



# **Microsoft .NET 6.0**

## **com Entity Framework SQL Server e MySQL**

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

## Sumário

|       |   |    |
|-------|---|----|
| 1.    | Introdução .....  | 3  |
| 2.    | Protocolo HTTP .....  | 3  |
| 3.    | Aplicações REST .....   | 5  |
| 4.    | Recursos Necessários.....   | 5  |
| 5.    | Iniciando o projeto .NET .....  | 6  |
| 6.    | O Conceito MVC.....   | 10 |
| 7.    | Extensões .NET para Visual Studio Code .....                              | 10 |
| 8.    | Controladores .....   | 11 |
| 8.1.  | Tipos de Dados Retornados pelos Controladores .....                       | 13 |
| 9.    | Models.....   | 16 |
| 10.   | Criando Contexto com Entity Framework e escolhendo o Banco de Dados ..... | 17 |
| 11.1. | Entity Framework com SQL Server.....                                      | 18 |
| 11.2. | Entity Framework com MySQL.....   | 20 |
| 11.   | Implementando o Método GET no controlador .....                           | 22 |
| 12.   | Testes.....   | 23 |
| 13.   | Fonte.....  | 24 |

## 1. Introdução

O Microsoft NET ou simplesmente .NET é uma plataforma de desenvolvimento que vem apresentando inovações no âmbito do desenvolvimento web. Trata-se de uma série de bibliotecas que são disponibilizadas para criação de APIs Web que seguem corretamente o protocolo HTTP.

Vamos trabalhar com o conceito de uma API Web devido às atuais demandas, onde a necessidade de compartilhamento do mesmo conjunto de informações possa ser disponibilizado para diversos meios. Uma API Web tem como objetivo justamente disponibilizar informações de uma solução para uma série de dispositivos (como computadores, interfaces Web e aplicações mobile), dispositivos estes que consomem esta API.

As APIs Web usam o protocolo HTTP para trafegar suas informações já que é o protocolo padrão de transferência de dados em aplicações Web. As informações que são trafegadas em uma API Web podem estar em qualquer formato em tese, porém, os formatos mais utilizados são o JSON (JavaScript Object Notation) e XML (eXtensible Markup Language).

## 2. Protocolo HTTP

Como a aplicação API Web usa o protocolo HTTP para o tráfego das informações, é importante entendermos algumas características e recursos que serão usadas pela aplicação API Web .

HTTP é um acrônimo para *HyperText Transfer Protocol*, ou Protocolo de Transferência de HiperTexto. Trata-se de um protocolo que estabelece como deve ocorrer a comunicação entre uma máquina cliente que faz pedidos para uma máquina servidora. Ele é normatizado por uma especificação, a RFC 2616 (<http://tools.ietf.org/html/rfc2616> ).

O protocolo HTTP possui um modelo de comunicação *client/server*, ou seja, a comunicação entre aplicações que rodam em diferentes dispositivos, por exemplo, um navegador e um servidor Web. A aplicação cliente faz um pedido de informações para a aplicação servidora. Cada pedido que a máquina cliente faz para o servidor é chamado de **requisição** ou **request**; ao passo que a resposta do servidor para cada pedido é chamada de **resposta** ou **response**.

Quando digitamos no navegador um endereço, por exemplo, <http://www.google.com.br> perceba que o endereço é composto por algumas partes: primeiro identificamos o protocolo de comunicação (http) e depois a identificação do servidor Web que desejamos acessar. Ao teclarmos <ENTER> no navegador, uma requisição HTTP é então encaminhada para os servidores do Google, onde ela será processada.

Toda requisição HTTP é composta basicamente por duas partes distintas: cabeçalho (*header*) e corpo (*body*). O cabeçalho contém algumas informações específicas da requisição, como o tipo de resposta esperada do servidor e até mesmo o tempo de timeout. Já o corpo pode conter informações adicionais que o cliente pode enviar para o servidor que estarão atreladas à requisição (*request*). O corpo não é obrigatório, mas o cabeçalho é.

Uma requisição, como no exemplo acima para o endereço Google, terá uma requisição semelhante a:

▼ General

Remote Address: [2800:3f0:4001:813::1018]:80  
Request URL: http://www.google.com.br/  
Request Method: GET  
Status Code: 302 Found

▼ Request Headers

view source

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8  
Accept-Encoding: gzip, deflate, sdch  
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.6,en;q=0.4  
Connection: keep-alive  
Cookie: PREF=ID=1111111111111111:FF=0:LD=pt-BR:TM=1443491446:LM=1443540595:V=1:5=gh12qPtsVNzmdBXz; SID=DQAAANUBAABkEXicTsiaE8WpgZYwGJ4d4D7ZgAfe5ddoUNI0v1mFe-mD058mNpKo8UFLtFweU4D93a8lMpr-AJnCj68G9aOwRJ2Z7LTKZN9Detx9CoFPYrCEGKu84je3jZ1UGevEVGy3a11Yk85EoxtpRkLaoaTZ1EfAd8IbpCc0r1zJIREf01XfAEJFoVBduuIFzjqeL6qkOXaXrqqlwSfwCd\_L661p8Lj1p5NbO9bwFfqv8J9-eF1DBXFGX8J3dWNB087cLzr2sXtcrPc69vHrm\_Enq3rY70XcFu5nSRBI8Nrit58NmH0wbXedQ1TeuRCxpUjh8FSTCwHrIAjev\_RCYr-aIhQ0YOVdpUbR75gWQw88clw6bgZyJv71uh38L1qfoTSRHPAwI8CQe-DpLJW2o8jnnnyWBAstNlypjGL5a7grwNhkOGFc5UB7PyUXt6nPC9GxCGCe4vyKc2-UsS0kbnDEZ8iyVLnBRICeio13ZmoexQ1GgUnk0t6Pw6PvBlnL5FhaD4vHn01oDXAdiyCIJ-KPhQJ0-SSYGZni3ib-IG9ICQCFcnLhf8IjV-M4isFus8IG26-7DqSCUe20HNqhi-gj0h13J0-GfDnRLPqPvTXLeoEd9obMp-hSqsMjUdu8; HSID=ARIM88uRu9E7YD8kq; APISID=iNfNYwe8DkhE1I3a/AUVAJQAE1zVbsQH1K; NID=72=aGcdEBLyot5tS9n9rmtMIR6qIIC38cG21cwbU7JexUm89tSCNrf8Jv2kkXgmmw1wyuxJsscgmPNoXpJ5zxwWbnzoPj2JXoT1w6Dq9v3edCcF\_6NccClyYy88GSdja9Rlwy\_e4rw5HHCW794NdWz-SmQimH3fIVpuUuJeQ6rB-fr9WCImPzczYpL2Fz\_rGaBK5Gy7-2Gw84QfCu3eJ498xGH5T4GhxQ4s-sdJw5tbjJ\_IoaxEJT304M0IKdEH\_rplC4gjWkSK5ZUxtIsF6sU5J0v-tUH5yJ1G8G1E3FsPmDpw0rq5qFrAraFAUEJkXcPhvoE7A  
Host: www.google.com.br  
Upgrade-Insecure-Requests: 1  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.71 Safari/537.36  
X-Chrome-UMA-Enabled: 1  
X-Client-Data: CKO2yQEIqbbJQJbtSkBCO+IyqEI/ZXKAQI8mMoBCKCayGE=

