**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.

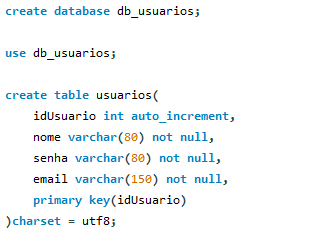
Com essas ferramentas já instaladas e o café pronto vamos iniciar criando nossa base de dados que iremos consumir na API.

**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.**

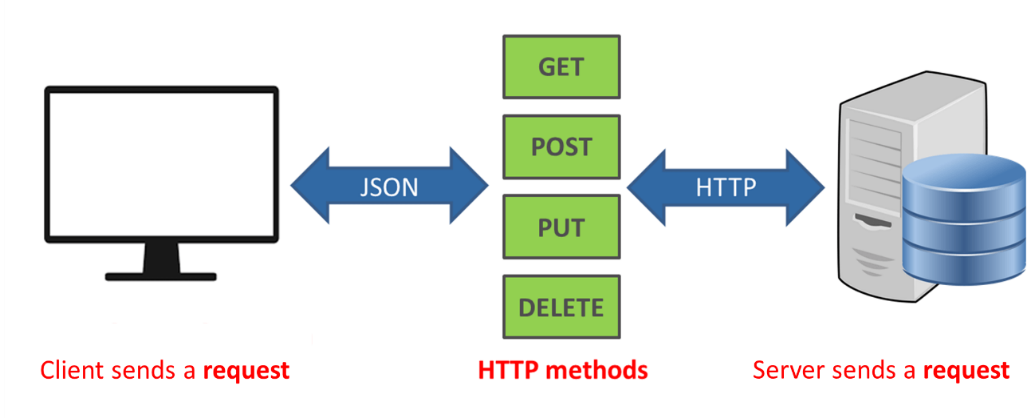
**POST**: O **POST** é utilizado quando precisamos enviar dados para algum lugar, pense no post como uma caixa, na qual nossa aplicação coloca todos os seus dados e envia para alguma outra aplicação ou base de dados, deixando assim os dados protegidos e bem estruturados.

**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>

Uma API Rest funciona através de rotas, que identificam o que nós queremos consumir dela, o que iremos criar aqui funcionará da seguinte forma:

**“Quero inserir um novo usuário”**

*https://localhost:porta/api/controller/Inserir*

Para podermos inserir um usuário iremos fazer uma requisição através do método **POST** enviando os dados do usuário para a rota acima.

**“Quero ver todos os usuários”**

*https://localhost:porta/api/controller/TodosOsUsuarios*

Para podermos ver todos os usuários iremos fazer uma requisição através do método **GET** para a **rota** acima e ele irá nos retornar um JSON ou XML com os dados.

**“Quero deletar um usuário”**

*https://localhost:porta/api/controller/Deletar*

Para podermos deletar um usuário iremos fazer uma requisição através do método **DELETE** enviando o id do usuário para a rota acima.

**“Quero alterar um usuário”**

*https://localhost:porta/api/controller/Alterar*

Para podermos alterar um usuário iremos fazer uma requisição através do método **PUT** enviando os dados do usuário alterado para a rota acima.

***Todas as rotas foram apenas exemplos, na nossa API vai ficar um pouco diferente.***

**Entendendo JSON e XML**

JSON e XML são formatos de arquivos para a troca de dados na web, ambos formatos utilizados em APIs e no nosso caso não será diferente, como o XML é um formato mais antigo e um pouco mais complexo de se manipular, estarei utilizando JSON.

JSON significa **Javascript Object Notate,** e ele não tem esse nome atoa, o seu formato de armazenar os dados é idêntico ao um objeto javascript.

**OBS:** *Não confunda objeto javascript com objeto “Classe”, eles são parecidos, mas tem suas particularidades.*

Vou apresentar uma simples comparação de como iremos tratar o JSON na nossa aplicação.

