**Construindo APIs Restful com .NET Core usando Entity Framework e JWT  
Parte - I**

Olá desenvolvedor, tudo bem? Espero que esteja bem e com ânimo, pois nesta série de 4 artigos irei mostrar construir APIs Restful profissionais, utilizando Entity Framework Core para acesso a dados e JWT (JSON Web Tokens) para a segurança.

Meu objetivo é que no final desses artigos você tenha aprendido a importância do padrão restful e como utilizar ele para criar suas próprias APIs de forma fácil, segura e rápida. E o melhor de forma gratuita, pois assim que o ensino deveria ser.

**Requisitos**

Sem mais delongas, já irei apresentar as ferramentas e as versões que iremos utilizar:

* Visual Studio 2017 ou superior (Vou usar o 2019)
* .NET Core 2.1 ou superior (Vou usar o 2.1)
* MySQL 8 ou superior (Vou usar o 8.0.16)
* Advanced REST Client ou qualquer software para testar requisições HTTP
* Conhecimento básico sobre MVC.
* Café, muito café, você vai precisar.

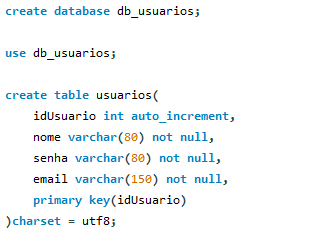
Com essas ferramentas já instaladas e o café pronto vamos iniciar criando nossa base de dados que iremos consumir na API.

**OBS:** *Estou utilizando o MySQL, porém o mesmo código funciona para qualquer outro banco relacional, fique à vontade para escolher o de sua preferência.*

Para não complicar e deixar maçante, criei uma base de dados simples com os seguintes campos:

* Id
* Nome
* Senha
* Email

Código para criação do banco:



**Entendendo o REST e as APIs RESTful**

Antes de sair codificando nossas API por aí, vamos primeiro entender o que é o padrão REST e onde ele é utilizado, já escrevi sobre isso em outro artigo, porém irei dar uma rápida revisada sobre seu conceito, caso você já saiba o que é uma API REST e onde ela é utilizada, pode pular pro próximo tópico sem problemas.

Apesar de parecer um bicho de 7 cabeças, REST, não passa de um modelo para projetar softwares que necessitam de comunicação via rede, ou seja, esse padrão é utilizado para enviar / receber informações utilizando o protocolo HTTP, criado por Roy Fielding (um dos principais criadores do protocolo HTTP). Parece complicado? Mas vamos lá que vai ficar fácil!

Em seu literal, REST significa “Representational State Transfer” ou “Representação do Estado de Referência”, como já mencionado, ele utiliza o padrão HTTP, para sua comunicação, nisso **ele tem 4 principais agentes, são eles: POST, GET, PUT, DELETE.**

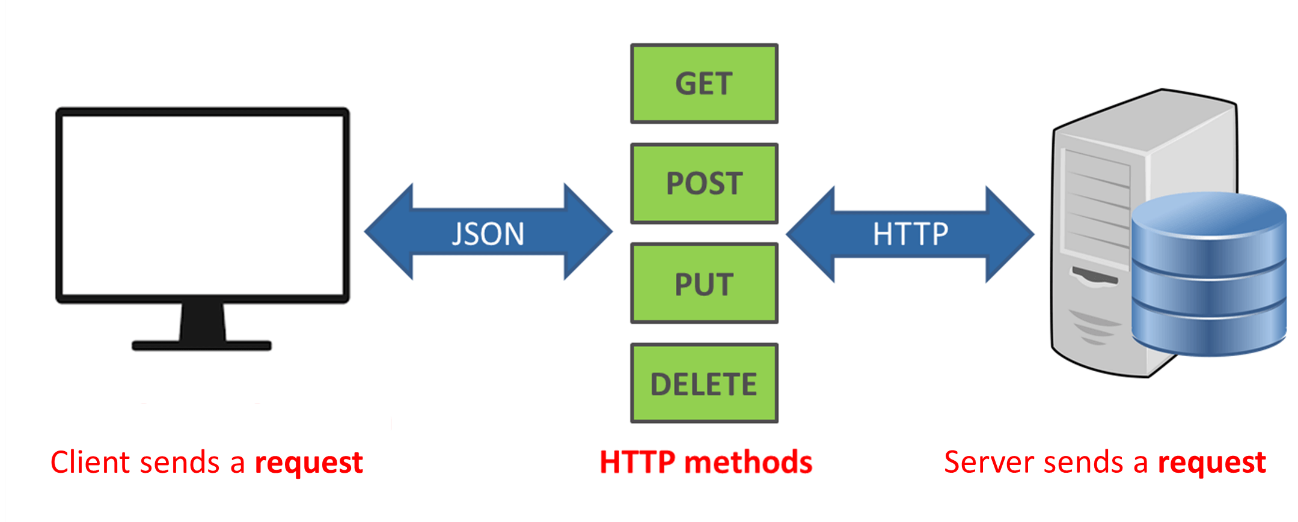
**POST**: O **POST** é utilizado quando precisamos enviar dados para algum lugar, pense no post como uma caixa, na qual nossa aplicação coloca todos os seus dados e envia para alguma outra aplicação ou base de dados, deixando assim os dados protegidos e bem estruturados.