Fonte: <https://www.treinaweb.com.br/>

No exemplo de requisição acima, vemos que muitas informações são enviadas do cliente para o servidor. Uma das informações é o **Request Method**, que informa o tipo de requisição que nesse caso é uma requisição do tipo GET. O protocolo HTTP tem uma série de métodos, como GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE e CONNECT. Nós, na maioria do tempo, utilizamos mais os métodos GET, POST, PUT e DELETE. Os significados destes métodos estão na tabela abaixo:

| Método | Significado semântico   |
|--------|---|
| GET    | Significa que queremos “pegar” algo no servidor: uma página, por exemplo. Requisições GET fazem com que o servidor devolva algo para o cliente, algo que estava “dentro” do servidor  |
| POST   | Significa que estamos querendo incluir alguma coisa no servidor. Por exemplo, se temos uma página de cadastro de usuários, a requisição que vai fazer com que o servidor faça o insert no banco de dados deve ser uma requisição POST, afinal, estamos criando um novo item que vai ficar no servidor |
| PUT    | Significa que estamos querendo atualizar alguma coisa no servidor   |
| DELETE | Significa que estamos querendo apagar alguma coisa do servidor  |

Fonte: <https://www.treinaweb.com.br/>

Uma resposta de um servidor Web, será algo como na imagem abaixo. Veja que muitas informações compõem uma resposta. Veja por exemplo, o campo **status:200**, que indica que requisição foi recebida com sucesso.

▼ Response Headers

alt-svc: quic="www.google.com:443"; p="1"; ma=600,quic=":443"; p="1"; ma=600  
alternate-protocol: 443:quic,p=1  
cache-control: private, max-age=0  
content-encoding: gzip  
content-type: text/html; charset=UTF-8  
date: Wed, 28 Oct 2015 06:30:00 GMT  
expires: -1  
p3p: CP="This is not a P3P policy! See http://www.google.com/support/accounts/bin/answer.py?hl=en&answer=151  
server: gws  
set-cookie: NID=72=Ej\_ojlac972Nmtke89YCWAdxtaMnmzQR-XExP8aBM7KP6N8vC014AFfTJ\_1Cdoobrdrac8oxbGhj8uPh7PN62qtFSpDxbX2qXj1taFx9BLhsWpXBF\_QovgCF4jt960embuejWuS2N71BDHFZRo\_jtWztgoGLYzLpjdbdb9s8qBuCFSh7wi6zEACvbGovq7dPeNr72ADKwY1k9rONw; expires=Thu, 28-Apr-2016 06:30:00 GMT; path=/; domain=.google.com.br; HttpOnly  
status: 200  
x-frame-options: SAMEORIGIN  
x-xss-protection: 1; mode=block

Fonte: <https://www.treinaweb.com.br/>

4

Os status possuem um padrão que seguem os significados de acordo com a tabela abaixo:

| Status | Descrição  |
|--------|--|
| 200    | OK. Significa que o servidor entendeu a requisição e a processou sem problemas.  |
| 302    | Found. Significa que o recurso solicitado de fato existe no servidor (status típico de requisições GET)  |
| 401    | Unauthorized. Significa que você tentou acessar algum recurso do servidor que exige autenticação para acesso, e você ainda não realizou este processo.                                     |
| 404    | Not Found. Significa que você solicitou algum recurso no servidor que não existe no lugar que você indicou. Por exemplo: se você tenta acessar alguma página de algum site que não existe. |
| 500    | Internal Server Error. Significa que o servidor encontrou um erro durante o processamento da requisição.   |

Fonte: <https://www.treinaweb.com.br/>

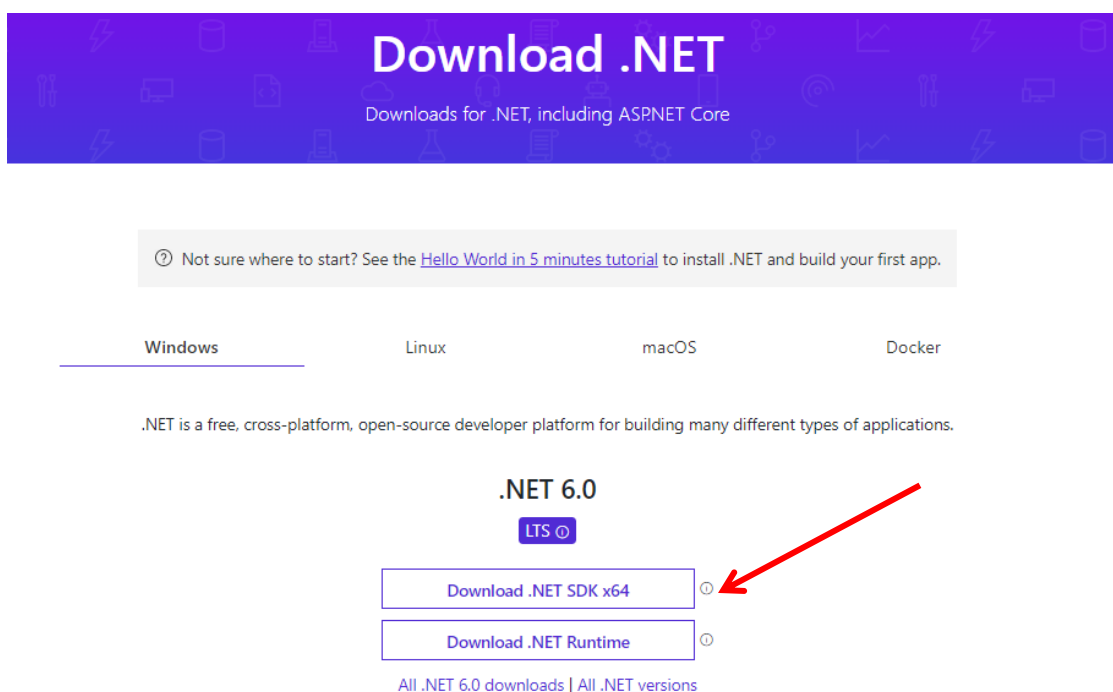
Através destes status HTTP que o cliente sabe se a requisição que ele disparou deu certo ou não.