Imagine que você tem a seguinte classe:

public class Produto()

{

int IdProduto { get; set; }  
 string NomeProduto { get; set; }

double PrecoProduto { get; set; }

}

Void Main()  
{  
 var novoProduto = new Produto  
 {  
 IdProduto = 0,  
 NomeProduto = “Processador I7”,  
 PrecoProduto = 1800  
 }  
}

Agora quando nós formos **serializar** o objeto **novoProduto** ele se transformaria no seguinte documento JSON:

**Serializar:** *Transformar os dados das variáveis em um documento JSON*

{

“IdProduto”: “0”,  
 “NomeProduto”: “Processador I7”,  
 “PrecoProduto”: “1800”

}

E é esse o formato de dados que iremos enviar para nossa aplicação, quando a aplicação receber, ela vai **deserializar,** e transformar em um novo objeto **produto,** irá validar os dados e se tudo estiver correto, irá retornar uma mensagem de sucesso, caso tenha algo errado irá retornar uma mensagem de erro.

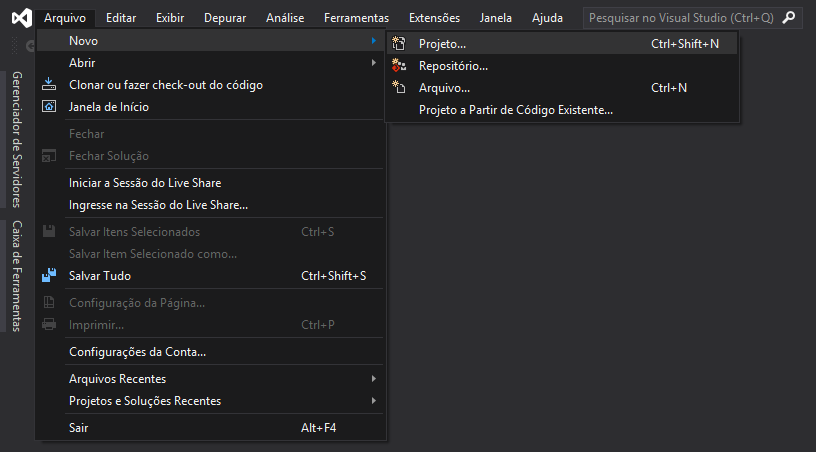
**Deserializar:** *Extrair os dados de um documento JSON, no nosso caso, extrair e colocar em um objeto.*

**Criando o projeto e configurando suas dependências**

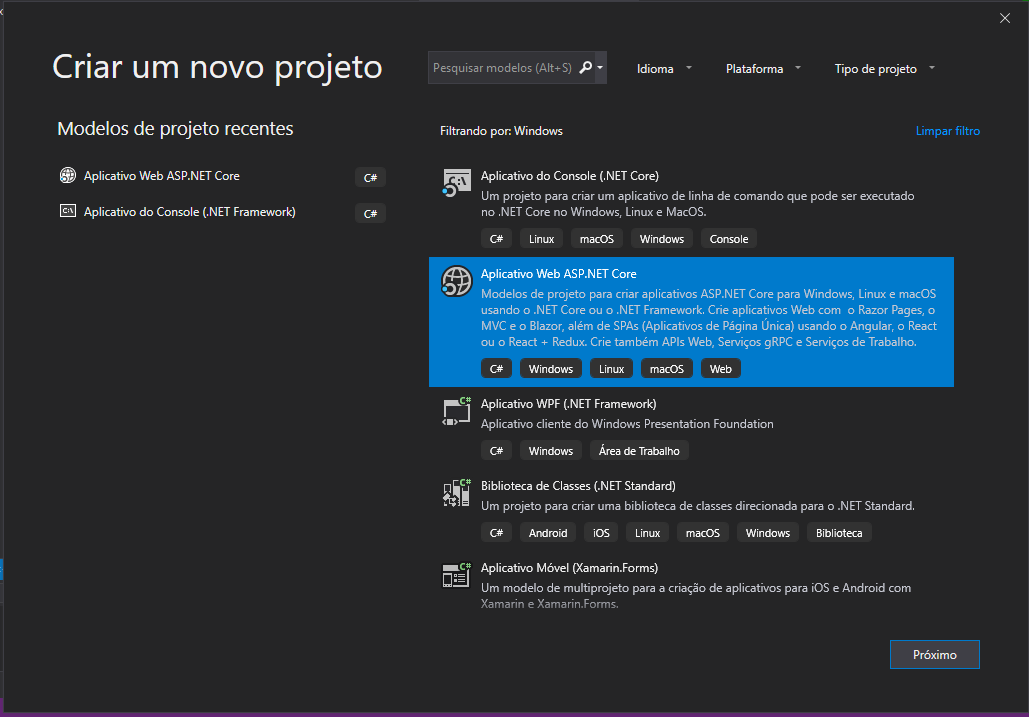
Vamos agora criar nosso projeto e configura – lo para que funcione de acordo com nossas necessidades.