**GET**: O **GET** é utilizado quando precisamos pegar dados de algum lugar, pense nele como uma vara de pesca, quando precisamos buscar nossos dados (no caso o peixe), utilizamos a vara de pesca (o **GET**) para busca — lo e trazer — lo já pronto para comer (tratar os dados).

**DELETE**: Com um nome bastante intuitivo, o **DELETE** geralmente é utilizado quando precisamos apagar algo da nossa base de dados ou excluir algum recurso da nossa aplicação.

**PUT**: O **PUT** é utilizado quando precisamos alterar algum dado ou recurso da nossa aplicação ou da nossa base de dados, pense no **PUT** como uma borracha, na qual você usa só quando precisa reescrever algum texto (dados).

Então quando nós precisamos acessar alguma base de dados, fazemos um “**Request**” (requisição) para nossa API, e ela nos retorna os dados em JSON ou XML como “Response” (resposta), imagine que minha aplicação precisa de todos os nomes cadastrados, então minha aplicação faz o seguinte processo:

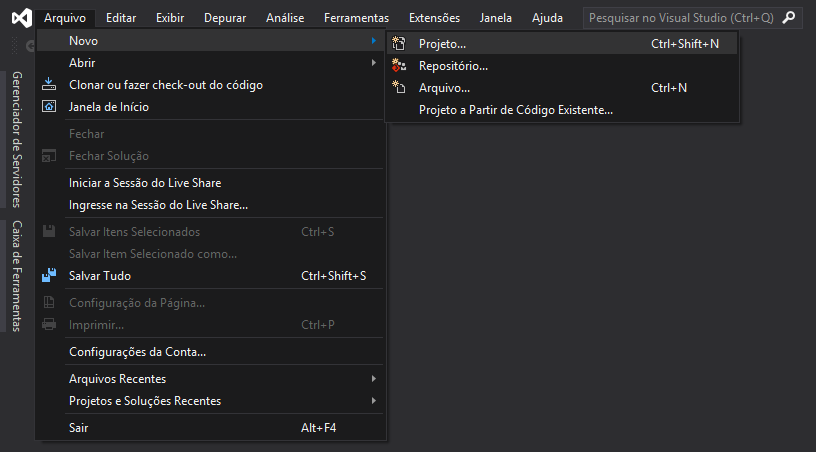
<https://phpenthusiast.com/theme/assets/images/blog/what_is_rest_api.png>

