



## **ASP.NET Core 3.1 com Entity Framework SQL Server**

*Profª Patrícia Gagliardo de Campos*  
*Disciplina: Desenvolvimento para Internet III*

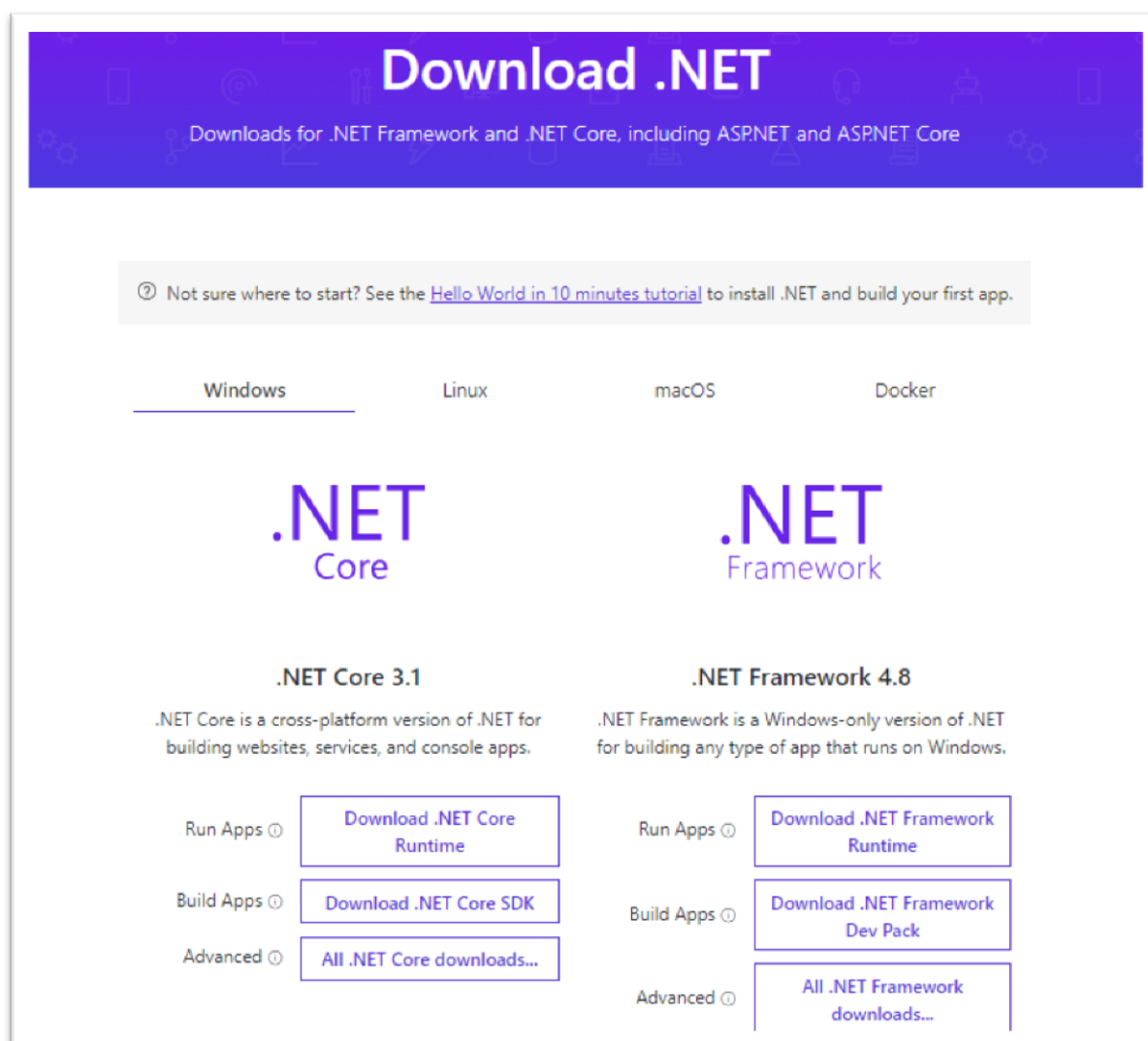
## Introdução

Nesse tutorial abordaremos um exemplo usando o **ASP.NET Core 3.1** para implementar um operação no banco de dados. Usaremos o **Entity Framework** para implementar as operações do CRUD.

Organizaremos o código através do **Repository Pattern**. O padrão Repository tem por objetivo abstrair a camada de acesso ao banco de dados, tornando transparente para a camada de negócio a tecnologia utilizada, como por exemplo, se o banco de dados é relacional como MsSQL, MySQL ou PostgreSQL, ou se é utilizado um banco de dados não relacional como MongoDB, Cosmos DB (Azure), Dynamo DB(Aws). Essa camada também será responsável por todas as operações de CRUD (Create, Read, Update, Delete), já a camada de serviço terá acesso a base dados a partir da implementação destes repositórios.<sup>1</sup>

## Recursos Necessários

O projeto que desenvolveremos será implementado via CLI (*Command Line Interface*). Para isso é necessário instalar o ASP.NET CORE que você pode baixar do site <https://dotnet.microsoft.com/download>:



<sup>1</sup> Fonte <https://medium.com/@adlerpagliarini/c-net-core-criando-uma-aplica%C3%A7%C3%A3o-utilizando-repository-pattern-com-dois-orms-diferentes-dapper-97e8aa6ca35>

Como comentado anteriormente, o ASP.NET Core é multiplataforma e permite que o desenvolvimento seja realizado em qualquer sistema operacional.

Na linha de comando (pelo CMD do Windows ou terminal do Visual Studio Code) o comando abaixo nos exibe um help com as opções par ao comando dotnet:

## dotnet -h

O comando abaixo nos exibe a lista de *templates* para projetos que podemos desenvolver com o ASP.NET. Esses *templates* nada mais são que estruturas de pastas e arquivos já organizados de acordo como tipo de projeto que queremos desenvolver.

No nosso caso, vamos criar uma API e para isso vamos informar o tipo do projeto na criação do mesmo.

## dotnet new -h

Uso: new [options]

Opções:

-h, --help

Displays help for this command.

-l, --list

Lists templates containing the specified name. If no name is specified, lists all templates.

-n, --name

The name for the output being created. If no name is specified, the name of the current directory is used.

-o, --output

Location to place the generated output.

-i, --install

Installs a source or a template pack.

-u, --uninstall

Uninstalls a source or a template pack.

--nuget-source

Specifies a NuGet source to use during install.

--type

Filters templates based on available types. Predefined values are "project", "item" or "other".

--dry-run

Displays a summary of what would happen if the given command line were run if it would result in a template creation.

--force

Forces content to be generated even if it would change existing files.

--lang, --language

Filters templates based on language and specifies the language of the template to create.

--update-check

Check the currently installed template packs for updates.

--update-apply

Check the currently installed template packs for update, and install the updates.

Templates	Short Name	Language	Tags
-----			
Console Application	console	[C#], F#, VB	Common/Console
Class library	classlib	[C#], F#, VB	Common/Library
WPF Application	wpf	[C#]	Common/WPF
WPF Class library	wpflib	[C#]	Common/WPF
WPF Custom Control Library	wpfcustomcontrollib	[C#]	Common/WPF
WPF User Control Library	wpfusercontrollib	[C#]	Common/WPF
Windows Forms (WinForms) Application	winforms	[C#]	Common/WinForms
Windows Forms (WinForms) Class library	winformslib	[C#]	Common/WinForms
Worker Service	worker	[C#]	Common/Worker/Web
Unit Test Project	mstest	[C#], F#, VB	Test/MSTest
NUnit 3 Test Project	nunit	[C#], F#, VB	Test/NUnit
NUnit 3 Test Item	nunit-test	[C#], F#, VB	Test/NUnit
xUnit Test Project	xunit	[C#], F#, VB	Test/xUnit
Razor Component	razorcomponent	[C#]	Web/ASP.NET
Razor Page	page	[C#]	Web/ASP.NET
MVC ViewImports	viewimports	[C#]	Web/ASP.NET
MVC ViewStart	viewstart	[C#]	Web/ASP.NET
Blazor Server App	blazorserver	[C#]	Web/Blazor
ASP.NET Core Empty	web	[C#], F#	Web/Empty
ASP.NET Core Web App (Model-View-Controller)	mvc	[C#], F#	Web/MVC
ASP.NET Core Web App	webapp	[C#]	Web/MVC/Razor Pages
ASP.NET Core with Angular	angular	[C#]	Web/MVC/SPA
ASP.NET Core with React.js	react	[C#]	Web/MVC/SPA
ASP.NET Core with React.js and Redux	reactredux	[C#]	Web/MVC/SPA
Razor Class Library	razorclasslib	[C#]	Web/Razor/Library/Razor Class Library
ASP.NET Core Web API	webapi	[C#], F#	Web/WebAPI
ASP.NET Core gRPC Service	grpc	[C#]	Web/gRPC
dotnet gitignore file	gitignore		Config
global.json file	globaljson		Config
NuGet Config	nugetconfig		Config
Dotnet local tool manifest file	tool-manifest		Config
Web Config	webconfig		Config
Solution File	sln		Solution
Protocol Buffer File	proto		Web/gRPC