Abra seu Visual Studio 2017 ou superior.

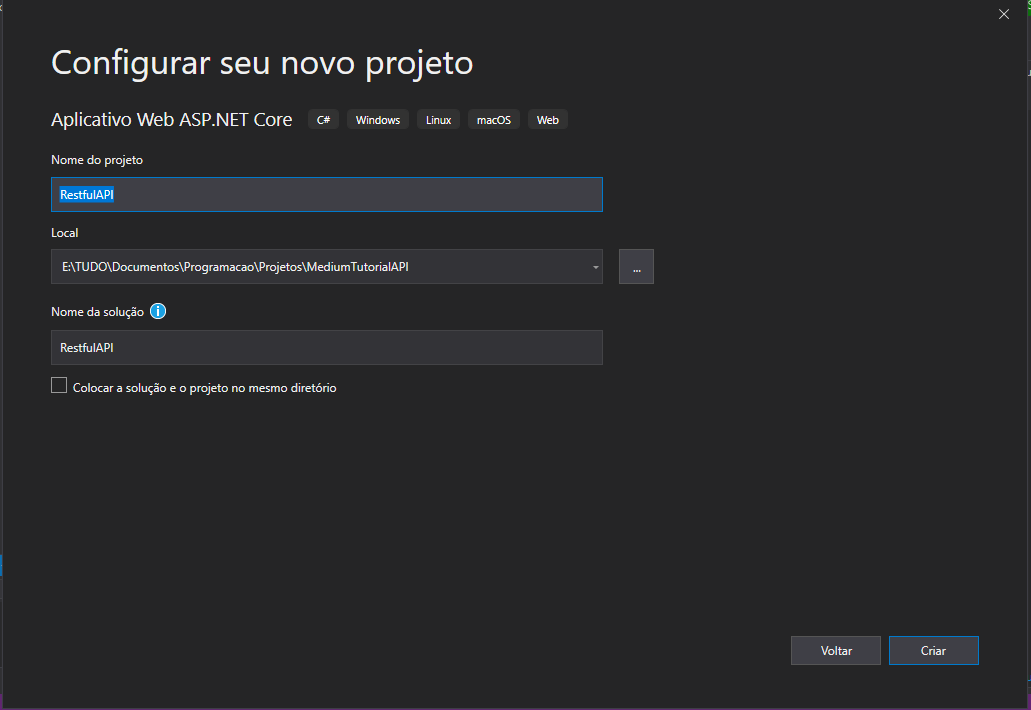
Com o Visual Studio já abeto vá em **Arquivo > Novo > Projeto** ou digite  
**Ctrl + Shift + N**

****

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**.