**Criando o projeto e configurando suas dependências**

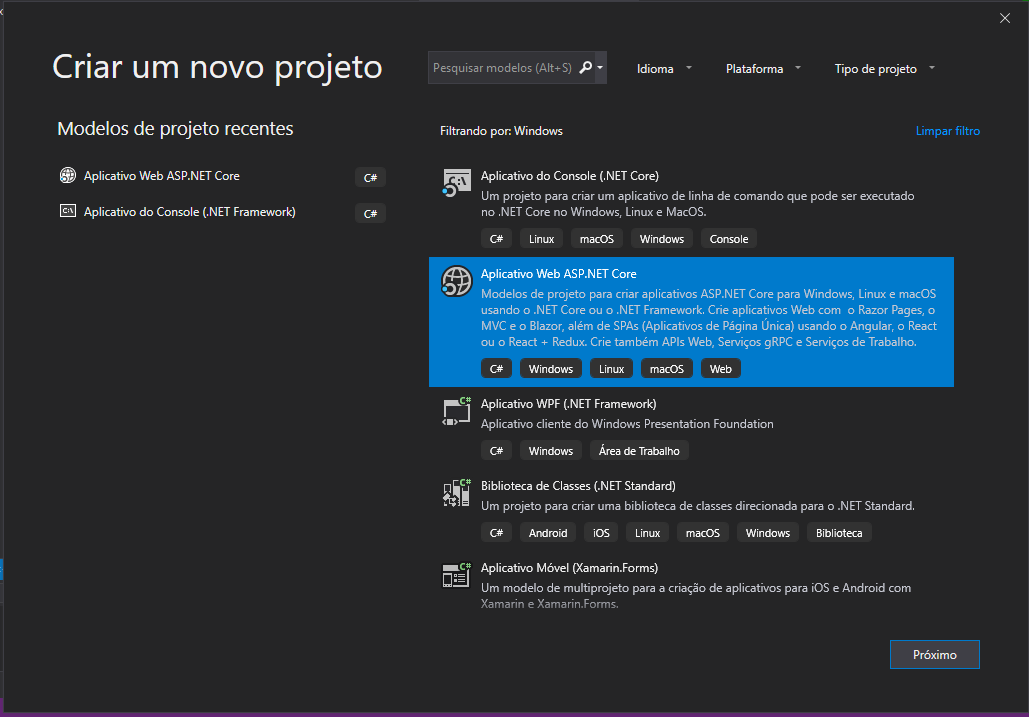
Vamos agora criar nosso projeto e configura – lo para que funcione de acordo com nossas necessidades.

Abra seu Visual Studio 2017 ou superior.

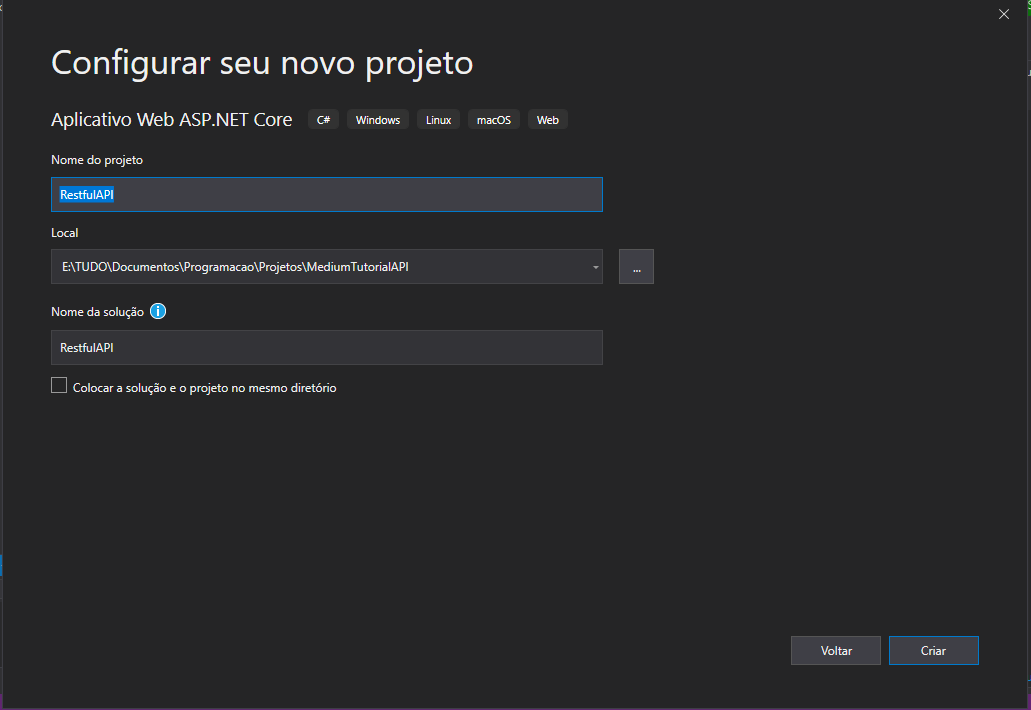
Com o Visual Studio já abeto vá em **Arquivo > Novo > Projeto** ou digite  
**Ctrl + Shift + N**

****

Selecione o template: **Aplicativo Web ASP .NET CORE** e clique em **próximo**.



Dê um nome para seu projeto, escolhe o lugar onde ele irá ficare clique em **próximo**.