Examples:

dotnet new mvc --auth Individual

dotnet new xunit

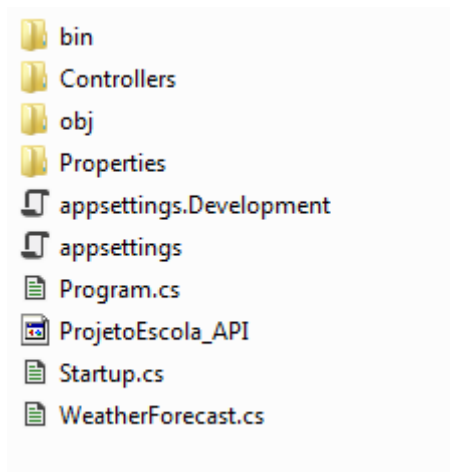
dotnet new --help

## Iniciando o projeto ASP.NET

Vamos iniciar o projeto ASP.NET Core 2 usando o CLI (*Command Line Interface* do ASP.NET). Usaremos o comando abaixo para criar o projeto:

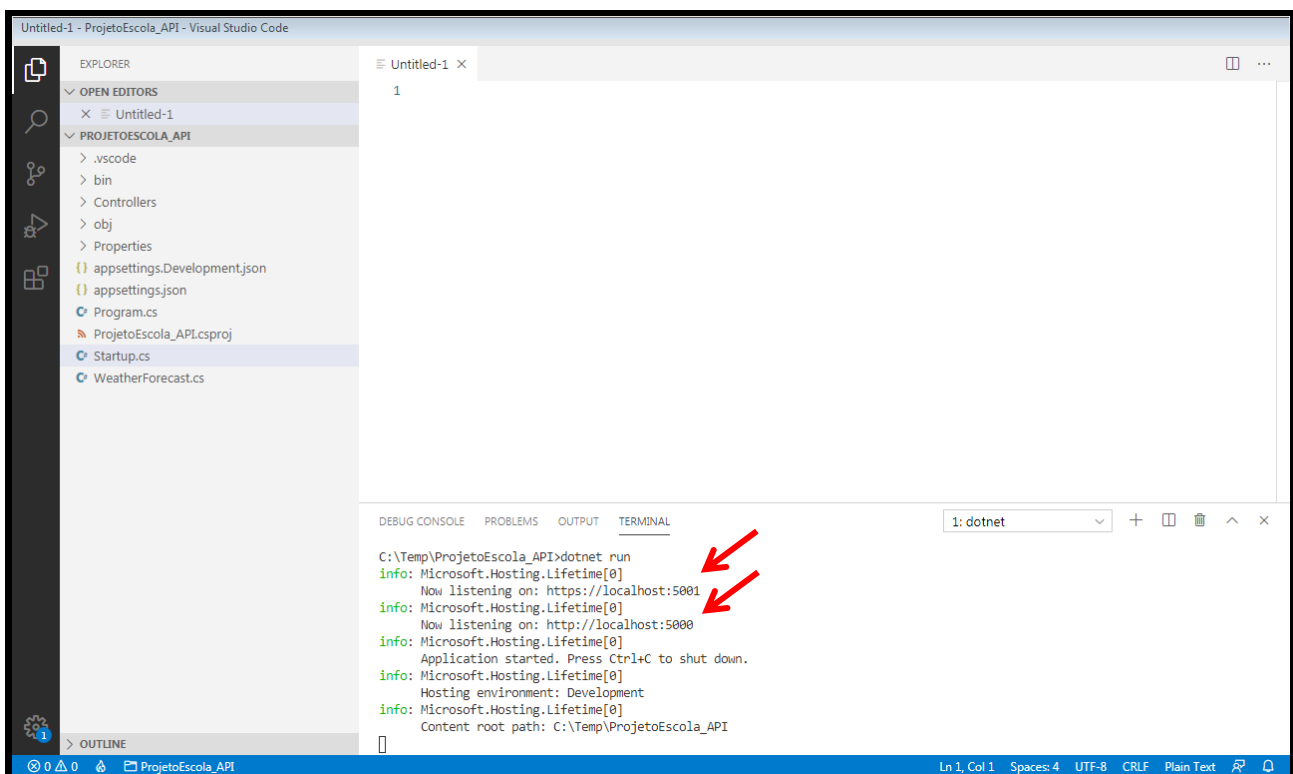
**dotnet new webapi -n ProjetoEscola\_API**

O comando acima irá criar uma pasta para o projeto com o seguinte conteúdo. Se a pasta **bin** e **obj** não estiver aparecendo, não se preocupe, pois sempre que você rodar a aplicação será feito um *building* e essas pastas e os respectivos arquivos (dll's) serão criados.

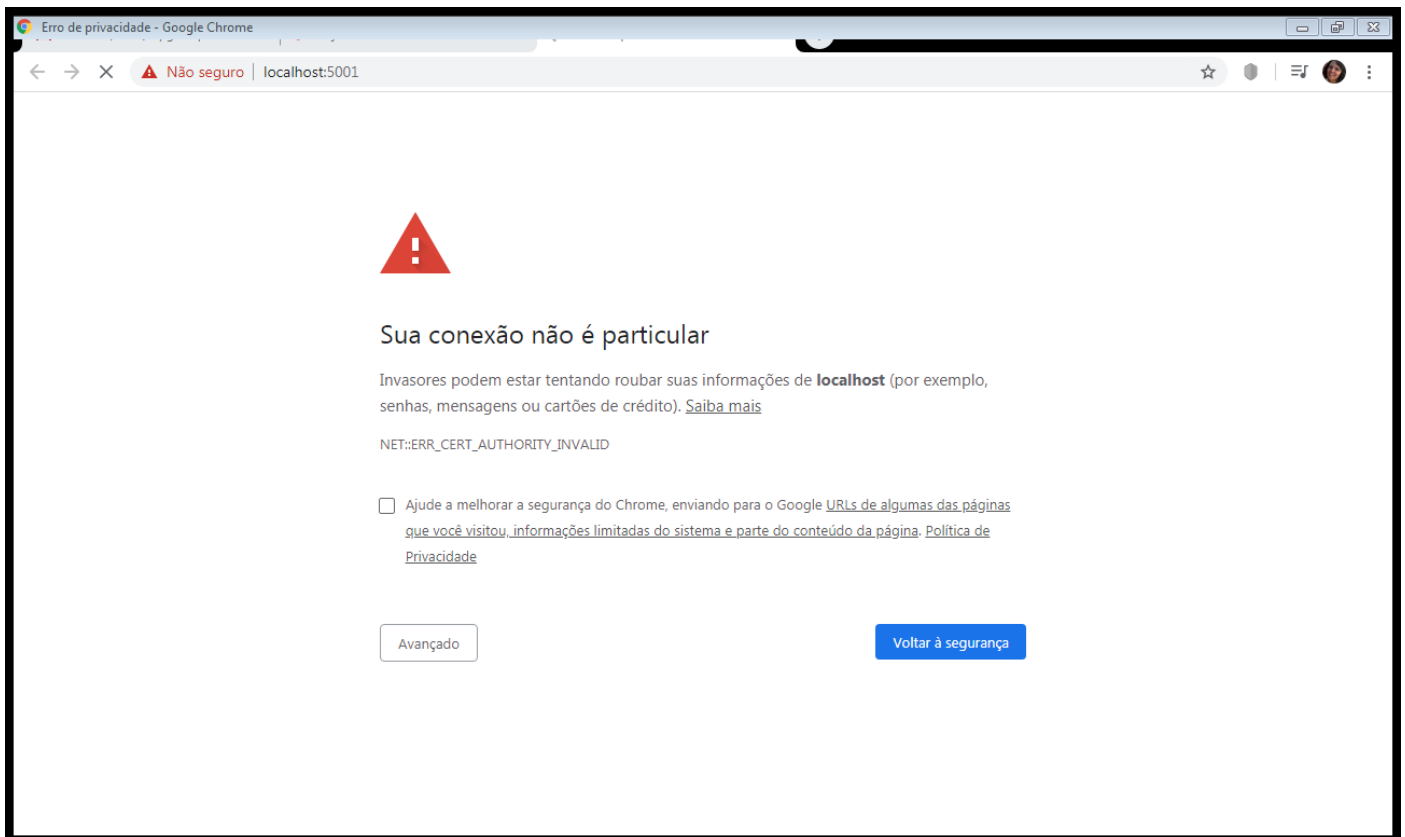


O comando abaixo irá rodar o projeto. Veja que no console algumas mensagens aparecerão informando sobre a porta onde a aplicação estará rodando:

**dotnet run**

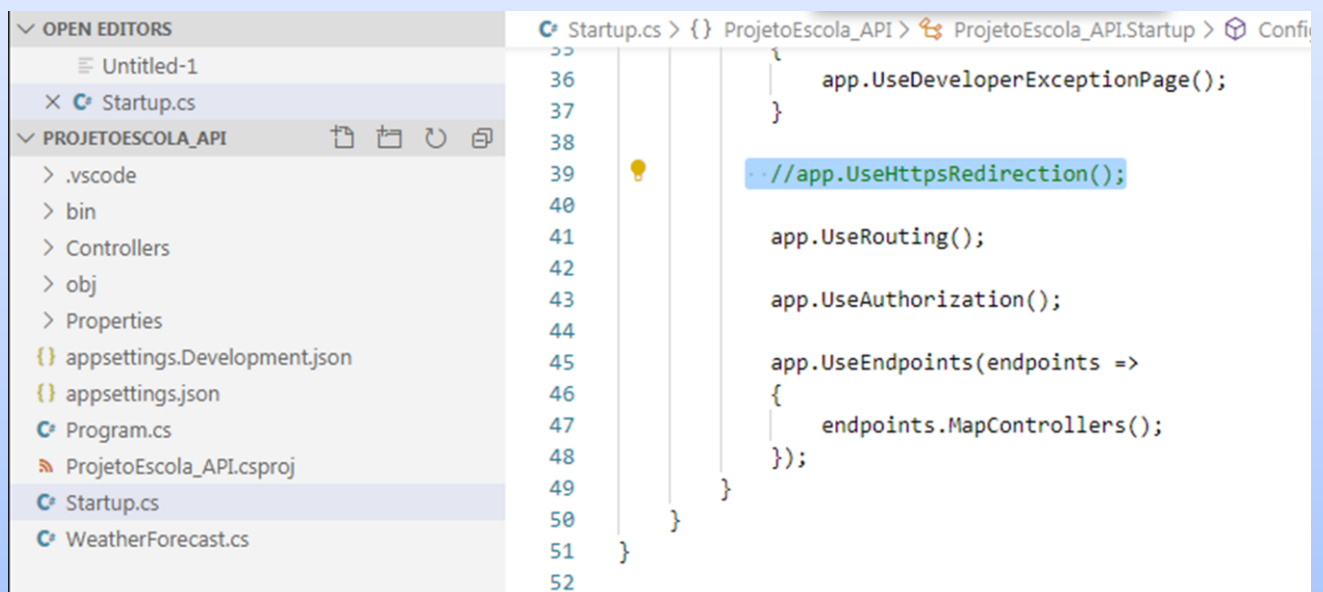


Se verificarmos um desses endereços no navegador temos a seguinte página:

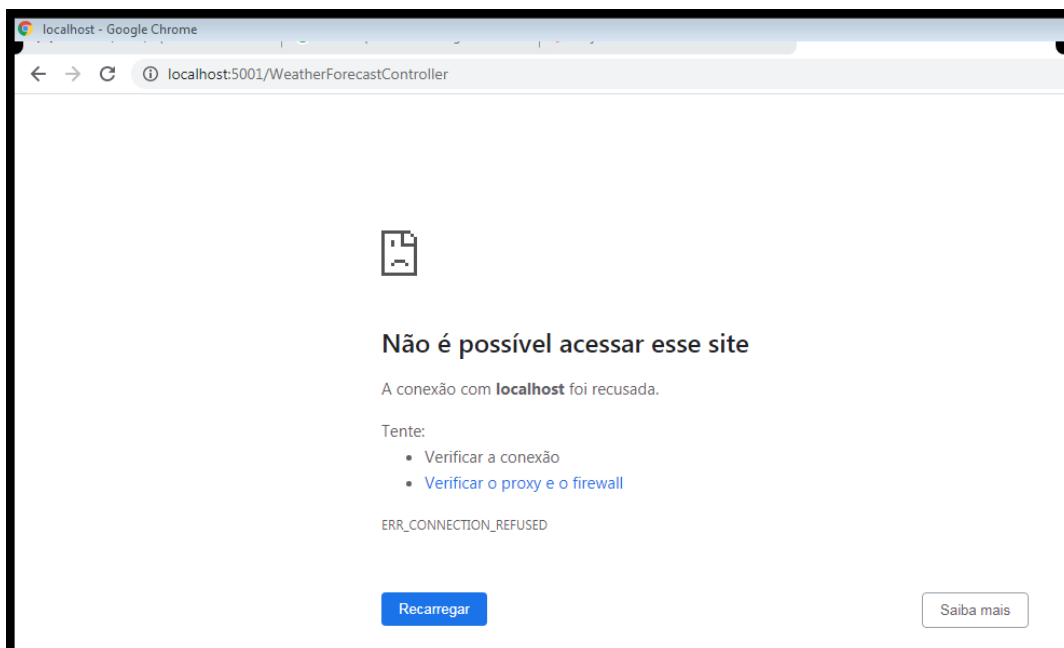


A princípio, pode aparecer uma tela informando um problema com o certificado da página. Isso acontece porque nesse caso estamos acessando a aplicação via HTTPS. Nesse caso é só clicar em avançado para continuar.

A aplicação pode estar configurada para sempre abrir através do HTTPS. Para alterar isso, basta alterar o arquivo **Startup.cs** comentando a linha referente a essa configuração. Como mostra no exemplo abaixo:



Nesse primeiro momento como não criamos nenhuma rota para a nossa aplicação a página não exibirá nenhuma informação. É normal!



## Entendendo a Estrutura da Aplicação

Quando rodamos a aplicação (através do comando no terminal **dotnet run** ou **CRTL+F5**), o primeiro arquivo que é executado é o **Program.cs**.

### Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace ProjetoEscola_SQLServer
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Veja que **Host** faz com que a aplicação rode em um servidor web (IIS ou Kestrel)

Aqui a classe Startup é chamada. Essa classe contém configurações da aplicação

```
}
```

### Startup.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace ProjetoEscola_SQLServer
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to
        the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllers();
        }

        // This method gets called by the runtime. Use this method to configure the HT
        TP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }


            //app.UseHttpsRedirection();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {

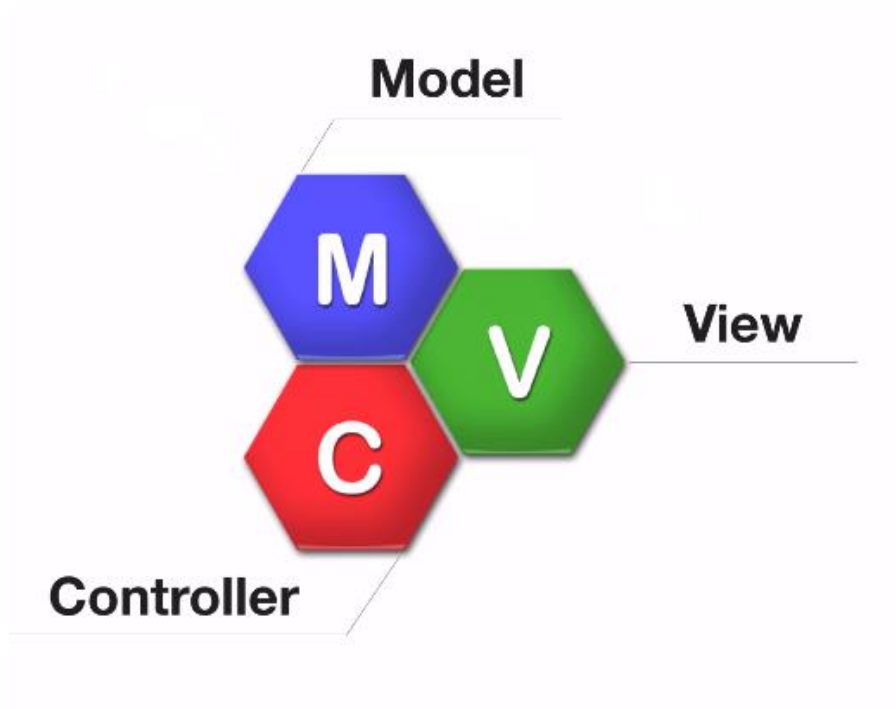
```



Aqui ocorre uma injeção de dependência, onde o ASP.NET detecta que este é um arquivo de configuração e carrega as opções definidas no projeto. Mais especificamente as configurações contidas no arquivo **launchSettings.json** dentro da pasta Properties