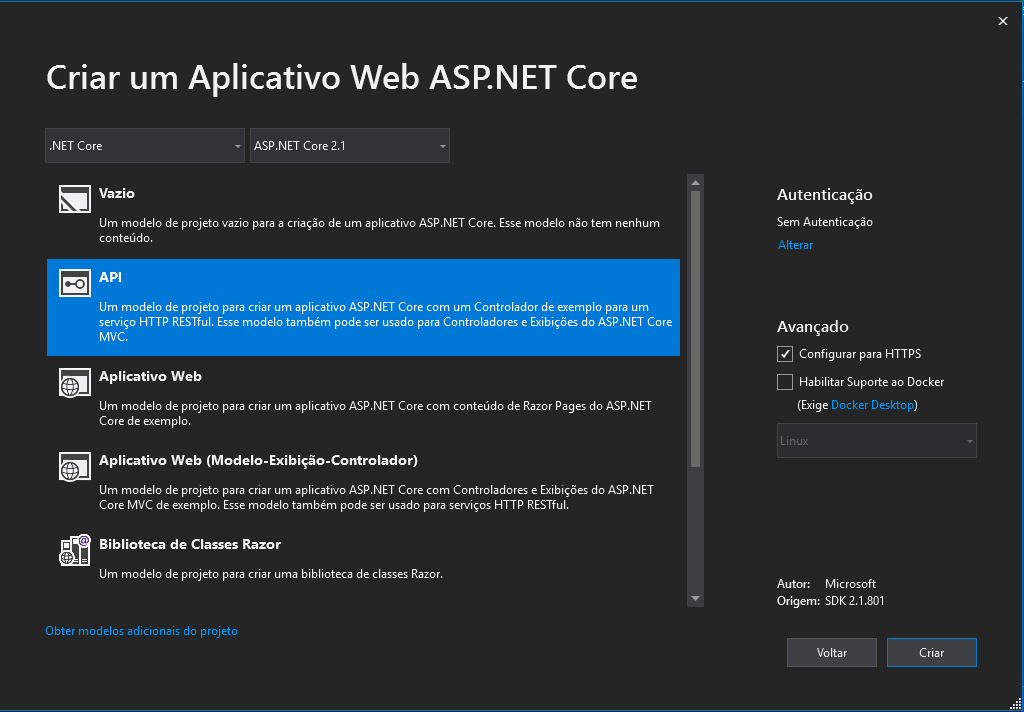
Aqui você irá selecionar a versão do .NET Core e o template para a aplicação WEB.

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.**

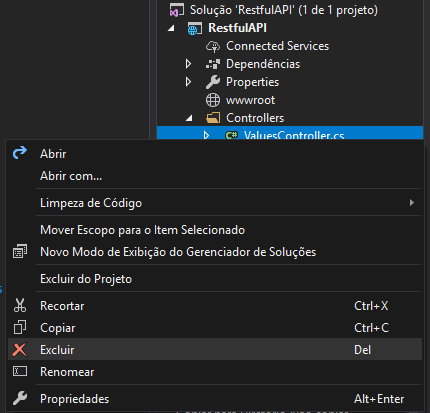
****

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.**

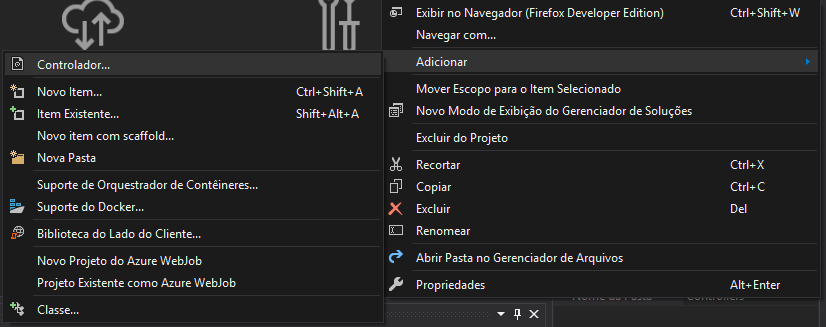
Vamos deletar esse **controller** e criar um totalmente em branco, para termos um melhor entendimento da funcionalidade da **API**.

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**.