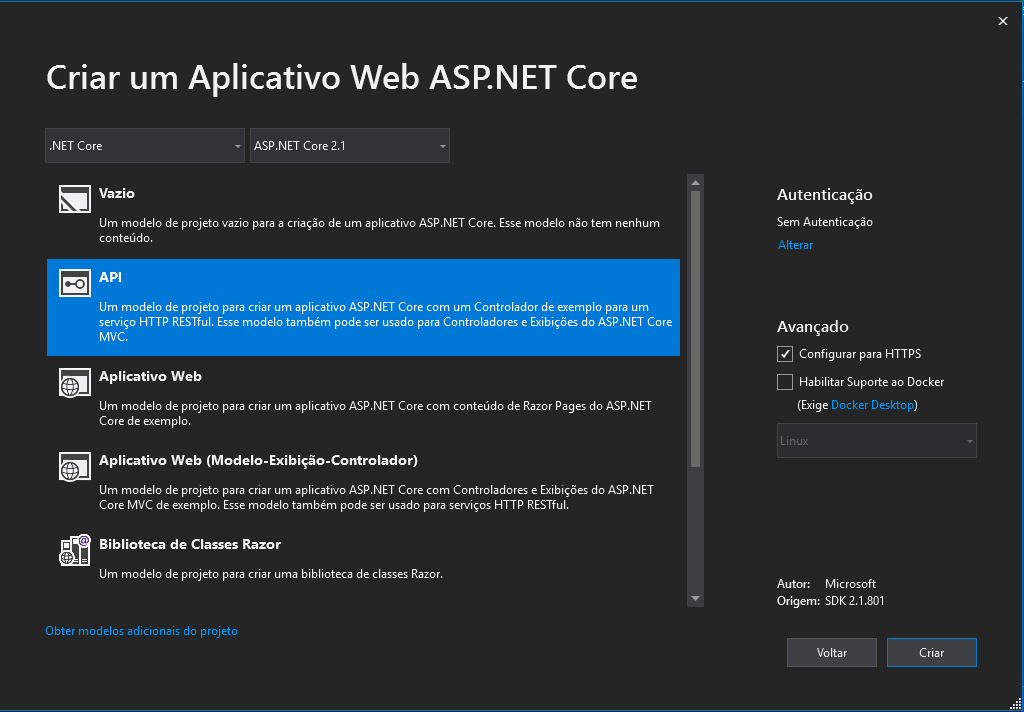
Aqui você irá selecionar a versão do .NET Core e o template para a aplicação WEB.

Selecione o template **API**.

Deixe por padrão **.NET Core.**

Na versão deixe por padrão **2.2.1 ou superior.**

E deixe marcado para **configurar HTTPS.**

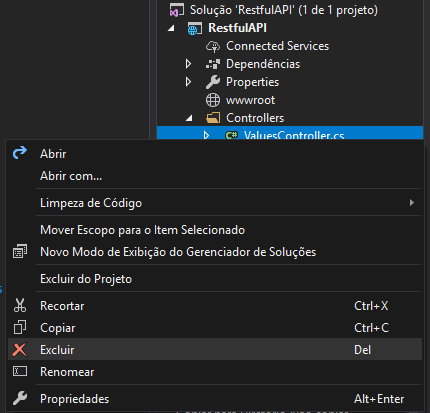
****

Agora vamos criar o **controller** que iremos utilizar para criar nossas rotas de requisição REST.

Por padrão o projeto Web API já vem com um **controller** chamado **ValuesController** já criado com alguns exemplos de métodos com os verbos **HTTP.**

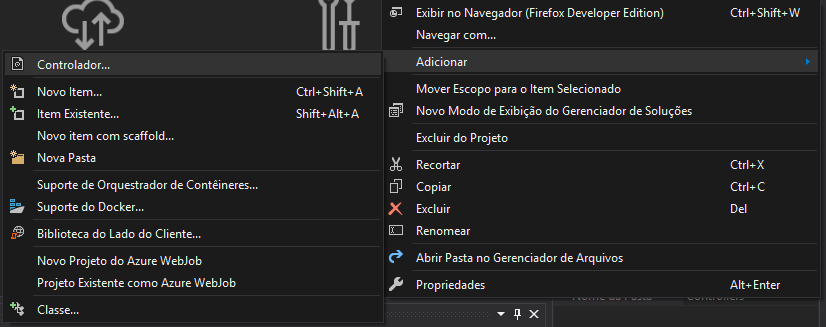
Vamos deletar esse **controller** e criar um totalmente em branco, para termos um melhor entendimento da funcionalidade da **API**.

No gerenciador de soluções do Visual Studio clique na pasta **Controllers** e depois clique com o botão direito no arquivo **ValuesController.cs** aperte **del** ou clique em **excluir**.