```
    endpoints.MapControllers();  
  });  
}  
}
```

## Revisando o conceito MVC





## Model-View-Controller

“ A camada de **Modelo** (MODEL) possui **duas responsabilidades**. Buscar os **dados** que sua aplicação usa e fazem parte da regra de negócio, **onde localiza-se cálculos**, procedimentos de verificações específicas em relação ao **domínio das regras do negócio**. ”



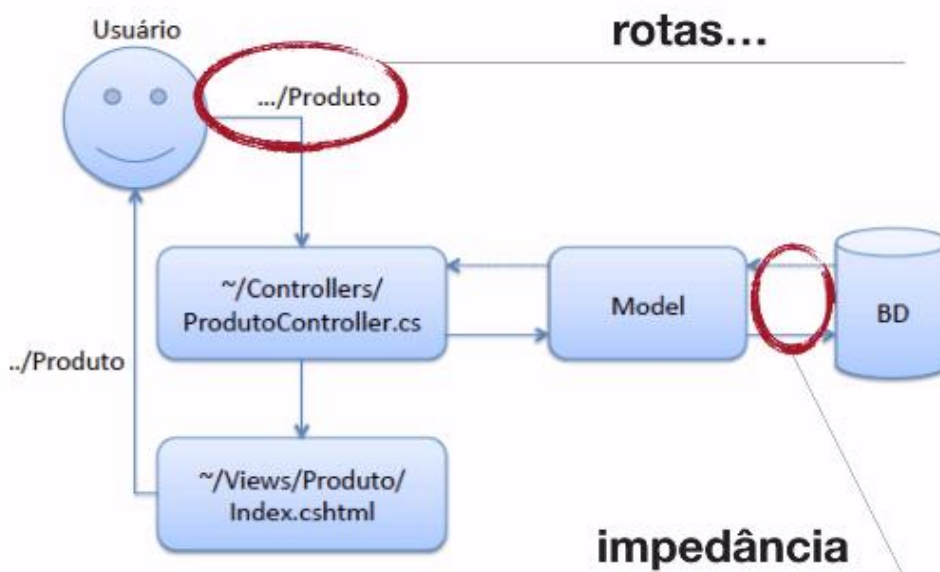
## Model-View-Controller

“ A camada de **Apresentação** (VIEW) é como o aplicativo mostra o resultado das operações e os dados. Normalmente podem ser uma bela página usando as novas tecnologias front-end, até representações de objetos em JSON ou XML. ”



# Model-View-Controller

“A camada de **Controle** (CONTROLLER) pode gerenciar **sessões e cookies**, além de ser a camada que fará o papel de determinar a melhor **maneira de representar/retornar os dados**, seja por meio de uma **renderização de página HTML** ou composição de um objeto em **JSON**.”



Extensões .NET para Visual Studio Code



C# ms-dotnettools.csharp

Microsoft | 6.827.295 | ★★★★★ | Repository | License

C# for Visual Studio Code (powered by OmniSharp).

[Disable](#) [Uninstall](#) This extension is enabled globally.



C# Extensions jchannon.csharpextensions

jchannon | 648.512 | ★★★★★ | Repository

C# IDE Extensions for VSCode

[Uninstall](#) This extension is enabled globally.

[Details](#) [Contributions](#) [Changelog](#)



NuGet Package Manager jmrog.vscodenuget-package-manager

jmrog | 317.970 | ★★★★★ | Repository | License

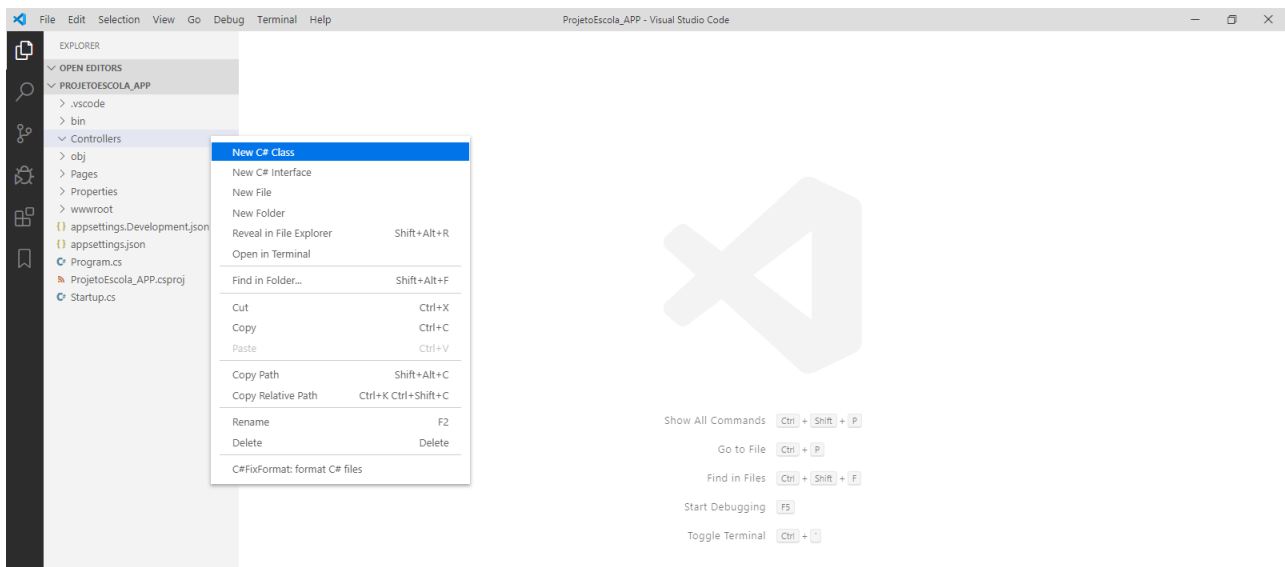
Add or remove .NET Core 1.1+ package references to/from your project's .csproj or .fsproj file using Code's Command Palette.

[Uninstall](#) This extension is enabled globally.

[Details](#) [Contributions](#) [Changelog](#)

## Controladores

Vamos criar a pasta na raiz do projeto **Controllers** e dentro dela uma nova classe C# chamada **AlunoController.cs**:



Por padrão, um **controller** (ou classe controladora) do C# deve ter o sufixo **Controller** no nome da classe e obrigatoriamente herdar a classe **Controller**. Todo controlador precisa ser colocado dentro da pasta **Controllers**. Essa é uma convenção para projetos que irão ter a estrutura MVC. Os métodos públicos dos controladores irão tratar as requisições web. Esses métodos são chamados de **Action**. No arquivo criado (**AlunoController.cs**) temos uma **Action** chamada **Index** que devolve uma resposta do tipo **ActionResult**<sup>2</sup>. Dentro do método vamos colocar o código da regra de negócio da aplicação.

Adicionaremos as seguintes linhas na nova classe:

Esses comandos em **colchetes** são atributos. Servem para configurar características do controlador.

```
AlunoController.cs
using Microsoft.AspNetCore.Mvc;

namespace ProjetoEscola_SQLServer.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AlunoController : Controller
    {
        public AlunoController(Parameters)
        {
        }
    }
}
```

Digitando **CTRL+.** sobre o nome da classe, o ASP acrescenta automaticamente a respectiva biblioteca (using ...)

Para adicionar um construtor, podemos usar o atalho digitando **ctor**

Nesse controlador, vamos adicionar os **métodos** para tratarmos as rotas:

```
AlunoController.cs
using Microsoft.AspNetCore.Mvc;

namespace ProjetoEscola_SQLServer.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AlunoController : Controller
    {
        public AlunoController()
        {
            // construtor
        }

        [HttpGet]
        public IActionResult Get()
        {
            return Ok();
        }

        [HttpGet("{AlunoRA}")]
        public IActionResult Get(string AlunoRA)
        {
        }
    }
}
```

A função **Ok()** pertence a classe **Controller** e retorna o **Status 200** do protocolo HTTP

<sup>2</sup> Consulte [Tipos de retorno de ação do controlador em ASP.NET Core API Web](#) no final do arquivo para saber outros tipos que os métodos podem retornar.