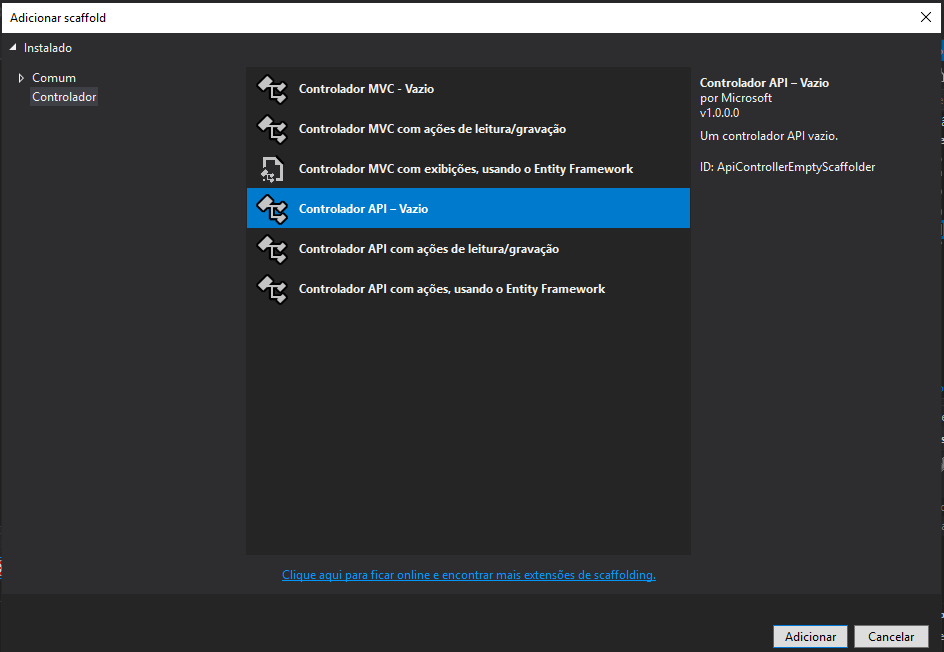


Agora nós vamos adicionar o nosso próprio **controller** e configura-lo para poder criar nossas rotas de requisição para a **API**.

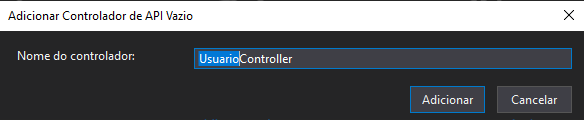
Na **gerenciador de soluções** clique com o botão direito na pasta **controllers > adicionar > controllador.**

****

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",

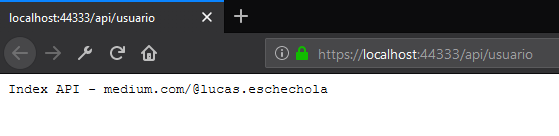
Na linha **23** altere de:

"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.

*Caso você esteja perdido, essa é a terceira parte de uma sequência de artigos no qual ensinam a criar uma API Restful utilizando .Net Core, se você ainda não leu as outras partes, recomendo dar uma breve olhada para o melhor entendimento.*

**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)**

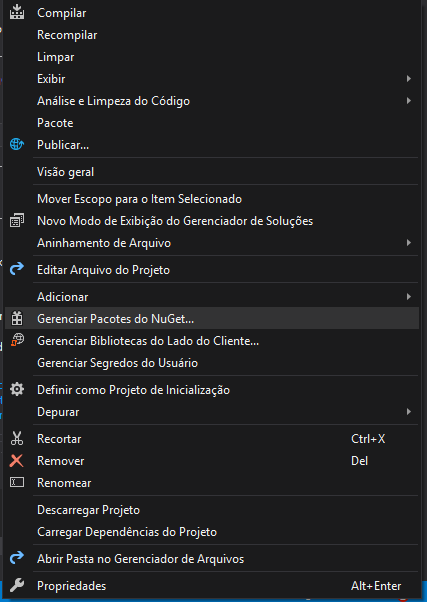
**Adicionando o Entity Framework ao projeto**

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.**

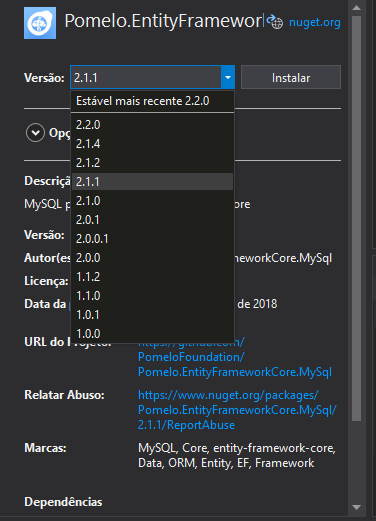
****

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:

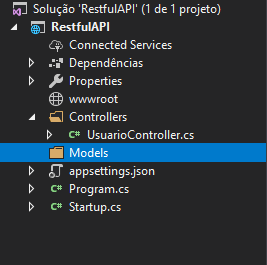
**Code First**: Code First em sua tradução literal significa “Código Primeiro” e é exatamente isso que o Entity faz. As tabelas da base de dados são criadas a partir das classes modelos, então se o sistema tiver as classes modelo Usuario e Produto, **serão criadas 2 tabelas, a tabela Usuario contendo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