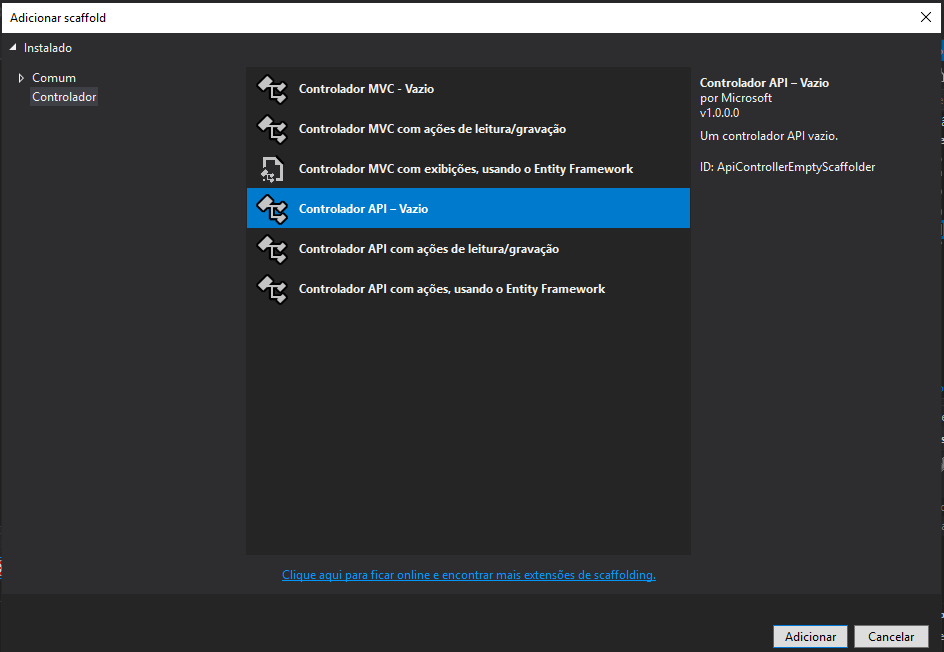


Agora nós vamos adicionar o nosso próprio **controller** e configura-lo para poder criar nossas rotas de requisição para a **API**.

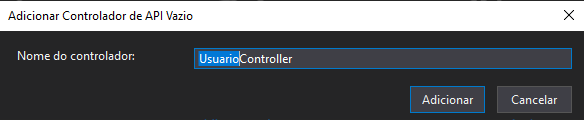
Na **gerenciador de soluções** clique com o botão direito na pasta **controllers > adicionar > controllador.**

****

Selecione o template **API - Vazio** e clique em **adicionar.**

****

Dê o nome de **UsuarioController** e clique em **adicionar,** após isso será gerado um arquivo **UsuarioController.cs** que será onde nós iremos colocar nossas **rotas** e o que elas irão fazer.



Abra o arquivo **UsuarioController.cs**, e vamos criar o método Index, que será a tela padrão que irá aparecer na API na hora que compilarmos o projeto.

Digite o seguinte código dentro da classe **UsuarioController**:

**OBS**: *A mensagem pode ser a que você quiser, aqui só é um exemplo rs.*

public IActionResult Index()

{

return Ok("Index API - medium.com/@lucas.eschechola");

}

Da mesma forma que já vem criado um **controller** chamado **values**, a API já vem configurada para acessar esse **controller** por padrão.

Como não queremos que ele acesse **values** e sim **usuário** nós precisamos alterar isso no arquivo **launchSettings.json.**

Vá em **Properties >** **launchSettings.json.**

Na linha **15** altere de:

"launchUrl": "api/values",

Pra:

"launchUrl": "api/usuario",

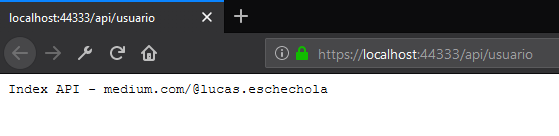
Na linha **23** altere de:

"launchUrl": "api/values",

Pra:

"launchUrl": "api/usuario",

Apertando **F5** ou clicando em **Iniciar** teremos a seguinte tela:



Pronto, agora nossa api já está criada e configurada para rodar por padrão o controller **Usuario**, na **próxima parte do artigo** nós iremos adicionar o **Entity** **Framework** ao nosso projeto e criar as nossas **5 rotas** para acesso aos dados.