```

    {
        return Ok();
    }

    [HttpPost]
    public IActionResult post()
    {
        return Ok();
    }

    [HttpPut("{AlunoRA}")]
    public IActionResult put(string AlunoRA)
    {
        return Ok();
    }

    [HttpDelete("{AlunoRA}")]
    public IActionResult delete(string AlunoRA)
    {
        return Ok();
    }
}
}

```

## Models

Vamos criar uma pasta na raiz do projeto chamada **Models** e dentro dela uma classe chamada **Aluno.cs** para representar a entidade, ou seja, a tabela com os dados para usarmos na nossa aplicação.

### Aluno.cs

```

namespace ProjetoEscola_SQLServer.Models
{
    public class Aluno
    {
        public int Id { get; set; }
        public string RA { get; set; }
        public string Nome { get; set; }
        public int codCurso { get; set; }
    }
}

```

Para usarmos o Entity Framework toda tabela precisa ter o campo **Id**

## Criando Contexto com Entity Framework usando SQL Server

Para acessarmos o banco de dados vamos usar o **Entity Framework**. A ideia é usar os métodos do **Entity** para fazer toda a persistência de dados, ou seja, todas as operações no banco de dados.

Para usarmos o **Entity** é necessário criar uma classe onde vamos definir o banco de dados que usaremos e as respectivas tabelas.

Vamos criar uma pasta na raiz do projeto chamada **Data** para armazenar esse arquivo e chamaremos de **EscolaContext.cs**.

*Preste atenção se as pastas e arquivos criados estão seguindo as orientações de localização!*

Para usarmos o **Entity Framework** precisamos acrescentar a biblioteca no projeto. Para isso, no terminal executamos o seguinte comando:

**dotnet add package Microsoft.EntityFrameworkCore.SqlServer**

```
DataContext.cs
using Microsoft.EntityFrameworkCore;

namespace ProjetoEscola_SQLServer.Data
{
    public class EscolaContext: DbContext
    {
    }
}
```

Nessa classe definiremos o contexto de dados na nossa aplicação, assim como as tabelas. Nesse caso, só teremos a tabela Alunos, por isso nossa classe ficará da seguinte forma:

```
EscolaContext.cs
using Microsoft.EntityFrameworkCore;
using ProjetoEscola_SQLServer.Models;

namespace ProjetoEscola_SQLServer.Data
{
    public class EscolaContext: DbContext
    {
        public EscolaContext(DbContextOptions<EscolaContext> options): base (options)
        {
        }

        public DbSet<Aluno> Aluno {get; set;}
    }
}
```

Precisamos informar ao projeto que usaremos o banco **SQL Server** e isso fazemos no arquivo **Startup.cs** acrescentando as linhas abaixo:

```
Startup.cs
...
using Microsoft.EntityFrameworkCore;
using ProjetoEscola_SQLServer.Data;
```

```

namespace ProjetoEscola_SQLServer
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to
        the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<EscolaContext>(
                x => x.UseSqlServer(Configuration.GetConnectionString("StringConexaoSQLServer"))
            );
            services.AddControllers();
        }
    }
}
...

```

**EscolaContext** refere-se ao contexto criado na classe **EscolaContext.cs**

No arquivo **appsettings.Development.json** precisamos declarar essa string de conexão como abaixo:

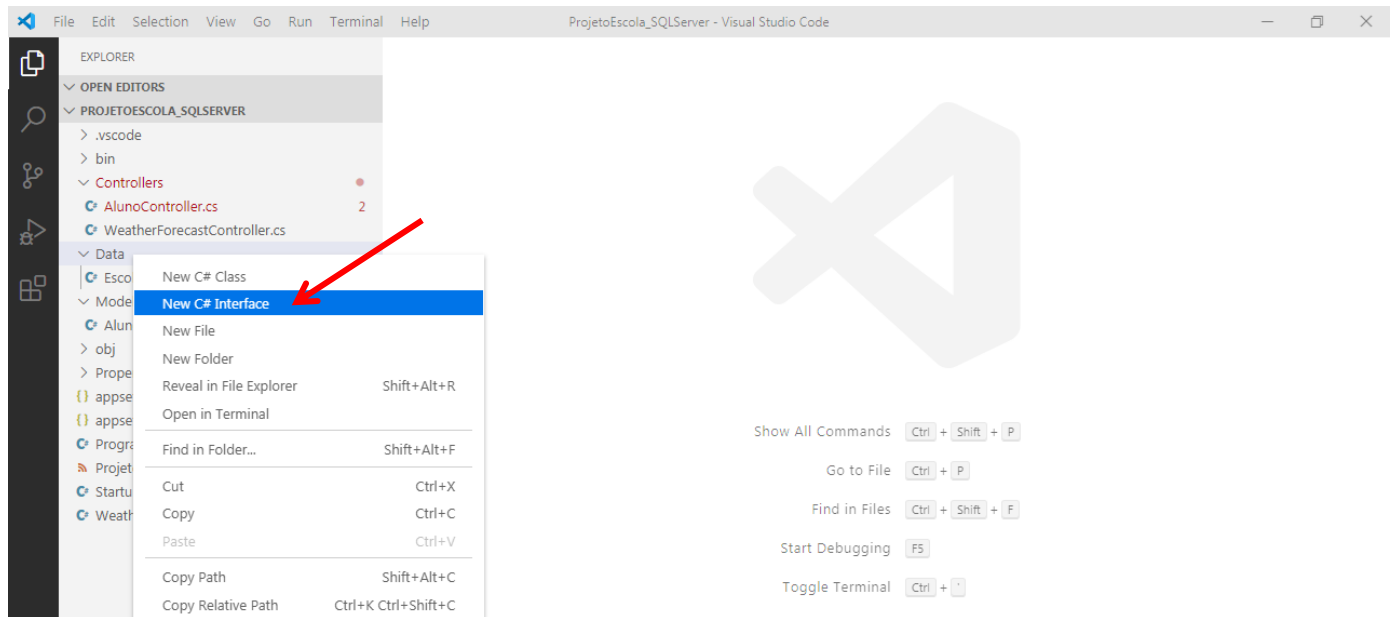
```

appsettings.Development.json
{
  "ConnectionStrings": {
    "StringConexaoSQLServer": "Data Source=regulus.cotuca.unicamp.br;Initial Catalog=p
    atricia;User ID=BDXXXXX;Password=sua-senha"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}

```

## Repositórios

Na pasta **Data**, vamos criar duas classes para representar nossos dados, através do conceito de repositórios. A primeira será uma classe de interface e vamos chama-la de **IRepository.cs** e a outra classe será a implementação do que definirmos na classe de interface e a chamaremos de **Repository.cs**.



### IRepository.cs

```
namespace ProjetoEscola_SQLServer.Data
{
    public interface IRepository
    {
    }
}
```

Em seguida, vamos criar a classe **Repository.cs** que implementará o que definirmos na interface:

### Repository.cs

```
namespace ProjetoEscola_SQLServer.Data
{
    public class Repository: IRepository
    {
    }
}
```

Na classe **IRepository.cs** vamos criar inicialmente classes genéricas para as operações básicas do CRUD. Sendo que esses métodos receberão um parâmetro genérico que identificaremos como T:

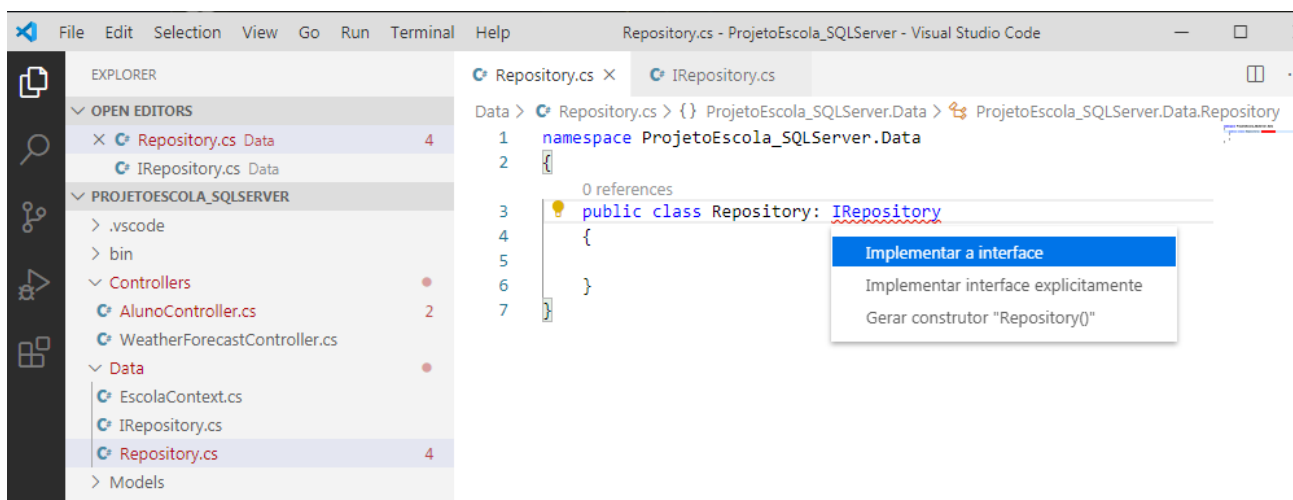


## IRepository.cs