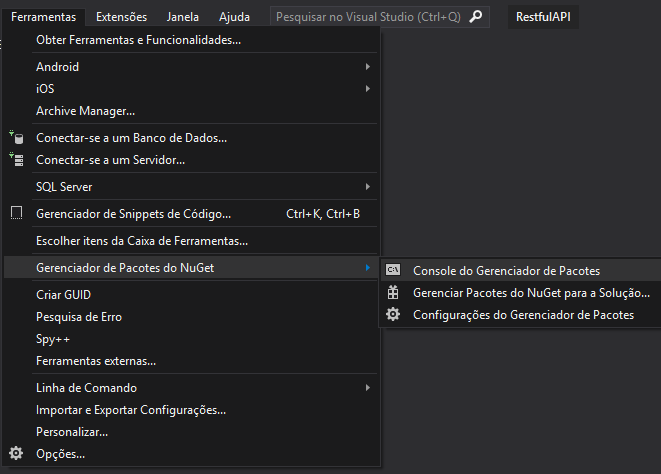
A estrutura do seu projeto deve ficar mais ou menos parecida com essa:



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.

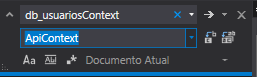
O arquivo db\_usuariosContext.cs é onde contém as configurações das tabelas e de conexão com o banco de dados, porém por uma questão organizacional, vamos mudar alguns detalhes

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

Troque o nome da classe para ApiContext.cs.

Clique na classe e aperte Ctrl + F.

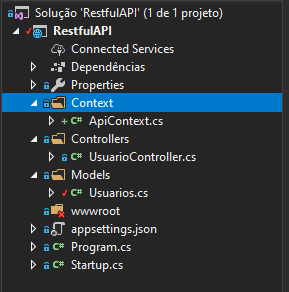
E altere todas as ocorrências de db\_usuariosContext.cs para ApiContext.



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 Context, 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:

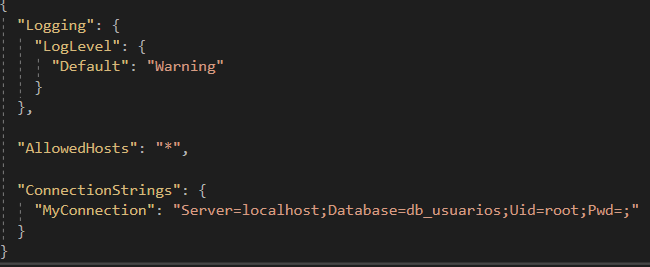
"ConnectionStrings": {

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

}

**OBS:** *Coloque os dados de conexão com seu banco, no meu caso os dados são os* ***defaults***

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:

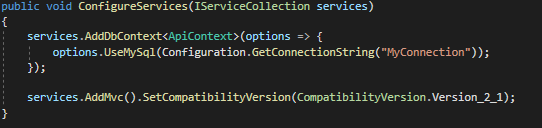
services.AddDbContext<ApiContext>(options => {

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.

**SE VOCÊ CHEGOU ATÉ AQUI, PARABÉNS E FIQUE CALMO, A PARTE MAIS CHATA E DIFÍCIL FOI ESSA!!!! HEHE.**

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

Oláááááááá desenvolvedor, tudo bem? Hoje nós iremos para a terceira parte do nosso artigo, e agora finalmente, chegou a parte lega, que é em si ver essa bagaça funcionando, nessa parte nós iremos criar nossa classe de **aplicação**, que irá se comunicar com o banco de dados e inserir nosso usuário, validar os campos da classe, utilizando **DataAnnotations** e criar nosso **controller** com os métodos HTTP para fazer finalmente o que tanto queríamos, a API funcionar.

Bom, vou começar pela parte mais fácil, que é a validação dos campos utilizando DataAnnotations, é bem fácil e prático, então pega o café e bora começar.

*Caso você esteja perdido, essa é a terceira parte de uma sequência de artigos no qual ensinam a criar uma API Restful utilizando .Net Core, se você ainda não leu as outras partes, recomendo dar uma breve olhada para o melhor entendimento.*

**DataAnnotations**

**DataAnnotations** como o nome sugere são “**anotações de dados**”, ou melhor, são formas de validar os dados de um objeto utilizando “anotações” em cada campo, facilita muito e diminui muito código, validações que geralmente iriam necessitar de muitas linhas de código podem ser feitas em 2 ou 3 e diretamente na nossa classe.