**Construindo APIs Restful com .NET Core usando Entity Framework e JWT  
Parte – II**

Olá desenvolvedor, tudo bem? Fico feliz que tenha conseguido chegar até aqui, nessa parte nós iremos configurar o Entity Framework e criar nossos métodos de acesso a dados.

Essa é a segunda parte de uma pequena sequência de artigos, recomendo que você de uma lida rápida na parte onde eu explico alguns conceitos e faço as configurações iniciais do projeto.

**Entendendo o Entity Framework**

Antes de instalar o Entity Framework no nosso projeto e criar os nossos modelos, precisamos primeiro entender o que ele é e como ele vai nos ajudar (e muito) no acesso aos dados.

O Entity Framework atualmente é uma das principais ferramentas de Mapeamento Objeto Relacional ou ORM, ele nos ajuda a transformar nossas classes em tabelas e a manipular os dados dentro delas utilizando classes.

Teoricamente parece bem complexo de entender, porém vou tentar deixar as coisas um pouco mais simples, vamos fazer uma analogia.

Imagine que você tem a seguinte tabela no seu banco de dados:  
**Tabela: usuario**

|  |  |
| --- | --- |
| Id | Int |
| Nome | varchar(80) |
| Senha | varchar(80) |

Para fazer uma inserção de dados nessa tabela, normalmente utilizaríamos a query:

INSERT INTO usuario VALUES(default, ‘Lucas’, ‘Eschechola’);

Ao fazer nosso software realizar essa query na nossa base de dados, a gente pode ter alguns problemas, por exemplo:

* Formato de dados inválidos;
* Injeção de SQL;
* Querys erradas;
* Exceções;
* Querys gigantes e complexas;

Afim de evitar essa série de problemas, utiliza – se o Entity Framework para facilitar o acesso e tratamento dos dados. Mas calma, não estou falando que o jeito tradicional (mostrado acima) está errado, apenas estou mostrando o porquê do Entity ser utilizado.

Continuando nossa analogia, imagine que temos também uma classe no nosso sistema com os seguintes atributos.

public class Usuario  
{  
 public int Id { get; set; }  
 public string Nome { get; set; }  
 public string Senha { get; set; }  
}

Do jeito tradicional teríamos que tirar os dados da classe e coloca – los na nossa query e ficaria da seguinte forma:

var usuario = new Usuario  
{  
 Nome = “Lucas”,  
 Senha = “Eschechola”  
};  
  
query.Comando = String.Format(“INSERT INTO usuario VALUES(default, ‘{0}’, {1}”, usuario.Nome, usuario.Senha);

query.ExecuteNonQuery();

Já com o Entity Framework, nós não precisamos montar a query e executar diretamente no nosso banco de dados, podemos simplesmente passar a classe já com os dados e pedir para ele inserir no nosso banco, evitando os problemas já mencionados. Ficaria da seguinte forma.

private readonly UsuarioDBContext \_contexto;

var usuario = new Usuario  
{  
 Nome = “Lucas”,  
 Senha = “Eschechola”  
};  
  
\_contexto.Add(usuario);  
\_contexto.SaveChanges();

Aqui o Entity instanciou um objeto chamado **UsuarioDBContext** , essa classe contém as configurações e informações do nosso banco de dados, ela é usada para acesso.

Para o método **Add()** foi passado o objeto com os dados que queríamos inserir, e ele foi o responsável pelo tratamento e inserção dos dados, abstraindo do sistema o tratamento e acesso direto a base de dados, deixando o código mais limpo, com menos falhas e mais fácil de leitura.

**Resumindo:**

Método tradicional (ADO .NET)

**Dados do usuário > Leitura > Tratamento > Inserção**

Entity Framework

**Dados do usuário > Leitura > Entity(Tratamento e Inserção)**

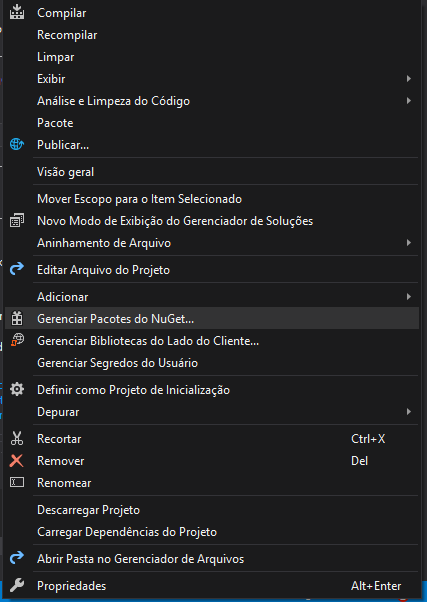
**Adicionando o Entity Framework ao projeto**