```
using System.Threading.Tasks; // teclar CTRL+. sobre o tipo Task para acrescentar biblioteca

namespace ProjetoEscola_SQLServer.Data
{
    public interface IRepository
    {
        // Métodos genéricos
        void Add<T>(T entity) where T : class;
        void Update<T>(T entity) where T : class;
        void Delete<T>(T entity) where T : class;
        Task<bool> SaveChangesAsync();
    }
}
```

Agora vamos implementar as interfaces em **Repository.cs**. Ao teclar **CTRL+.** sobre a dependência **IRepository**, aparece uma opção se desejamos implementar a interface. Ao selecionar essa opção, automaticamente são construídos os respectivos métodos.



## Repository.cs

```
using System.Threading.Tasks;

namespace ProjetoEscola_SQLServer.Data
{
    public class Repository : IRepository
    {
        public void Add<T>(T entity) where T : class
        {
            throw new System.NotImplementedException();
        }

        public void Delete<T>(T entity) where T : class
        {
            throw new System.NotImplementedException();
        }

        public Task<bool> SaveChangesAsync()
        {
        }
    }
}
```

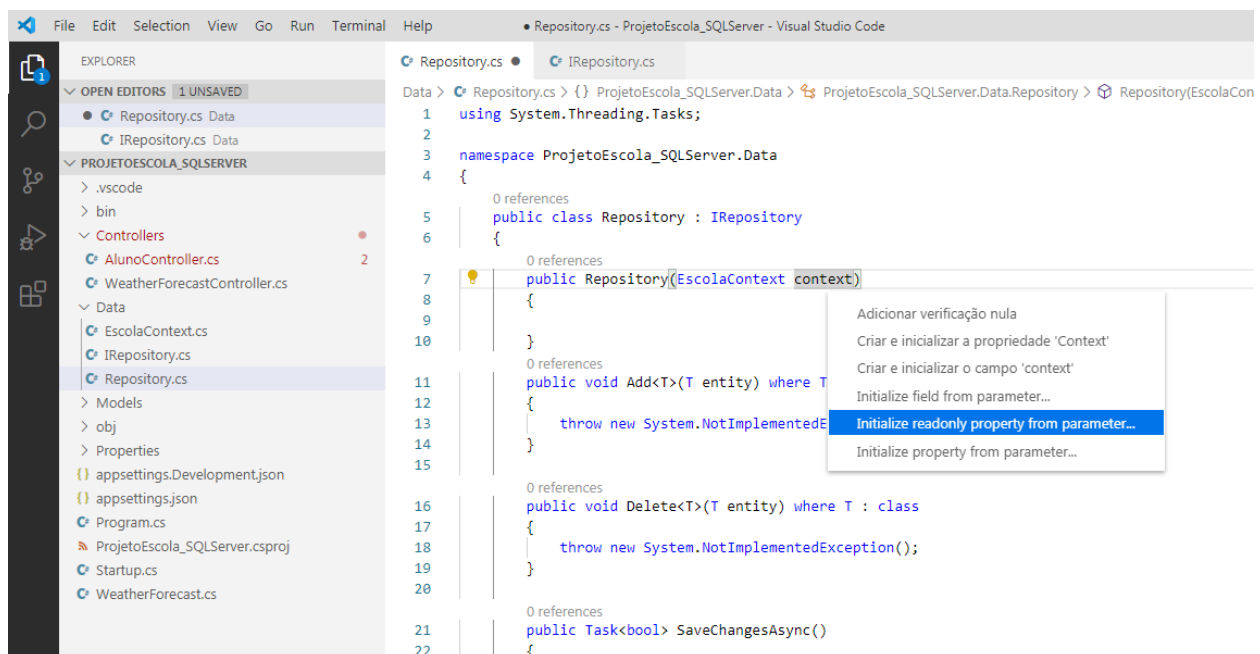
```

        throw new System.NotImplementedException();
    }

    public void Update<T>(T entity) where T : class
    {
        throw new System.NotImplementedException();
    }
}

```

Na classe **Repository.cs** temos que criar um construtor que irá receber nosso *context*, ou seja, as informações do banco de dados que o projeto usará.



A ideia de implementar as operações de acesso ao banco de dados via repositório tem o intuito de deixar o código mais organizado e enxuto. Veja como ficou a classe com as implementações:

### Repository.cs

```

using System.Threading.Tasks;

namespace ProjetoEscola_SQLServer.Data
{
    public class Repository : IRepository
    {
        public EscolaContext Context { get; }
        public Repository(EscolaContext context)
        {
            this.Context = context;
        }

        public void Add<T>(T entity) where T : class
        {
            //throw new System.NotImplementedException();
            this.Context.Add(entity);
        }
    }
}

```

```

    }

    public void Delete<T>(T entity) where T : class
    {
        //throw new System.NotImplementedException();
        this.Context.Remove(entity);
    }

    public async Task<bool> SaveChangesAsync()
    {
        // Como é tipo Task vai gerar thread, então vamos definir o método como assíncrono (async)
        // Por ser assíncrono, o return deve esperar (await) se tem alguma coisa para salvar no BD
        // Ainda verifica se fez alguma alteração no BD, se for maior que 0 retorna true ou false
        return(await this.Context.SaveChangesAsync() > 0);
    }

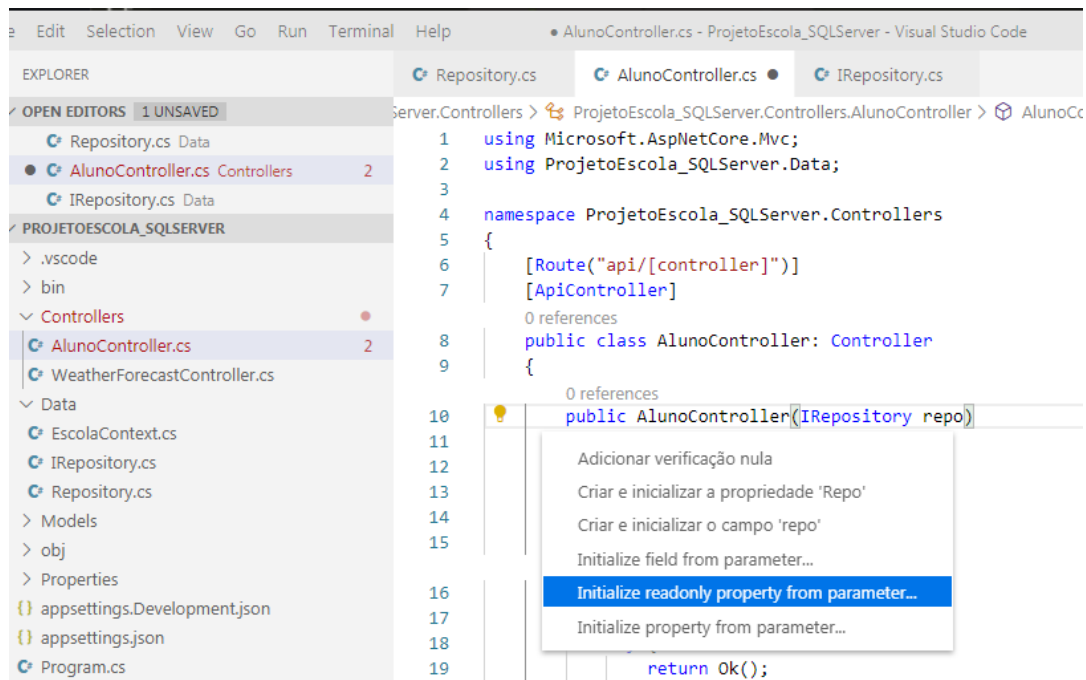
    public void Update<T>(T entity) where T : class
    {
        //throw new System.NotImplementedException();
        this.Context.Update(entity);
    }
}
}

```

Com o código acima, temos a implementação das operações de inserção, remoção e atualização para qualquer entidade do projeto.

## Ajustando o Controlador

Vamos fazer os ajustes necessários começando por “informar” que vamos usar uma interface para o repositório na declaração do construtor.



Vamos acrescentar um **try-catch** para cada método e tratar mensagens de erro:

### AlunoController.cs

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ProjetoEscola_SQLServer.Data;

namespace ProjetoEscola_SQLServer.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AlunoController : Controller
    {
        public IRepository Repo { get; }
        public AlunoController(IRepository repo)
        {
            this.Repo = repo;
            //construtor
        }

        [HttpGet]
        public IActionResult Get()
        {
            try
            {
                return Ok();
            }
            catch
            {
            }
        }
    }
}
```

```

        {
            return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
        }

    }

    [HttpGet("{AlunoRA}")]
    public IActionResult Get(string AlunoRA)
    {
        try
        {
            return Ok();
        }
        catch
        {
            return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
        }
    }

    [HttpPost]
    public IActionResult post()
    {
        try
        {
            return Ok();
        }
        catch
        {
            return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
        }
    }

    [HttpPut("{AlunoRA}")]
    public IActionResult put(string AlunoRA)
    {
        try
        {
            return Ok();
        }
        catch
        {
            return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
        }
    }

    [HttpDelete("{AlunoRA}")]
    public IActionResult delete(string AlunoRA)
    {

```

```

        try
        {
            return Ok();
        }
        catch
        {
            return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
        }
    }
}
}

```

## Implementando o método Add no controlador

Vamos começar pelo método **Post**. Vamos passar um objeto Aluno e vamos fazer que esse método seja assíncrono (async), pois várias aplicações poderão chamar esse método simultaneamente.

