aleatorio = new System.Random();
20             for (int i = 0; i < 10; i++){
21                 numerosDaBnadeiraVisa = aleatorio.Next(4000, 4999);
22             }
23
24             for (int i = 0; i < 10; i++){
25                 partePrimeirosDigitos = aleatorio.Next(1001, 8991);
26             }
27
28             for (int i = 0; i < 10; i++){
29                 parteSegundosDigitos = aleatorio.Next(10000001, 89999991);
```

## CRIANDO A API PARA CARTÃO

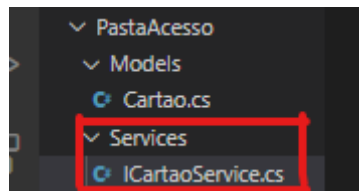
Na pasta “Controller”, crie a classe “CartaoController”, é nela que vamos definir a API para gerenciar informações de cartão.



Este novo controlador deve responder através da rota “/api/cartao” e para isso vamos adicionar o atributo Route acima do nome da classe.

```
namespace GeradorDeCartaoVirtual.Controllers
{
    [Route("/api/[controller]")]
    public class CartaoController : Controller
    {
    }
}
```

Agora, na pasta “PastaAcesso”, crie um novo diretório chamado Services. Nela adicione uma Interface chamada “ICartaoService”.



As implementações do método ListAsync devem retornar de forma assíncrona uma enumeração de categorias.

A classe Task, encapsulando o retorno, indica assincronia. Precisamos pensar em um método assíncrono devido ao fato de termos que esperar que o banco de dados conclua alguma operação para retornar os dados, e esse processo pode demorar um pouco. Observe também o sufixo “async”. É uma convenção que indica que nosso método deve ser executado de forma assíncrona.

```
Cartao.cs  CartaoController.cs  ICartaoService.cs X
PastaAcesso > Services > ICartaoService.cs
1  using System.Collections.Generic;
2  using System.Threading.Tasks;
3  using GeradorDeCartaoVirtual.PastaAcesso.Models;
4
5  namespace GeradorDeCartaoVirtual.PastaAcesso.Services
6  {
7      public interface ICartaoService
8      {
9
10         Task<IEnumerable<Cartao>> ListAsync();
11     }
12 }
13 }
```

Usando a interface podemos ter o comportamento desejado da implementação real. Usando um mecanismo conhecido como injeção de dependência, podemos implementar essas interfaces e isolá-las de outros componentes.

Vamos fazer isso. a seguir, na classe “CartaoController” onde vamos usar o serviço criado e realizar a injeção de dependências nesta classe:

```
Cartao.cs  CartaoController.cs X  ICartaoService.cs
Controllers > CartaoController.cs
1  using System.Collections.Generic;
2  using System.Threading.Tasks;
3  using Microsoft.AspNetCore.Mvc;
4  using GeradorDeCartaoVirtual.PastaAcesso.Models;
5  using GeradorDeCartaoVirtual.PastaAcesso.Services;
6
7  namespace GeradorDeCartaoVirtual.Controllers
8  {
9      [Route("/api/[controller]")]
10     public class CartaoController : Controller
11     {
12         private readonly ICartaoService _cartaoservice;
13
14         public CartaoController(ICartaoService cartaoservice){
15             _cartaoservice = cartaoservice;
16         }
17
18         [HttpGet]
19
20         public async Task<IEnumerable<Cartao>> GetAllAsync()
21         {
22             var cartao = await _cartaoservice.ListAsync();
23             return cartao;
24         }
25     }
```

Definimos uma função de construtor para nosso controlador (*um construtor é chamado quando uma nova instância de uma classe é criada*) e ele recebe uma instância de “ICartaoService”. Isso significa que a instância pode ser qualquer coisa que implemente a interface de serviço. Eu armazeno esta instância em um campo privado, somente leitura “\_cartaoservice”. Usaremos esse campo para acessar os métodos da implementação do nosso serviço de categoria.

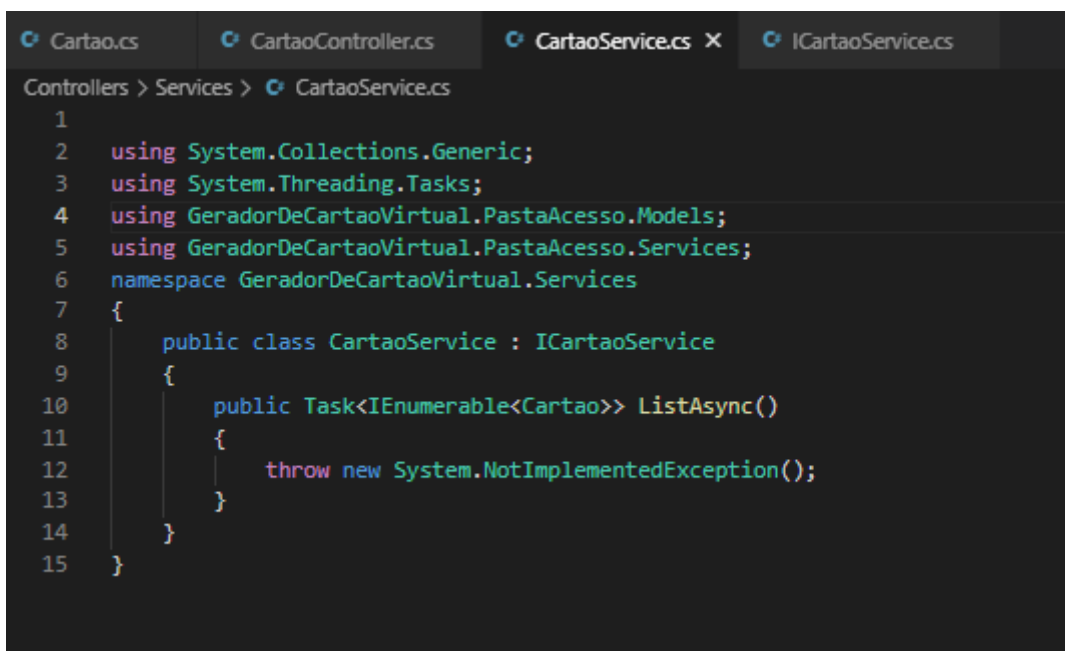
O atributo “HttpGet” diz ao pipeline do ASP.NET Core para usá-lo para lidar com solicitações GET (*esse atributo pode ser omitido, mas é melhor escrevê-lo para facilitar a legibilidade*).

O método usa nossa instância de serviço de cartão para listar todos os cartões e, em seguida, retorna os cartões para o cliente. O pipeline do framework lida com a serialização de dados para um objeto JSON.

## IMPLEMENTANDO O SERVIÇO PARA OBTERMOS OS CARTÕES

Precisamos implementar o serviço para retornar os cartões a partir da nossa interface “ICartaoService” usando uma classe concreta que vamos chamar de “CartaoService”.

Vamos criar uma pasta chamada “Service” dentro da pasta “GeradorDeCartaoVirtual”.