Agora nós vamos importar o Entity para o projeto que havíamos criado no artigo anterior, para isso iremos utilizar o pacote:

**Pomelo.EntityFrameworkCore.MySql**

**OBS:** *No início do primeiro artigo eu tinha comentado que funcionaria com qualquer base de dados relacional e de fato, funciona, você só precisa procurar qual o pacote do Entity Framework baixar para sua base de dados, uma pesquisa rápida no Google já resolve seu problema.*

Clique com o botão direito no projeto e depois em **Gerenciar Pacotes do NuGet.**

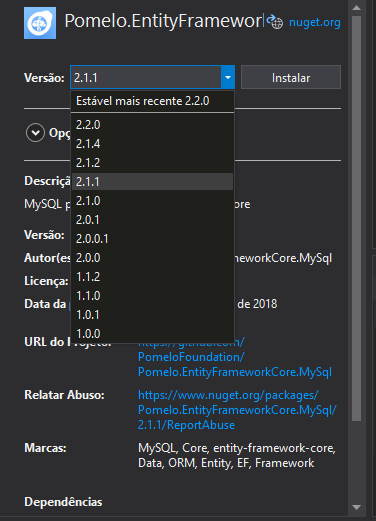
****

Essa opção vai abrir o gerenciador de pacotes do nosso projeto, no canto superior esquerdo, na barra de pesquisa, pesquise por:

**Pomelo.EntityFrameworkCore.MySql**

**OBS:** *Você deve instalar a versão compatível com o .NET Core que está utilizando no projeto, no nosso caso é a versão 2.1, então instale a versão 2.1 desse pacote.*

No canto direito do gerenciador de pacotes mude a versão para 2.1 e clique em instalar:



Com o Entity já instalado, agora nós vamos criar nosso modelo de classe que será o que iremos utilizar pra receber e enviar os dados para o banco, mas antes disso preciso explicar as 2 formas de se lidar com uma base de dados utilizando o Entity Framework: Code First e Database First.

A diferença entre eles é bem simples:

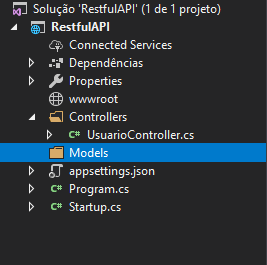
**Code First**: Code First em sua tradução literal significa “Código Primeiro” e é exatamente isso que o Entity faz. As tabelas da base de dados são criadas a partir das classes modelos, então se o sistema tiver as classes modelo Usuario e Produto, **serão criadas 2 tabelas, a tabela Usuario contendo os mesmos campos da classe e a tabela Produto também contendo os mesmos campos da classe.**

**Database First:** Database First em sua tradução literal significa “Base de dados Primeiro” e ao contrário do Code First, você já tem suas tabelas e bases de dados criadas, e o Entity cria pra você as classes modelos de acordo com suas bases de dados, **então se você tem a tabela Usuario e Produto, o Entity vai criar a classe Usuario e Produto contendo os mesmos campos da base de dados.**

Como no nosso caso já temos nossa base de dados criada, vamos utilizar o Database First para que ele crie modelos de classe de acordo com nossa tabela.

Clique com o botão direito no projeto > Adicionar > Nova Pasta e de o nome **Models** para essa pasta**.**

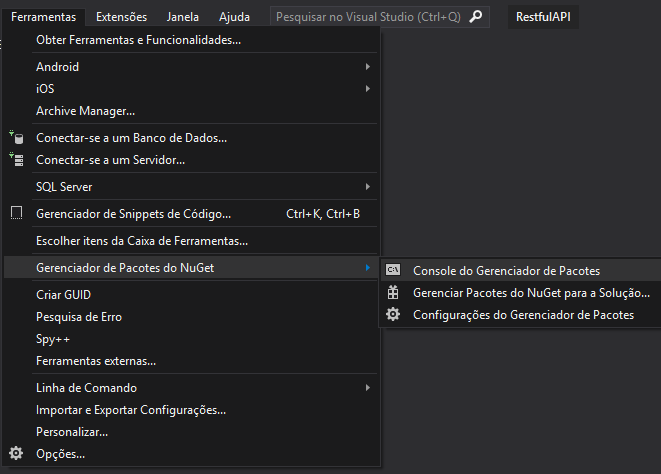
A estrutura do seu projeto deve ficar mais ou menos parecida com essa:



Tudo pronto para receber nosso modelo de classe, que no caso será só um (Usuario).