### Repository.cs

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ProjetoEscola_SQLServer.Data;
using ProjetoEscola_SQLServer.Models;

namespace ProjetoEscola_SQLServer.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AlunoController : Controller
    {
        public IRepository Repo { get; }
        public AlunoController(IRepository repo)
        {
            this.Repo = repo;
            //construtor
        }

        [HttpGet]
        public IActionResult Get()
        {
            try
            {
                return Ok();
            }
            catch
            {
                return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha

```

Passando a interface  
como parâmetro

```

no acesso ao banco de dados.");
    }

}

[HttpGet("{AlunoRA}")]
public IActionResult Get(string AlunoRA)
{
    try
    {
        return Ok();
    }
    catch
    {
        return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
    }
}

```

```

[HttpPost]
public async Task<IActionResult> post(Aluno model)
{
    try
    {
        this.Repo.Add(model);
        //
        if (await this.Repo.SaveChangesAsync()) {
            //return Ok();
            return Created($"{"/api/aluno/{model.Id}", model);
        }
    }
    catch
    {
        return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
    }
    return BadRequest();
}

```

Transformando método em assíncrono (*async*) e passando objeto Aluno como parâmetro

Chamando o método Add() do repositório

Espera (*await*) rodar o método SaveChangesAsync() do repositório

Informa que as alterações foram aplicadas no BD e chama uma rota

// continuação do código omitido

## Configurando o uso do Repositório

Precisamos adicionar mais uma configuração, onde informaremos ao projeto a viabilidade da injeção de dependência para o repositório. Ou seja, isso significa que quando alguém pedir o **IRepository** é para devolver o **Repository**. No arquivo **Startup.cs** a seguinte linha faz essa ligação:

### Startup.cs

```
// código acima omitido
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<EscolaContext>(
        x => x.UseSqlServer(Configuration.GetConnectionString("StringConexaoSQLServer"))
    );

    services.AddControllers();
    services.AddScoped<IRepository, Repository>();
}

// continuação do código omitido
```

## Método Get do Repositório

Vamos implementar os métodos GET no repositório. Iniciamos pela classe IRepository.cs:

### IRepository.cs

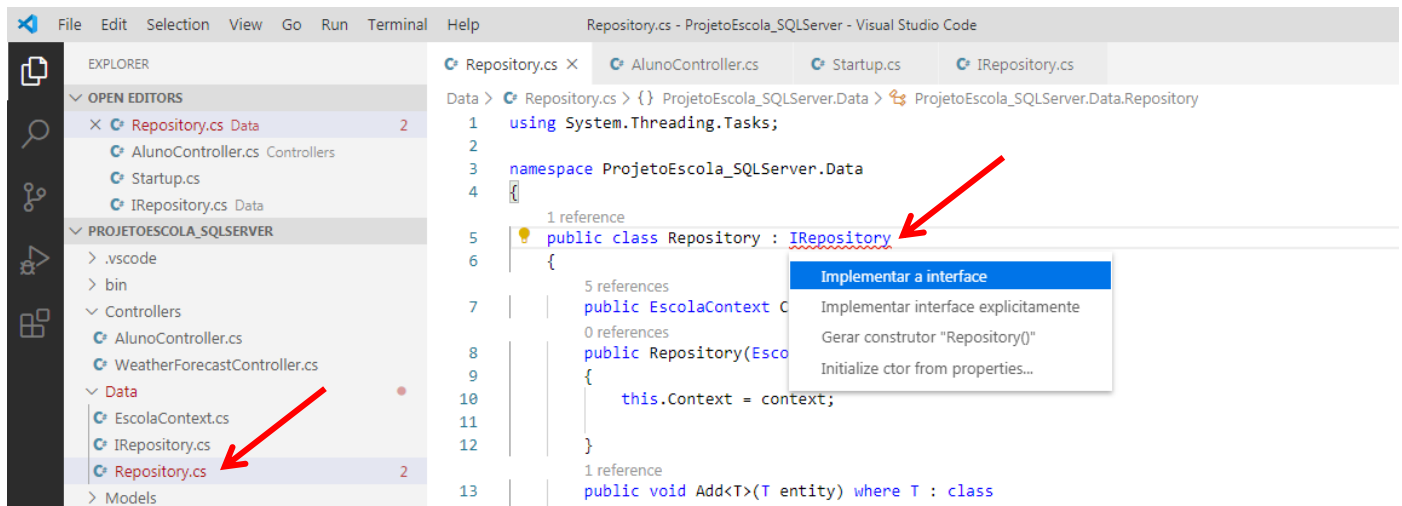
```
using System.Threading.Tasks;
using ProjetoEscola_SQLServer.Models;

namespace ProjetoEscola_SQLServer.Data
{
    public interface IRepository
    {
        // Métodos genéricos
        void Add<T>(T entity) where T: class;
        void Update<T>(T entity) where T: class;
        void Delete<T>(T entity) where T: class;
        Task<bool> SaveChangesAsync();

        // Métodos GET
        Task<Aluno[]> GetAllAlunosAsync();
        Task<Aluno[]> GetAllAlunosAsyncByRa(string RA);
    }
}
```

Ao fazer isso, a classe **Repository.cs** indicará que está inconsistente com a interface ao fazer com que o nome da dependência fique com um alerta de erro (sublinhado vermelho). Ao teclar **CTRL+.** sobre o nome da dependência IRepository os respectivos métodos serão criados:





Abaixo vemos os métodos criados:

### Repository.cs

// código acima omitido

```
public Task<Aluno[]> GetAllAlunosAsync()
{
    throw new System.NotImplementedException();
}

public Task<Aluno> GetAllAlunosAsyncByRa(string RA)
{
    throw new System.NotImplementedException();
}
```

Abaixo temos a implementação dos respectivos métodos:

### Repository.cs

```
using System.Linq;
using System.Threading.Tasks;
using ProjetoEscola_SQLServer.Models;
using Microsoft.EntityFrameworkCore;

namespace ProjetoEscola_SQLServer.Data
{
    public class Repository : IRepository
    {
        public EscolaContext Context { get; }
        public Repository(EscolaContext context)
        {
            this.Context = context;
        }
        public void Add<T>(T entity) where T : class
        {
            //throw new System.NotImplementedException();
            this.Context.Add(entity);
        }
    }
}
```

```

    }

    public void Delete<T>(T entity) where T : class
    {
        //throw new System.NotImplementedException();
        this.Context.Remove(entity);
    }

    public async Task<bool> SaveChangesAsync()
    {
        // Como é tipo Task vai gerar thread, então vamos definir o método como assíncrono (async)
        // Por ser assíncrono, o return deve esperar (await) se tem alguma coisa para salvar no BD
        // Ainda verifica se fez alguma alteração no BD, se for maior que 0 retorna true ou false
        return(await this.Context.SaveChangesAsync() > 0);
    }

    public void Update<T>(T entity) where T : class
    {
        //throw new System.NotImplementedException();
        this.Context.Update(entity);
    }

    public async Task<Aluno[]> GetAllAlunosAsync()
    {
        //throw new System.NotImplementedException();
        //Retornar para uma query qualquer do tipo Aluno
        IQueryable<Aluno> consultaAlunos = this.Context.Alunos;

        consultaAlunos = consultaAlunos.OrderBy(a => a.RA);

        // aqui efetivamente ocorre o SELECT no BD
        return await consultaAlunos.ToArrayAsync();
    }

    public async Task<Aluno> GetAllAlunosAsyncByRa(string RA)
    {
        //throw new System.NotImplementedException();
        //Retornar para uma query qualquer do tipo Aluno
        IQueryable<Aluno> consultaAlunos = this.Context.Alunos;

        consultaAlunos = consultaAlunos.OrderBy(a => a.RA)
                                         .Where(aluno => aluno.RA == RA);