### 3. Aplicações REST

Como o tipo de aplicação que vamos trabalhar é do tipo REST, é muito importante a leitura do artigo **Mas, afinal de contas, o que é REST?** (<https://www.treinaweb.com.br/blog/rest-nao-e-simplesmente-retornar-json-into-alem-com-apis-rest/>) para esclarecer muitos pontos sobre o assunto.

### 4. Recursos Necessários

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



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

Você pode baixar o **.Net SDK**. Depois de instalado, na linha de comando (pelo CMD do Windows ou terminal do Visual Studio Code) o comando abaixo nos exibe um help com as opções para o comando dotnet:

**dotnet -h**

O comando abaixo nos exibe a lista de *templates* para projetos que podemos desenvolver com o .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**

Esses modelos correspondem à sua entrada:

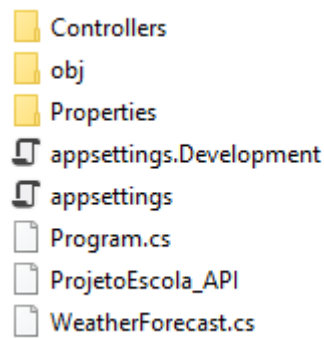
| Nome do modelo                                  | Nome Curto          | Idioma     | Tags                       |
|---|---------------------|------------|----------------------------|
| -----   | -----               | -----      | -----                      |
| Aplicativo do Console                           | console             | [C#],F#,VB | Common/Console             |
| Aplicativo do Windows Forms                     | winforms            | [C#],VB    | Common/WinForms            |
| Aplicativo WPF                                  | wpf                 | [C#],VB    | Common/WPF                 |
| Arquivo de Configuração da Web                  | webconfig           |            | Config                     |
| arquivo de dotnet gitignore                     | gitignore           |            | Config                     |
| Arquivo de manifesto da ferramenta local Dotnet | tool-manifest       |            | Config                     |
| Arquivo de Solução                              | sln                 |            | Solution                   |
| Arquivo EditorConfig                            | editorconfig        |            | Config                     |
| arquivo global.json                             | globaljson          |            | Config                     |
| ASP.NET Core Empty                              | web                 | [C#],F#    | Web/Empty                  |
| ASP.NET Core gRPC Service                       | grpc                | [C#]       | Web/gRPC                   |
| ASP.NET Core Web API                            | webapi              | [C#],F#    | Web/WebAPI                 |
| ASP.NET Core Web App                            | webapp,razor        | [C#]       | Web/MVC/Razor Pages        |
| ASP.NET Core Web App (Model-View-Controller)    | mvc                 | [C#],F#    | Web/MVC                    |
| 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                |
| Biblioteca de Classes                           | classlib            | [C#],F#,VB | Common/Library             |
| Biblioteca de Classes do Windows Forms          | winformslib         | [C#],VB    | Common/WinForms            |
| Biblioteca de Controles do Windows Forms        | winformscontrollib  | [C#],VB    | Common/WinForms            |
| Blazor Server App                               | blazorserver        | [C#]       | Web/Blazor                 |
| Blazor WebAssembly App                          | blazorwasm          | [C#]       | Web/Blazor/WebAssembly/PWA |
| Configuração do NuGet                           | nugetconfig         |            | Config                     |
| MSTest Test Project                             | mstest              | [C#],F#,VB | Test/MSTest                |
| MVC ViewImports                                 | viewimports         | [C#]       | Web/ASP.NET                |
| MVC ViewStart                                   | viewstart           | [C#]       | Web/ASP.NET                |
| NUnit 3 Test Item                               | nunit-test          | [C#],F#,VB | Test/NUnit                 |
| NUnit 3 Test Project                            | nunit               | [C#],F#,VB | Test/NUnit                 |
| Protocol Buffer File                            | proto               |            | Web/gRPC                   |
| Razor Class Library                             | razorclasslib       | [C#]       | Web/Razor/Library/Razor    |
| Class Library                                   |                     |            |                            |
| Razor Component                                 | razorcomponent      | [C#]       | Web/ASP.NET                |
| Razor Page                                      | page                | [C#]       | Web/ASP.NET                |
| Worker Service                                  | worker              | [C#],F#    | Common/Worker/Web          |
| WPF Class library                               | wpflib              | [C#],VB    | Common/WPF                 |
| WPF Custom Control Library                      | wpfcustomcontrollib | [C#],VB    | Common/WPF                 |
| WPF User Control Library                        | wpfusercontrollib   | [C#],VB    | Common/WPF                 |
| xUnit Test Project                              | xunit               | [C#],F#,VB | Test/xUnit                 |

## 5. Iniciando o projeto .NET

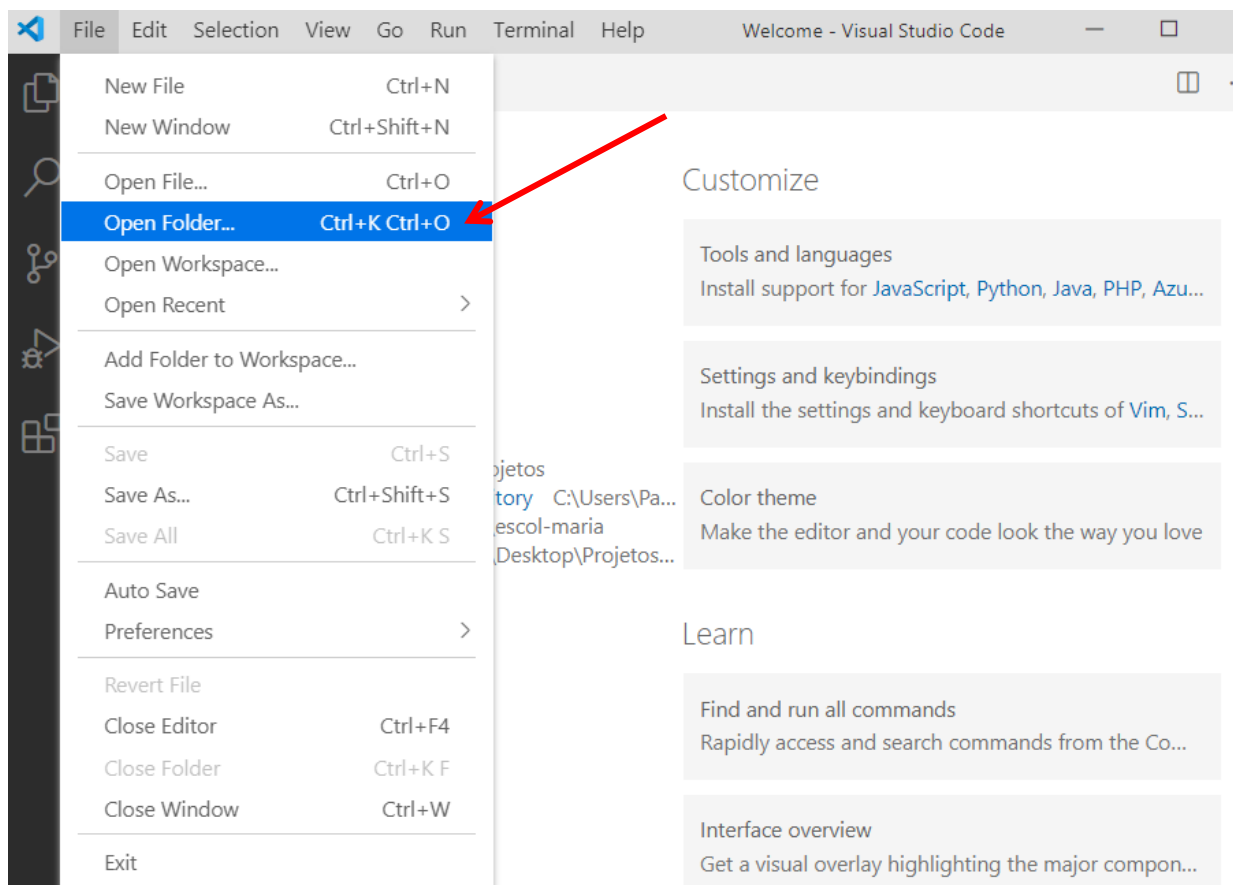
Vamos iniciar o projeto .NET Core 2 usando o CLI (*Command Line Interface* do .NET). Usaremos o comando abaixo para criar o projeto:

**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.



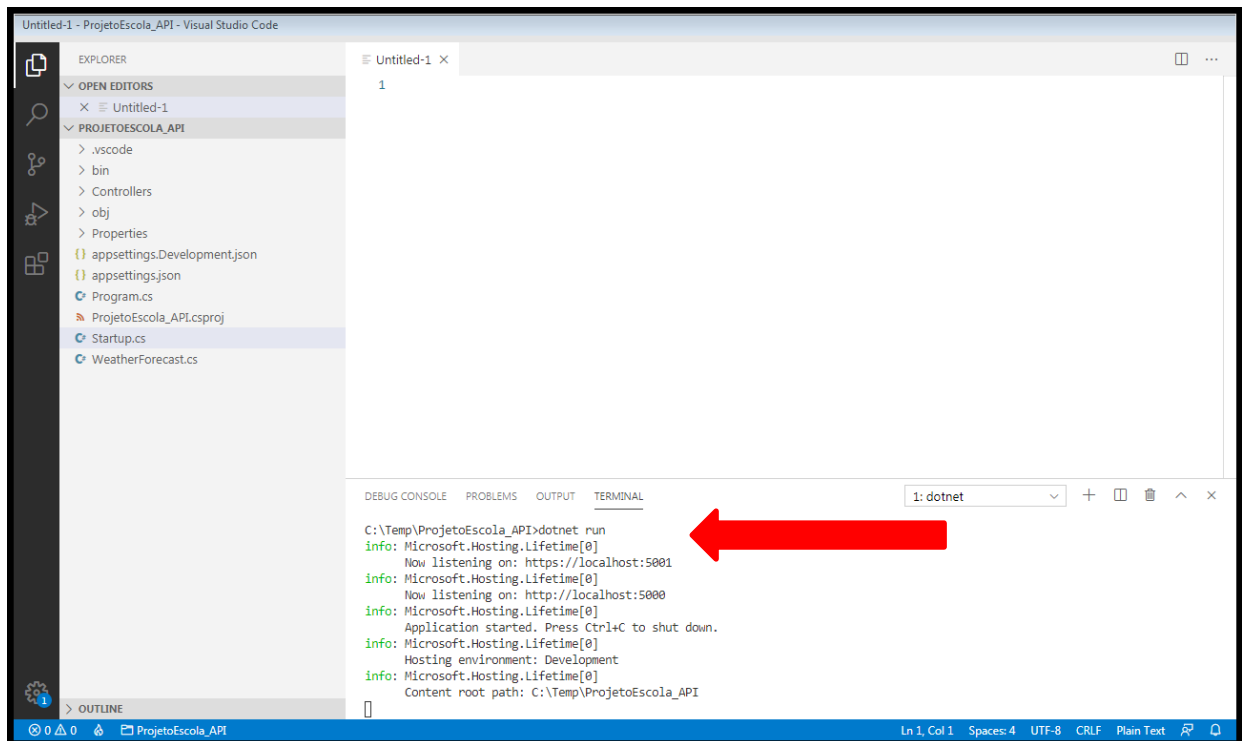
Podemos usar o **Visual Studio Code** ou **Visual Studio Enterprise** ou **Community** para editar o projeto. Por questões de facilidade, vamos usar o **Visual Studio Code**. Vamos abrir a pasta do projeto no **Visual Studio Code**:



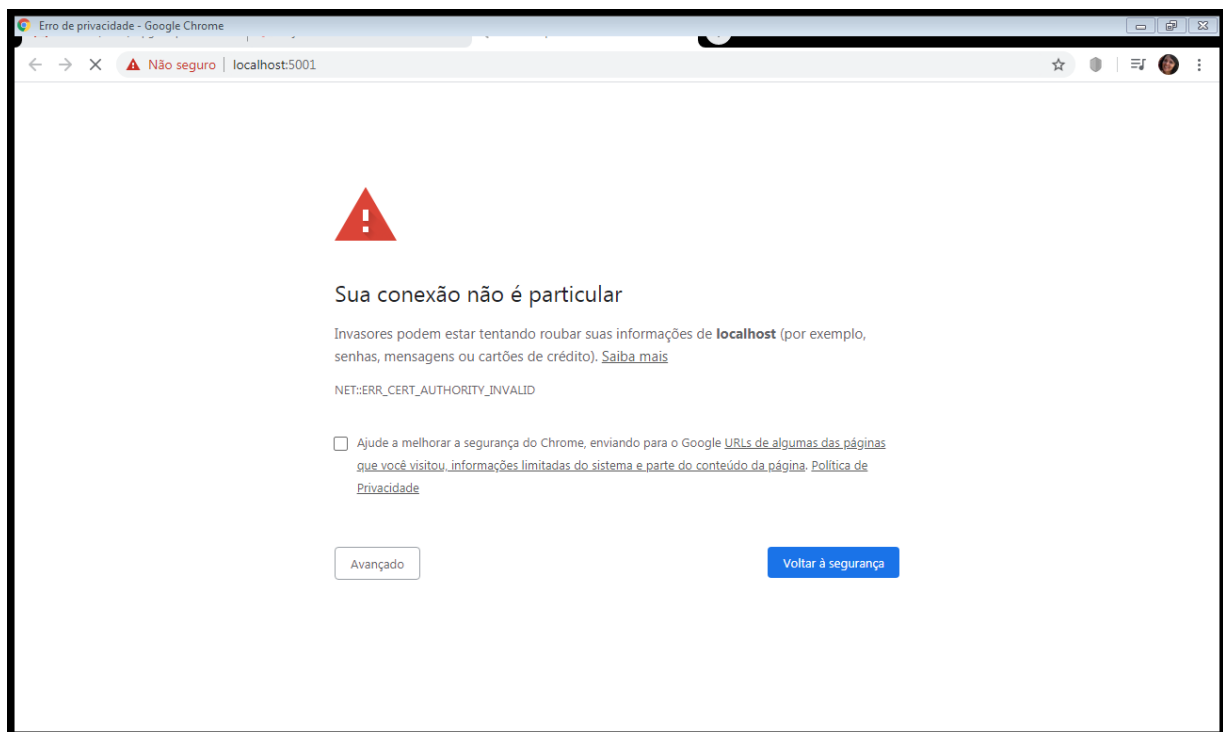
Vá até a pasta onde o projeto **ProjetoEscola\_API** e abra-a pelo **Visual Studio Code**:

Abra o terminal do **Visual Studio Code** e digite 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 protocolo HTTPS. Para alterar isso, basta alterar o arquivo **Program.cs** comentando a linha referente a essa configuração. Como mostra no exemplo abaixo:

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

//app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```



Nesse primeiro momento como não criamos nenhuma rota para a nossa aplicação a página não exibirá nenhuma informação. É normal! Mas com exemplo, o projeto possui a classe **WeatherForecast** onde podemos ver a API em execução:



localhost:7062/WeatherForecast

```
[{"date": "2022-05-19T11:19:06.6343015-03:00", "temperatureC": 3, "temperatureF": 37, "summary": "Bracing"}, {"date": "2022-05-20T11:19:06.6354233-03:00", "temperatureC": 21, "temperatureF": 69, "summary": "Balmy"}, {"date": "2022-05-21T11:19:06.6354283-03:00", "temperatureC": 50, "temperatureF": 121, "summary": "Hot"}, {"date": "2022-05-22T11:19:06.6354286-03:00", "temperatureC": 39, "temperatureF": 102, "summary": "Balmy"}, {"date": "2022-05-23T11:19:06.6354287-03:00", "temperatureC": 52, "temperatureF": 125, "summary": "Mild"}]
```

## 6. O Conceito MVC

Vamos trabalhar com o conceito MVC (Model-View-Control) no projeto e para isso é muito importante que você leia alguns artigos sobre o assunto caso você não faça a menor ideia do que seja isso! Vou deixar aqui dois artigos para essa leitura:

- **Introdução ao Padrão MVC** (<https://www.devmedia.com.br/introducao-ao-padrao-mvc/29308> )
- **Afinal, o que é MVC?** (<https://www.treinaweb.com.br/blog/o-que-e-mvc/>)

## 7. Extensões .NET para Visual Studio Code

Como sugestão, vou indicar algumas extensões do **Visual Studio Code** para edição de projeto .NET ou C#. Essas extensões ajudam na edição e identificação de erros.



**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.vscode-nuget-package-manager

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

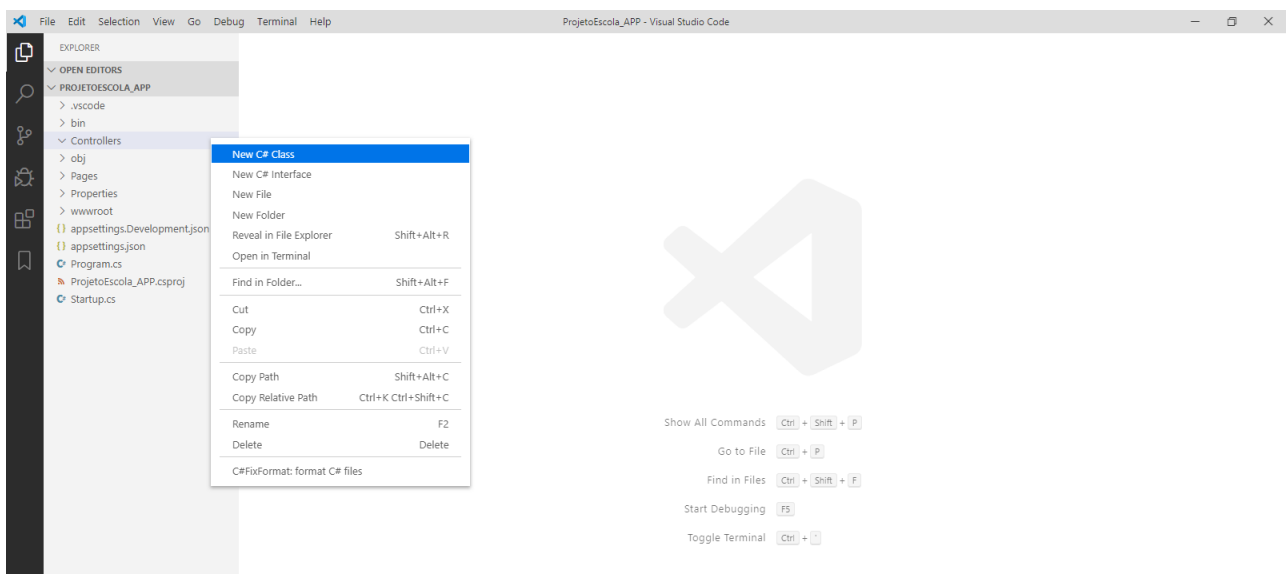
Add or remove .NET Core 1.1+ package references to/from your project's .csproj or .fsproj file using Code's Command Palette.

**Uninstall** This extension is enabled globally.

Details Contributions Changelog

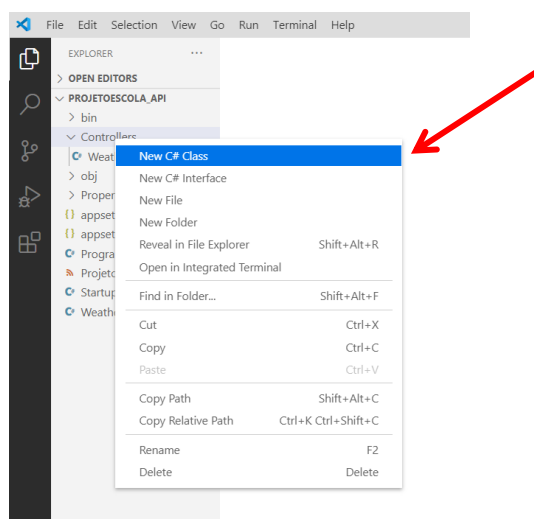
## 8. Controladores

Vamos criar na pasta **Controllers** uma nova classe C# chamada **HomeController.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 que irão tratar as requisições web são chamados de **Action**. No arquivo criado (**HomeController.cs**) temos uma **Action** chamada **Index** que devolve uma resposta do tipo **ActionResult**<sup>1</sup>. Dentro do método vamos colocar o código da regra de negócio da aplicação.