```
1
2 using System.Collections.Generic;
3 using System.Threading.Tasks;
4 using GeradorDeCartaoVirtual.PastaAcesso.Models;
5 using GeradorDeCartaoVirtual.PastaAcesso.Services;
6 namespace GeradorDeCartaoVirtual.Services
7 {
8     public class CartaoService : ICartaoService
9     {
10         public Task<IEnumerable<Cartao>> ListAsync()
11         {
12             throw new NotImplementedException();
13         }
14     }
15 }
```

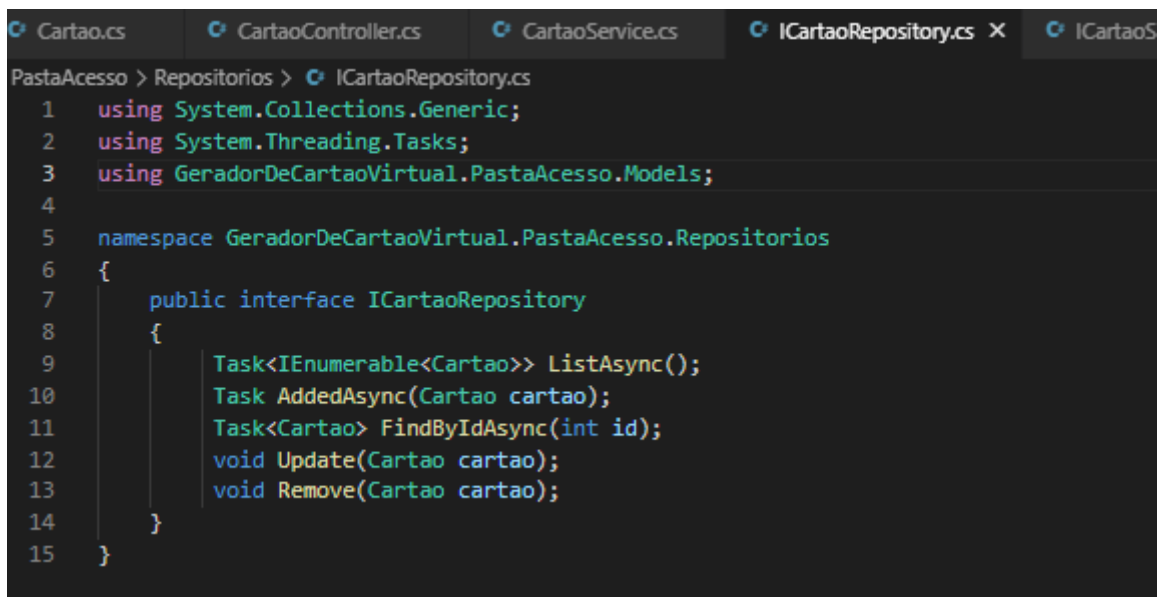
Acima temos o código básico para a implementação da interface. Precisamos acessar o banco de dados e retornar todos os cartões gerados, então precisamos

retornar esses dados para o cliente. Existe um padrão de repositório que é usado para gerenciar dados de banco de dados.

Nosso serviço precisa se comunicar com um repositório de cartão para obter a lista de objetos.

## O REPOSITÓRIO CARTÃO E O ACESSO DE DADOS

Crie uma pasta chamada “Repositórios” dentro da pasta “PastaAcesso” e depois crie uma interface com o nome “ICartaoRepository”. Agora reescreva o código abaixo na interface:

A screenshot of a Visual Studio code editor with a dark theme. The top of the editor shows several tabs: 'Cartao.cs', 'CartaoController.cs', 'CartaoService.cs', 'ICartaoRepository.cs' (which is the active tab), and 'ICartaoS...'. The breadcrumb navigation at the top of the editor area reads 'PastaAcesso > Repositorios > ICartaoRepository.cs'. The code in the editor is as follows:

```
1  using System.Collections.Generic;
2  using System.Threading.Tasks;
3  using GeradorDeCartaoVirtual.PastaAcesso.Models;
4
5  namespace GeradorDeCartaoVirtual.PastaAcesso.Repositorios
6  {
7      public interface ICartaoRepository
8      {
9          Task<IEnumerable<Cartao>> ListAsync();
10         Task AddedAsync(Cartao cartao);
11         Task<Cartao> FindByIdAsync(int id);
12         void Update(Cartao cartao);
13         void Remove(Cartao cartao);
14     }
15 }
```

Abra a classe “CartaoService” da pasta “service” da raiz do projeto e inclua o código a seguir:



```
using System.Collections.Generic;
using System.Threading.Tasks;
using GeradorDeCartaoVirtual.PastaAcesso.Models;
using GeradorDeCartaoVirtual.PastaAcesso.Services;
using GeradorDeCartaoVirtual.PastaAcesso.Repositorios;
namespace GeradorDeCartaoVirtual.Services
{
    public class CartaoService : ICartaoService
    {
        private readonly ICartaoRepository _cartaoRepository;

        public CartaoService(ICartaoRepository cartaoRepository)
        {
            _cartaoRepository = cartaoRepository;
        }

        public async Task<IEnumerable<Cartao>> ListAsync()
        {
            return await _cartaoRepository.ListAsync();
        }
    }
}
```

Agora temos que implementar a lógica real do repositório de cartões. Antes de fazer isso, temos que pensar em como vamos acessar o banco de dados.

Vamos usar o Entity Framework Core. Esse framework vem com o ASP.NET Core e expõe uma API amigável que nos permite mapear classes de nossos aplicativos para tabelas de banco de dados.

Na pasta raiz da nossa API, em “GeradorDeCartaoVirtual”, crie um novo diretório chamado “Persistence”. Este diretório terá tudo que precisamos para acessar o banco de dados, como implementações de repositórios. Dentro de “Persistence” crie uma classe chamada “AppDbContext”.