Vamos abrir o console do gerenciador de pacotes para colocarmos o comando que vai gerar nosso modelo automaticamente.

No Visual Studio, vá em **Ferramentas > Gerenciador de Pacotes do NuGet > Console do Gerenciador de Pacotes**



Irá abrir um pequeno console na parte de baixo do Visual Studio, nele digite o seguinte comando:

Scaffold-DbContext “Server=localhost;Database=db\_usuarios;Uid=root;Pwd=;” Pomelo.EntityFrameworkCore.MySql -OutputDir Models

Será criado o arquivo db usuariosContext.cs e o arquivo Usuario.cs na pasta Models.

O arquivo db usuariosContext.cs é onde contém as configurações das tabelas e de conexão com o banco de dados, porém por uma questão organizacional, vamos mudar alguns detalhes

Clique com o botão direito no arquivo db usuariosContext.cs > **Renomear**.

Troque o nome da classe para UsuarioContext.cs.

Clique na classe e aperte Ctrl + F.

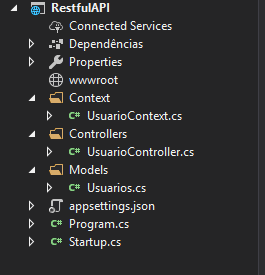
E altere todas as ocorrências de db usuariosContext.cs para UsuarioContext.



Agora por questão de organização vamos guardar o arquivo UsuarioContext.cs em uma pasta separada.

Clique com o botão direito no projeto > Adicionar > Nova Pasta > de o nome de **Context.**

Agora mova o arquivo UsuarioContext.cs para a pasta Contex, após isso a estrutura do seu projeto deve ficar parecida com isso:



**Segurança**

Por padrão o Entity Framework deixa nossa string de conexão no arquivo **UsuarioContext.cs**, mas, por questão de padrão de projeto e segurança vamos alterar isso.

Abra o arquivo **UsuarioContext.cs** e apague o método **OnConfiguring**

**(Linha 20 até Linha 28)**

No Gerenciador de Soluções abra o arquivo **appsettings.json**

Será aqui que nós iremos colocar as informações de conexão da nossa API.

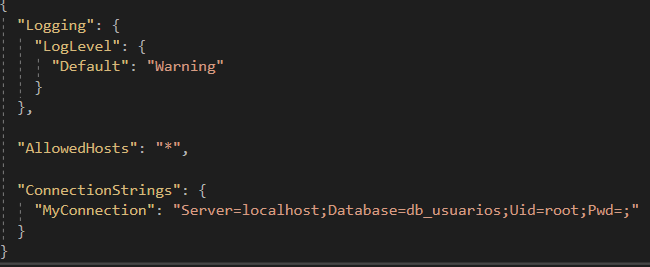
Coloque uma virgula no fim de **"AllowedHosts": "\*"** e digite o seguinte código:

"ConnectionStrings": {

"MyConnection": "Server=localhost;Database=db\_usuarios;Uid=root;Pwd=;"

}

Seu arquivo **appsettings.json** ficará semelhante a isso:



Agora vamos finalizar adicionando essa configuração no arquivo **startup.cs.**

Abra o arquivo **startup.cs** e localize o método **ConfigureServices.**

Precisamos adicionar as namespaces que vamos utilizar, então adicione:

using Microsoft.EntityFrameworkCore;

Dentro de **ConfigureServices** digite o seguinte código:

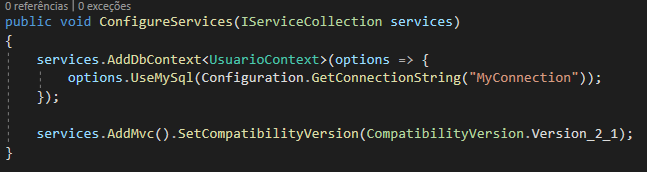
services.AddDbContext<UsuarioContext>(options => {

options.UseMySql(Configuration.GetConnectionString("MyConnection"));

});

**OBS:** *Provavelmente vai dar erro em UsuarioContext, você precisa adicionar a namespace dele, no nosso caso é using* ***RestfulAPI.Models.***

Seu método **ConfigureServices** deve ficar semelhante a esse:



Agora já temos configurado o Entity Framework e os nossos modelos de classes criados, na próxima parte do artigo iremos criar a classe aplicação onde vai receber os dados do **controller** e inserir no banco de dados.