Com as extensões do Visual Studio Code acima indicadas, no menu de contexto para criação de arquivo, aparecerá a opção de criação de classe C#.



Adicionaremos as seguintes linhas na nova classe:

<sup>1</sup> Consulte [Tipos de retorno de ação do controlador em .NET Core API Web](#) no final do arquivo para saber outros tipos que os métodos podem retornar.

### AlunoController.cs

```
using Microsoft.AspNetCore.Mvc;

namespace ProjetoEscola_API.Controllers
{
    [ApiController]
    [Route("/")]
    public class HomeController: ControllerBase
    {
    }
}
```

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

A classe **ControllerBase** fornece muitas funcionalidades padrão para lidar com solicitações HTTP

O atributo **[ApiController]** indica que esta classe se refere à um controlador da API. Ou seja, esta classe irá tratar requisições HTTP que poderão chegar até a API pela rota especificada.

Neste controlador, vamos adicionar um **método** para tratarmos as rotas, ou seja, nesse caso quando o usuário realizar requisição para o endereço <http://localhost:5176> ou <https://localhost:7062>, o método **Inicio()** deverá ser executado. A API pode ter vários controladores, cada um será identificado por uma rota através do atributo **[Route()]**. No caso do controlador **HomeController**, a rota definida por **[Route("/")]** indica que o acesso será pelos endereços acima, ou seja, pelo endereço “raiz” da API.

### HomeController.cs

```
using Microsoft.AspNetCore.Mvc;

namespace ProjetoEscola_API.Controllers
{
    [ApiController]
    [Route("/")]
    public class HomeController: ControllerBase
    {
        [HttpGet]
        public String Inicio()
        {
            return "Funcionou!";
        }
    }
}
```

Temos aqui o primeiro método desse controlador. Este método responderá às **requisições GET** que chegarem a este controlador, de acordo com a respectiva rota (Route). Este método retornará uma **string** que será processada pelo navegador/usuário requisitante.

O controlador Home exibirá o conteúdo referente à rota raiz do projeto (<http://localhost:5176> ou <https://localhost:7062>) por causa do atributo **[Route("/")]**.

← → ↻ ⓘ localhost:5176

Funcionou!

## 8.1. Tipos de Dados Retornados pelos Controladores

O ASP.NET Core oferece as seguintes opções para tipos de retorno de ação do controlador de API Web:

- **Tipo específico**
- **ActionResult**
- **ActionResultT<>**

### Tipo específico

A ação mais simples retorna um tipo de dados complexo ou primitivo (por exemplo, string ou um tipo de objeto personalizado). Considere a seguinte ação, que retorna uma **String**:

```
[HttpGet]
public String Inicio()
{
    return "Funcionou!";
}
```

Sem condições conhecidas para proteger contra ações durante a execução, retornar um tipo específico pode ser suficiente. A ação precedente não aceita parâmetros, assim validações de parâmetros não são necessárias.

Quando múltiplos tipos de retorno são possíveis, é comum combinar um retorno do tipo **ActionResult** com o tipo de retorno primitivo ou complexo. Tanto o **ActionResult** como o **ActionResultT<T>** são necessários para acomodar este tipo de ação.

```
[HttpGet]
public ActionResult Inicio()
{
    return new ContentResult
    {
        ContentType = "text/html",
        Content = "<h1>API Projeto Escola: funcionou!!!!</h1>"
    };
}
```

← → ↻ localhost:5000

**API Projeto Escola: funcionou!!!!**

### Retornando IEnumerable<T> or IQueryable<T>

O ASP.NET armazena em *buffers* os resultados de ações que retornam **IEnumerable<T>** antes de enviá-los para a resposta da requisição. Para iterações assíncronas considere **IAsyncEnumerable<T>**. Por fim, o modo de iteração baseia-se no tipo concreto subjacente que está sendo retornado. O MVC automaticamente armazena em *buffer* qualquer tipo concreto que implementa **IAsyncEnumerable<T>**.

Considere a ação abaixo que retorna registros de produtos como **IEnumerable<Product>**:

```
[HttpGet("syncsale")]
public IEnumerable<Product> GetOnSaleProducts()
{
    var products = _repository.GetProducts();

    foreach (var product in products)
    {
        if (product.IsOnSale)
        {
            yield return product;
        }
    }
}
```

O **IAsyncEnumerable<Product>** equivalente da ação anterior é:

```
[HttpGet("asynsale")]
public async IAsyncEnumerable<Product> GetOnSaleProductsAsync()
{
    var products = _repository.GetProductsAsync();

    await foreach (var product in products)
    {
        if (product.IsOnSale)
        {
            yield return product;
        }
    }
}
```

## ActionResult

O **ActionResult** tipo de retorno é apropriado quando vários tipos **ActionResult** de retorno são possíveis em uma ação. Os tipos **ActionResult** representam vários códigos de status HTTP. Qualquer classe não abstrata que deriva de **ActionResult** qualifica como um tipo de retorno válido. Alguns tipos de retorno comuns nessa categoria são **BadRequestResult (400)**, **NotFoundResult (404)** e **OkObjectResult (200)**. Como alternativa, métodos de conveniência na classe **ControllerBase** podem ser usados para retornar tipos **ActionResult** de uma ação. Por exemplo, **return BadRequest();** é uma forma resumida de **return new BadRequestResult();**

Como há vários tipos de retorno e caminhos nesse tipo de ação, é necessário usar o **[ProducesResponseType]** atributo. Esse atributo produz detalhes de resposta mais descritivos para páginas de ajuda da API Web geradas por ferramentas como o **Swagger**. **[ProducesResponseType]** indica os tipos conhecidos e os códigos de status HTTP a serem retornados pela ação.

## Ação Síncrona

Considere a seguinte ação síncrona em que há dois tipos de retorno possíveis:

```
[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK, Type = typeof(Product))]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public IActionResult GetById(int id)
{
    if (!_repository.TryGetProduct(id, out var product))
    {
        return NotFound();
    }

    return Ok(product);
}
```

Na ação anterior:

- Um código de **status 404** é retornado quando **id** o produto representado por não existe no armazenamento de dados subjacente. O **método NotFound** é invocado como uma forma resumida para **return new NotFoundResult();**.
- Um código de **status 200** é retornado com o objeto **Product** quando o produto existe. O método **Ok** é invocado como uma forma resumida para **return new OkObjectResult(product);**.