```
using Microsoft.EntityFrameworkCore;
using GeradorDeCartaoVirtual.Persistence;

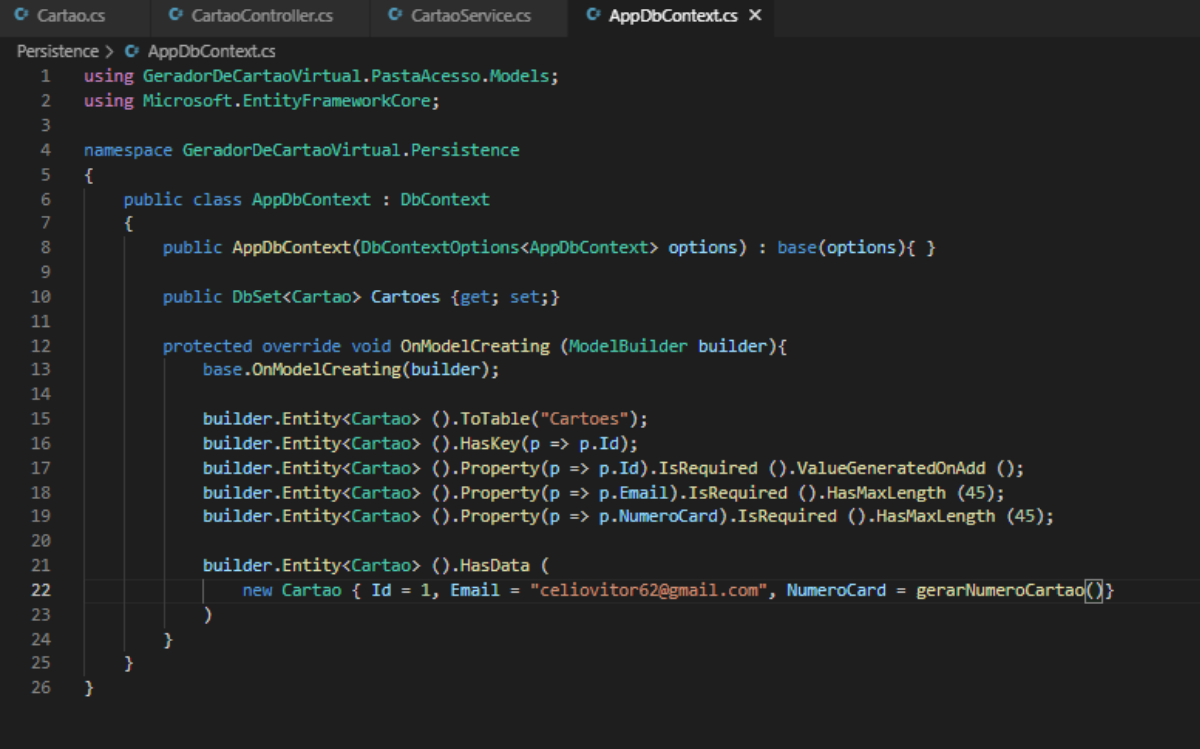
namespace GeradorDeCartaoVirtual.Persistence
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
        {
        }
    }
}
```

O construtor que adicionamos a essa classe é responsável por passar a configuração do banco de dados para a classe base através da injeção de dependência.

Agora, temos que criar duas propriedades DbSet. Essas propriedades são conjuntos que mapeiam modelos para tabelas de banco de dados.

Além disso, temos que mapear as propriedades dos modelos para as respectivas colunas da tabela, especificando quais propriedades são chaves primárias, estrangeiras e tipos da coluna.

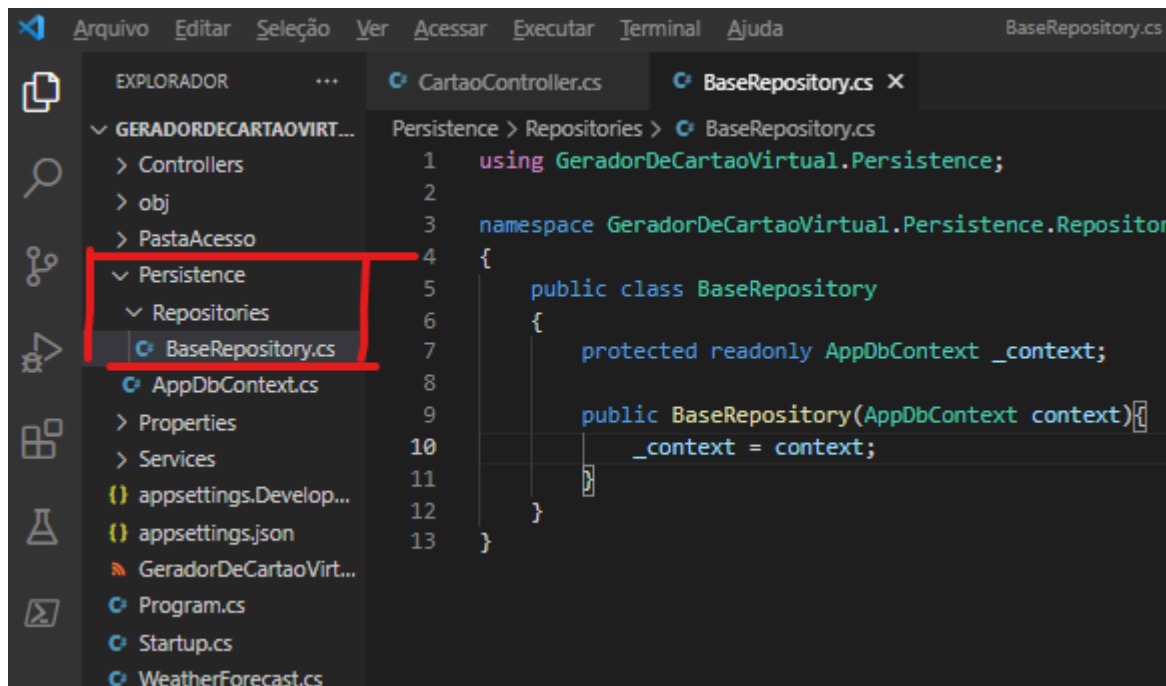
Altere a classe “AppDbContext” da seguinte maneira:



```
Persistence > AppDbContext.cs
1  using GeradorDeCartaoVirtual.PastaAcesso.Models;
2  using Microsoft.EntityFrameworkCore;
3
4  namespace GeradorDeCartaoVirtual.Persistence
5  {
6      public class AppDbContext : DbContext
7      {
8          public AppDbContext(DbContextOptions<AppDbContext> options) : base(options){ }
9
10         public DbSet<Cartao> Cartoes {get; set;}
11
12         protected override void OnModelCreating (ModelBuilder builder){
13             base.OnModelCreating(builder);
14
15             builder.Entity<Cartao> ().ToTable("Cartoes");
16             builder.Entity<Cartao> ().HasKey(p => p.Id);
17             builder.Entity<Cartao> ().Property(p => p.Id).IsRequired ().ValueGeneratedOnAdd ();
18             builder.Entity<Cartao> ().Property(p => p.Email).IsRequired ().HasMaxLength (45);
19             builder.Entity<Cartao> ().Property(p => p.NumeroCard).IsRequired ().HasMaxLength (45);
20
21             builder.Entity<Cartao> ().HasData (
22                 new Cartao { Id = 1, Email = "celiovitor62@gmail.com", NumeroCard = gerarNumeroCartao() }
23             )
24         }
25     }
26 }
```

Neste código definimos para quais tabelas nossos modelos devem ser mapeados. Além disso, definimos as chaves primárias, usando o método HasKey, as colunas da tabela, usando o método Property e algumas restrições.

Tendo implementado a classe context do banco de dados, podemos implementar o repositório de cartões. Adicione uma nova pasta chamada Repositories dentro da pasta Persistence e adicione uma nova classe chamada “BaseRepository”.



Esta classe é uma classe abstrata que todos os nossos repositórios herdarão.

A classe “BaseRepository” recebe uma instância do nosso ApplicationDbContext através da injeção de dependências e expõe uma propriedade protegida chamada “\_context”, que dá acesso a todos os métodos que precisamos para manipular as operações do banco de dados.

Adicione uma nova classe na mesma pasta chamada “CategoriaRepository”. Agora vamos realmente implementar a lógica do repositório:

```
1 using System.Collections.Generic;
2 using System.Threading.Tasks;
3 using Microsoft.EntityFrameworkCore;
4 using GeradorDeCartaoVirtual.PastaAcesso.Models;
5 using GeradorDeCartaoVirtual.PastaAcesso.Repositorios;
6 using using GeradorDeCartaoVirtual.Persistence;
7
8 namespace GeradorDeCartaoVirtual.Persistence.Repositories
9 {
10     public class CartaoRepository
11     {
12         public CartaoRepository(AppDbContext context) : base(context)
13         { }
14         public async Task<IEnumerable<Cartao>> ListAsync()
15         {
16             return await _context.Cartoes.ToListAsync();
17         }
18         public async Task AddAsync(Cartao cartao)
19         {
20             await _context.Cartoes.AddAsync(cartao);
21         }
22         public async Task<Cartao> FindByIdAsync(int id)
23         {
24             return await _context.Cartoes.FindAsync(id);
25         }
26         public void Remove(Cartao cartao)
27         {
28             _context.Cartoes.Remove(cartao);
29         }
30         public void Update(Cartao cartao)
31         {
32             _context.Cartoes.Update(cartao);
33         }
34     }
35 }
```

O repositório herda da classe “BaseRepository” e implementa a interface ICartaoRepository.

Assim, implementamos todos os métodos da interface ICartaoRepository e agora temos uma implementação limpa do controlador de Cartões, do serviço e do repositório.

A última etapa antes de testar o aplicativo é vincular nossas interfaces às respectivas classes usando o mecanismo de injeção de dependências do ASP.NET core.

## CONFIGURANDO A INJEÇÃO DE DEPENDÊNCIAS NATIVA

Na raiz do aplicativo, abra a classe Startup. Essa classe é responsável por configurar todos os tipos de configurações quando o aplicativo é iniciado.

Altere o código do método "Configure Services", acessando o parâmetro "services" . veja abaixo:

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "GeradorDeCartaoVirtual", Version = "v1" });
    });

    services.AddDbContext<AppDbContext>(options => {
        options.UseInMemoryDatabase("geradordecartaovirtual-in-memory");
    });
    services.AddScoped<ICartaoRepository, CartaoRepository>();
    services.AddScoped<ICartaoService, CartaoService>();
}
```

Agora que configuramos nossas vinculações de dependência, precisamos fazer uma pequena alteração na classe Program, para que o banco de dados propague corretamente os dados iniciais. Esta etapa é necessária apenas ao usar o provedor de banco de dados em memória.

Assim, abra o arquivo "Program" e altere o seu código conforme abaixo:

```

Program.cs
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNetCore.Hosting;
6  using Microsoft.Extensions.Configuration;
7  using Microsoft.Extensions.Hosting;
8  using Microsoft.Extensions.Logging;
9
10 namespace GeradorDeCartaoVirtual
11 {
12     public class Program
13     {
14         public static void Main(string[] args)
15         {
16             //CreateHostBuilder(args).Build().Run();
17             var host = BuildWebHost(args);
18             using(var scope = host.Services.CreateScope())
19             using(var context = scope.ServiceProvider.GetService<AppDbContext>())
20             {
21                 context.Database.EnsureCreated();
22             }
23             host.Run();
24         }
25
26         public static IHostBuilder CreateHostBuilder(string[] args) =>
27             WebHost.CreateDefaultBuilder(args)
28                 .UseStartup<Startup>()
29                 .Build();
30     }
31 }
32

```

Foi necessário alterar o método Main para garantir que nosso banco de dados seja "criado" quando o aplicativo for iniciado, já que estamos usando um provedor de memória. Sem essa alteração, as categorias que queremos incluir não serão criadas.

Por fim, é só testar. Coloque na pasta raiz do seu aplicativo o comando **dotnet run** e abra no seu navegador o link que foi dado. Normalmente <https://localhost:5001/api/cartao> ou <http://localhost:5000/api/cartao> .