        // aqui efetivamente ocorre o SELECT no BD
        return await consultaAlunos.FirstOrDefaultAsync();
    }
}
}

```

Voltando no controlador, os métodos Get ficam assim:

## AlunoController.cs

// código acima omitido

[HttpGet]

public async Task<IActionResult> Get()

{

try

{

var result = await this.Repo.GetAllAlunosAsync();

return Ok(result);

}

catch

{

return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha no acesso ao banco de dados.");

}

}

[HttpGet("{AlunoRA}")]

public async Task<IActionResult> Get(string AlunoRA)

{

try

{

var result = await this.Repo.GetAllAlunosAsyncByRa(AlunoRA);

return Ok(result);

}

catch

{

return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha no acesso ao banco de dados.");

}

}

// continuação do código omitido

Método do repositório

Método do repositório

## Método Put

Para implementar o método **Put** vamos alterar a classe AlunoController.cs ficando assim:

## AlunoController.cs

// código acima omitido

[HttpPut("{AlunoRA}")]

public async Task<IActionResult> put(string AlunoRA, Aluno model)

{

try

{

//verifica se existe aluno a ser alterado

var aluno = await this.Repo.GetAllAlunosAsyncByRa(AlunoRA);

if (aluno == null) return NotFound(); //método do EF

```

        this.Repo.Update(model);
        //
        if (await this.Repo.SaveChangesAsync()) {
            //return Ok();
            //pegar o aluno novamente, agora alterado para devolver pela rota abaixo
            aluno = await this.Repo.GetAllAlunosAsyncByRa(AlunoRA);
            return Created($"{"/api/aluno/{model.Ra}"}",aluno);
        }
    }
    catch
    {
        return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
    }
    // retorna BadRequest se não conseguiu alterar
    return BadRequest();
}

```

Veja que primeiramente usamos o método **GetAllAlunosAsyncByRa** para verificar se o aluno a ser alterado existe no banco de dados. Depois disso é realizado o Update().

Assim como nos outros métodos, a operação sendo bem sucedida é retornada uma rota com os dados do aluno em questão.

## Método Delete

Para implementar o método **Delete** vamos alterar a classe AlunoController.cs novamente ficando assim:

### AlunoController.cs

*// código acima omitido*

```

[HttpDelete("{AlunoRA}")]
public async Task<IActionResult> delete(string AlunoRA)
{
    try
    {
        //verifica se existe aluno a ser excluído
        var aluno = await this.Repo.GetAllAlunosAsyncByRa(AlunoRA);
        if (aluno == null) return NotFound(); //método do EF

        this.Repo.Delete(aluno);
        //
        if (await this.Repo.SaveChangesAsync()) {
            return Ok();
        }
    }
    catch
    {
        return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha

```

```

no acesso ao banco de dados.");
    }
    // retorna BadRequest se não conseguiu deletar
    return BadRequest();
}

```

De forma muito parecida com o método **Put**, usamos o método **GetAllAlunosAsyncByRa** para verificar se o aluno a ser alterado existe no banco de dados. Depois disso é realizado o Delete().

Se a operação for bem sucedida, o método retorna um status 200 pela função Ok().

A versão final da classe AlunoController.cs fica assim:

#### AlunoController.cs

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ProjetoEscola_SQLServer.Data;
using ProjetoEscola_SQLServer.Models;

namespace ProjetoEscola_SQLServer.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AlunoController : Controller
    {
        public IRepository Repo { get; }
        public AlunoController(IRepository repo)
        {
            this.Repo = repo;
            //construtor
        }

        [HttpGet]
        public async Task<IActionResult> Get()
        {
            try
            {
                var result = await this.Repo.GetAllAlunosAsync();
                return Ok(result);
            }
            catch
            {
                return this.StatusCode(StatusCode.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
            }
        }

        [HttpGet("{AlunoRA}")]
        public async Task<IActionResult> Get(string AlunoRA)

```

```

    {
        try
        {
            var result = await this.Repo.GetAllAlunosAsyncByRa(AlunoRA);
            return Ok(result);
        }
        catch
        {
            return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
        }
    }

[HttpPost]
public async Task<IActionResult> post(Aluno model)
{
    try
    {
        this.Repo.Add(model);
        //
        if (await this.Repo.SaveChangesAsync()) {
            //return Ok();
            return Created($"api/aluno/{model.Ra}", model);
        }
    }
    catch
    {
        return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
    }
    // retorna BadRequest se não conseguiu incluir
    return BadRequest();
}

[HttpPut("{AlunoRA}")]
public async Task<IActionResult> put(string AlunoRA, Aluno model)
{
    try
    {
        //verifica se existe aluno a ser alterado
        var aluno = await this.Repo.GetAllAlunosAsyncByRa(AlunoRA);
        if (aluno == null) return NotFound(); //método do EF

        this.Repo.Update(model);
        //
        if (await this.Repo.SaveChangesAsync()) {
            //return Ok();
            //pegar o aluno novamente, agora alterado para devolver pela rota a
baixo

            aluno = await this.Repo.GetAllAlunosAsyncByRa(AlunoRA);
            return Created($"api/aluno/{model.Ra}", aluno);
        }
    }
    catch
    {
        return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
    }
}

```

```

    }
}
catch
{
    return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
}
// retorna BadRequest se não conseguiu alterar
return BadRequest();
}

[HttpDelete("{AlunoRA}")]
public async Task<IActionResult> delete(string AlunoRA)
{
    try
    {
        //verifica se existe aluno a ser excluído
        var aluno = await this.Repo.GetAllAlunosAsyncByRa(AlunoRA);
        if (aluno == null) return NotFound(); //método do EF

        this.Repo.Delete(aluno);
        //
        if (await this.Repo.SaveChangesAsync()) {
            return Ok();
        }
    }
    catch
    {
        return this.StatusCode(StatusCodes.Status500InternalServerError, "Falha
no acesso ao banco de dados.");
    }
    // retorna BadRequest se não conseguiu deletar
    return BadRequest();
}
}
}
}

```

## Testes com o Postman

Antes de realizar os testes, verifique se a conexão VPN está ativa. Lembre-se que nesse exemplo acessaremos o servidor REGULUS.COTUCA.UNICAMP.BR. Verifique senha e nomes de tabelas e campos usados no exemplo.

No **Visual Studio Code** tecle **CTRL+F5** para executar a aplicação.

Vamos lá!

Iniciaremos os testes com o método **Get**:

GET http://localhost:5000/api/aluno/ Send Save

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies Code

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (4) Test Results Status: 200 OK Time: 37.77s Size: 948 B Save Response

Pretty Raw Preview Visualize JSON ≡

```

1 [
2   {
3     "id": 56,
4     "ra": "20111",
5     "nome": "Luana",
6     "codCurso": 19
7   },
8   {
9     "id": 58,
10    "ra": "20111",
11    "nome": "Luana Luana",
12    "codCurso": 19
13  },
14 ]

```

Teste com o método **Get** passando o RA:

GET http://localhost:5000/api/aluno/20444 Send Save

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies Code

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (4) Test Results Status: 200 OK Time: 5.96s Size: 200 B Save Response

Pretty Raw Preview Visualize JSON ≡

```

1 {
2   "id": 62,
3   "ra": "20444",
4   "nome": "Kaique",
5   "codCurso": 19
6 }

```

Teste com o método **Delete**:

DELETE http://localhost:5000/api/aluno/20555 Send Save

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies Code

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (3) Test Results Status: 200 OK Time: 384ms Size: 92 B Save Response

Pretty Raw Preview Visualize Text ≡



Teste com o método **Post**:

POST ▼ http://localhost:5000/api/aluno Send Save ▼

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings Cookies Code

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▼ Beautify

```
1 {  
2   "ra": "20555",  
3   "nome": "Alberto",  
4   "codCurso": 19  
5 }
```

Body Cookies Headers (5) Test Results Status: 201 Created Time: 2.71s Size: 231 B Save Response ▼

Pretty Raw Preview Visualize **JSON** ▼

```
1 {  
2   "id": 63,  
3   "ra": "20555",  
4   "nome": "Alberto",  
5   "codCurso": 19  
6 }
```

Teste com o método **Put**:

PUT ▼ http://localhost:5000/api/aluno/20444 Send ▼

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▼

```
1 {  
2   "id": 3,  
3   "ra": "20444",  
4   "nome": "Kaique Teste",  
5   "codCurso": 19  
6 }
```

## Fonte

- **Tutorial: criar uma API Web com ASP.NET Core**  
<https://docs.microsoft.com/pt-br/aspnet/core/tutorials/first-web-api?view=aspnetcore-3.1&tabs=visual-studio-code>
- **Tipos de retorno de ação do controlador em ASP.NET Core API Web**  
<https://docs.microsoft.com/pt-br/aspnet/core/web-api/action-return-types?view=aspnetcore-3.1>
- **Referência de ferramentas de Entity Framework Core-CLI .NET**  
<https://docs.microsoft.com/pt-br/ef/core/miscellaneous/cli/dotnet>
- **Curso ASP.NET Core e EF Core**  
Autor: Vinícius Andrade (Canal Youtube)