## Ação assíncrona

Considere a seguinte ação assíncrona em que há dois tipos de retorno possíveis:

```
[HttpPost]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> CreateAsync(Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return BadRequest();
    }

    await _repository.AddProductAsync(product);

    return CreatedAtAction(nameof(GetById), new { id = product.Id }, product);
}
```

Na ação anterior:

- Um código de **status 400** é retornado quando a descrição do produto contém "Widget XYZ". O **método BadRequest** é invocado como uma forma resumida para **return new BadRequestResult();**.
- Um código de **status 201** é gerado pelo **método CreatedAtAction** quando um produto é criado. Uma alternativa à chamada **CreatedAtAction** é **return new CreatedAtActionResult(nameof(GetById), "Products", new { id = product.Id }, product);**. Nesse caminho de código, o objeto **Product** é fornecido no corpo da resposta. Um **Location header** de resposta que contém a URL do produto recém-criado é fornecido.

Por exemplo, o modelo a seguir indica que as solicitações devem incluir as propriedades **Name** e **Description**. A falha ao fornecer **Name** e **Description** na solicitação faz com que a validação do modelo falhe.

```
public class Product
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }

    [Required]
    public string Description { get; set; }

    public bool IsOnSale { get; set; }
}
```

Se o atributo **[ApiController]** for aplicado, os erros de validação do modelo resultarão em um código de **status 400**.

## ActionResultT<>

ASP.NET Core inclui o tipo de retorno **ActionResultT<>** para ações do controlador de API Web. Ele permite que você retorne um tipo que deriva de **ActionResult** ou retorne um tipo específico. **ActionResult<T>** oferece os seguintes benefícios em relação ao tipo **IActionResult**:

- A propriedade **[ProducesResponseType]** do atributo **Type** pode ser excluída. Por exemplo, **[ProducesResponseType(200, Type = typeof(Product))]** é simplificado para **[ProducesResponseType(200)]**. O tipo de retorno esperado da ação é inferido de **T** em **ActionResult<T>**.
- Operadores de conversão implícita são compatíveis com a conversão de **T** e **ActionResult** em **ActionResult<T>**. **T** converte em **ObjectResult**, o que significa que **return new ObjectResult(T);** é simplificado para **return T;**

```
[HttpGet]
public ActionResult<List<Aluno>> GetAll()
{
    return _context.Aluno.ToList();
}
```

A maioria das ações tem um tipo de retorno específico. Condições inesperadas podem ocorrer durante a execução da ação, caso em que o tipo específico não é retornado. Por exemplo, o parâmetro de entrada de uma ação pode falhar na validação do modelo. Nesse caso, é comum retornar o tipo **ActionResult** adequado, em vez do tipo específico.

## 9. Models

A tabela usada na aplicação terá a estrutura abaixo. Você poderá optar se usará o banco SQL Server do Colégio, SQL Server Express (instalação local) ou MySQL (instalação local).



### Tabela Alunos no SQL Server

```
CREATE TABLE [dbo].[Aluno] (  
    [id] INT IDENTITY (1, 1) NOT NULL,  
    [ra] CHAR (5) NOT NULL,  
    [nome] VARCHAR (30) NULL,  
    [codCurso] INT NULL,  
    PRIMARY KEY CLUSTERED ([id] ASC)  
);  
  
Insert into Aluno (ra,nome,codCurso) values ('20001','Paulo',19)  
Insert into Aluno (ra,nome,codCurso) values ('20002','Ana',19)  
Insert into Aluno (ra,nome,codCurso) values ('20003','Pedro',15)
```

### Tabela Alunos no MySQL

```
CREATE TABLE aluno (  
    id INT AUTO_INCREMENT NOT NULL,  
    ra CHAR (5) NOT NULL,  
    nome VARCHAR (30) NULL,  
    codCurso INT NULL,  
    PRIMARY KEY CLUSTERED (id ASC)  
);  
  
Insert into Aluno (ra,nome,codCurso) values ('20001','Paulo',19)  
Insert into Aluno (ra,nome,codCurso) values ('20002','Ana',19)  
Insert into Aluno (ra,nome,codCurso) values ('20003','Pedro',15)
```

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_API.Models  
{  
    public class Aluno  
    {  
        public int id { get; set; }  
        public string? ra { get; set; }  
        public string? nome { get; set; }  
        public int codCurso { get; set; }  
    }  
}
```

Vamos usar o **Entity Framework** toda tabela precisa ter o campo **Id**

## 10. Criando Contexto com Entity Framework e escolhendo o Banco de Dados

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 o chamaremos de **EscolaContext.cs**.

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

## 11.1. Entity Framework com SQL Server

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

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



```
EscolaContext.cs
using Microsoft.EntityFrameworkCore;

namespace ProjetoEscola_API.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_API.Models;
using System.Diagnostics.CodeAnalysis;

namespace ProjetoEscola_API.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 **Program.cs** acrescentando as linhas abaixo:

## Program.cs

```
using Microsoft.EntityFrameworkCore;
using ProjetoEscola_API.Data;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

// Add DbContext
builder.Services.AddDbContext<EscolaContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("StringConexaoSQLServer"));
});
```

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

< código omitido >

Para acrescentar as importações corretas dentro da classe, tecla **CTRL+.** sobre os nomes dos objetos destacados, como no exemplo abaixo. O próprio **Visual Studio Code** irá sugerir as correções:

No arquivo **appsettings.Development.json** precisamos declarar essa string de conexão com o banco de dados. Aqui temos duas opções para a string de conexão dependendo se o banco de dados SQL Server que você irá optar por usar: **SQL Server do Colégio (REGULUS)** ou **SQL Server Express**. Se você optar por usar o servidor do Colégio (REGULUS) terá que fazer a conexão com a VPN para implementar o projeto. Se optar pelo SQL Server Express você terá que fazer a instalação no seu computador e nesse caso não dependerá da VPN.

- Utilizando o servidor **SQL Server do Colégio (REGULUS)**, nesse caso é necessário ativar a VPN:

## appsettings.Development.json

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

- Utilizando o servidor **SQL Server Express local**, ou seja, instalado no seu computador:

#### appsettings.Development.json

```
{
  "ConnectionStrings": {
    "StringConexaoSQLServer": "data source=.\SQLEXPRESS;Integrated Security=SSPI;Database=master"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```



## 11.2. Entity Framework com MySQL


Assim como funciona no SQL Server, para usarmos o **Entity Framework com MySQL** também é necessário instalar uma biblioteca no projeto através do comando abaixo:

**dotnet add package MySQL.EntityFrameworkCore**

#### EscolaContext.cs

```
using Microsoft.EntityFrameworkCore;

namespace ProjetoEscola_API.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_API.Models;
using System.Diagnostics.CodeAnalysis;

namespace ProjetoEscola_API.Data
{
    public class EscolaContext : DbContext
    {
        protected readonly IConfiguration Configuration;
    }
}
```

```

public EscolaContext(IConfiguration configuration)
{
    Configuration = configuration;
}

protected override void OnConfiguring(DbContextOptionsBuilder options)
{
    // connect to sql server with connection string from app settings
    options.UseSqlServer(Configuration.GetConnectionString("StringConexaoSQLServer"));
}

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

```

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

#### Program.cs

```

using Microsoft.EntityFrameworkCore;
using ProjetoEscola_API.Data;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

// Add DbContext
builder.Services.AddDbContext<EscolaContext>(options =>
{
    options.x.UseMySQL("server=localhost;database=patricia;user=root")
});
});

```

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

< código omitido >

Para acrescentar as importações corretas dentro da classe, tecle **CTRL+.** sobre o os nomes os objetos destacados, como no exemplo abaixo. O próprio **Visual Studio Code** irá sugerir as correções:

## 11. Implementando o Método GET no controlador

Usaremos o **Entity Framework** para implementar a persistência dos dados, ou seja, para realizar as operações no banco de dados. Por isso você perceberá que existe uma camada de abstração que isola o acesso direto ao banco de dados. Não usaremos as consultas SQL e sim os métodos do **Entity Framework**.

Uma das vantagens de se usar o **Entity Framework** é que daqui em diante no nosso projeto, não importa qual banco de dados estamos utilizando. O banco já foi definido no Entity e de agora em diante a persistência dos dados é abstraída, ou seja, os próximos comandos servirão para qualquer banco definido previamente.

Devemos informar ao controlador qual contexto de dados usaremos e modificaremos o método GetAll() de forma que todos os dados da tabela Alunos sejam resgatados. Veja que alteramos o tipo de retorno do método para uma lista de objetos.

### AlunoController.cs

```
using Microsoft.AspNetCore.Mvc;
using ProjetoEscola_API.Data;
using ProjetoEscola_API.Models;

namespace ProjetoEscola_API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AlunoController : ControllerBase
    {
        private EscolaContext _context;
        public AlunoController(EscolaContext context)
        {
            // construtor
            _context = context;
        }

        [HttpGet]
        public ActionResult<List<Aluno>> GetAll() {
            return _context.Aluno.ToList();
        }
    }
}
```

Definimos aqui qual contexto de dados usaremos nesse controlador e relacionamos juntamente ao controlador.

Alteramos o método GetAll() para que, através do método ToList() do Entity Framework os dados da tabela sejam resgatados

Com o uso do **Entity Framework**, não manipulamos diretamente o SQL para resgatar ou gravar os dados na tabela no banco de dados. Veremos mais detalhes do framework futuramente.

## 12. Testes

Antes de realizar os testes é necessário checar as seguintes informações:

- ✓ Se você estiver usando o servidor **SQL Server do Colégio (REGULUS)**: verifique se a VPN está ativa, verifique se a tabela foi criada e os dados inseridos
- ✓ Se você estiver usando o servidor **SQL Server Express (instalação local)**: verifique se a tabela foi criada e os dados inseridos. O serviço do SQL Server Express deve estar rodando a partir do momento que foi instalado, mas você pode conferir em **Painel de Controle/Ferramentas Administrativas/Serviços** e verificar se o serviço **SQL Server (SQLEXPRESS)** está em execução.
- ✓ Se você estiver usando o servidor **MySQL**, verifique através do painel de controle do XAMPP se está “startado” e se a tabela foi criada.

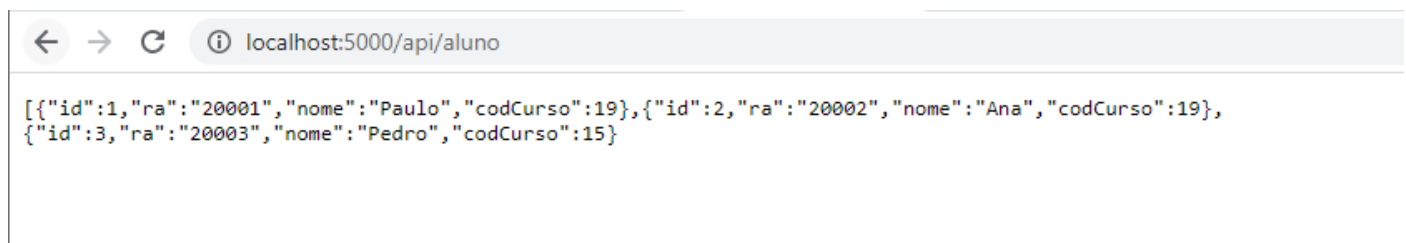
No **Visual Studio Code** tecle **CTRL+F5** para executar a aplicação ou digite no terminal:

dotnet run

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
```

Como o **método Get** apenas retornará dados do banco é possível realizar o teste pelo navegador. Se digitarmos na barra de endereço <https://localhost:5001/api/aluno> ou <http://localhost:5000/api/aluno> veja que os mesmos dados deverão ser exibidos no mesmo formato JSON no corpo da página.

O resultado deverá ser algo como na imagem abaixo:



## 13. Fonte

- **Tutorial: criar uma API Web com .NET Core**

[https://docs.microsoft.com/pt-br/learn/modules/build-web-api-aspnet-core/?WT.mc\\_id=dotnet-35129-website](https://docs.microsoft.com/pt-br/learn/modules/build-web-api-aspnet-core/?WT.mc_id=dotnet-35129-website)

<https://docs.microsoft.com/pt-br/aspnet/core/tutorials/first-web-api?view=aspnetcore-6.0&tabs=visual-studio-code>

- **Tipos de retorno de ação do controlador na API Web do ASP.NET Core**

<https://docs.microsoft.com/pt-br/aspnet/core/web-api/action-return-types?view=aspnetcore-6.0>

- **Formatar dados de resposta na API Web ASP.NET Core**

<https://docs.microsoft.com/pt-br/aspnet/core/web-api/advanced/formatting?view=aspnetcore-6.0>

- **Introdução ao EF Core**

<https://docs.microsoft.com/pt-br/ef/core/get-started/?tabs=netcore-cli>

- **Referência de ferramentas de Entity Framework Core-CLI .NET**

<https://docs.microsoft.com/pt-br/ef/core/miscellaneous/cli/dotnet>

- **Curso .NET Core e EF Core**

Autor: Vinícius Andrade (Canal Youtube)

- **Introdução ao Padrão MVC**

<https://www.devmedia.com.br/introducao-ao-padrao-mvc/29308>

- **Afinal, o que é MVC?**

<https://www.treinaweb.com.br/blog/o-que-e-mvc/>

- **C# (C Sharp) - APIs REST com .NET Web API**

Autor: Treinaweb

- **REST não é simplesmente retornar JSON: indo além com APIs REST**

<https://www.treinaweb.com.br/blog/rest-nao-e-simplesmente-retornar-json-into-alem-com-apis-rest/>

- **Entity Framework Core com MySQL**

<https://www.treinaweb.com.br/blog/entity-framework-core-com-mysql/>