Parece complicado, mas vamos para a prática, que vai ficar bem mais claro e fácil de entender.

Abra o projeto que criamos nos **artigos anteriores** e após isso abra a classe **Usuarios**.

No topo da classe **Usuarios** adicione a seguinte referência para podermos utilizar os **DataAnnotations:**

using System.ComponentModel.DataAnnotations;

Agora que a namespace foi adicionada, nós vamos adicionar a seguinte validação nos campos:

**IdUsuario** – Nenhuma validação;

**Nome** – Não poder nulo, mínimo 3 caracteres, máximo 80 caracteres, deve conter apenas letras;

**Senha** – Não pode ser nulo ou vazio, mínimo 6 caracteres, máximo 80 caracteres.

**Email** – Verificar se é um e-mail válido, não pode ser nulo ou vazio, mínimo 10 caracteres, máximo 150 caracteres.

Para gerar as validações necessárias adicione as seguintes **anotações** na classe **Usuario**

public partial class Usuarios

{

public int IdUsuario { get; set; }

[Required(ErrorMessage = "O nome deve ser inserido")]

[MinLength(3, ErrorMessage = "O nome deve conter no mínimo 3 caracteres")]

[MaxLength(80, ErrorMessage = "O nome deve conter no máximo 80 caracteres")]

[RegularExpression(@"^[ a-zA-Z á]\*$", ErrorMessage = "O nome deve conter apenas letras.")]

public string Nome { get; set; }

[Required(ErrorMessage = "A senha deve ser inserido")]

[MinLength(6, ErrorMessage = "A senha deve conter no mínimo 6 caracteres")]

[MaxLength(80, ErrorMessage = "A senha deve conter no máximo 80 caracteres")]

public string Senha { get; set; }

[Required(ErrorMessage = "O email deve ser inserido")]

[MinLength(10, ErrorMessage = "O email deve conter no mínimo 10 caracteres")]

[MaxLength(150, ErrorMessage = "O email deve conter no máximo 80 caracteres")]

[RegularExpression(@"\w+([-+.']\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*", ErrorMessage = "O Email é inválido, insira outro.")]

public string Email { get; set; }

}

Sem desespero vamos entender cada “**anotação**” e suas propriedades:

Perceba que, em todos os campos, eu passei um parâmetro chamado:  
**ErroMenssage = “...”,** dentro dele, contém a mensagem que será retornado caso o valor da variável seja inválido.

**Required**: Essa anotação diz que o campo deve ter algum conteúdo, ou seja, não pode ser vazio, caso o contrário o modelo se torna **inválido.**

**MinLength:** Essa anotação você passa o tamanho mínimo que a propriedade deve conter, caso seja menor o modelo se torna **inválido**.

**MaxLength:** Ao contrário do MinLength, essa anotação diz o tamanho máximo que a propriedade deve conter, caso seja maior o modelo se torna inválido.

**RegularExpression:** Valida o campo através de **Regex,** a propriedade deve estar no padrão regular imposto, caso o contrário o modelo retorna inválido.

**OBS:** *Caso você não tenha conhecimento sobre* ***Regex****, ou não faz nem ideia do que seja, de uma lida no seguinte artigo antes de continuarmos:*

[*http://www.macoratti.net/net\_regex.htm*](http://www.macoratti.net/net_regex.htm)

Na hora que formos receber os dados, vamos verificar se os dados se encaixam no padrão que definimos acima, caso não se encaixe, ele não passará para a camada de aplicação da nossa API (Onde nós iremos colocar a comunicação com o banco).

Agora nós já temos nossos campos validados, vamos falar um pouco sobre a arquitetura que vamos utilizar, bem simples, porém bem prática.

**Arquitetura de comunicação**

Agora vou mostrar uma arquitetura que iremos utilizar para comunicação, neste início farei sem a parte de autorização (Autorização é a parte mais complicada, deixarei para a próxima parte do artigo), então nossa API será dividida em três partes simples:

**Recebimento > Validação > Aplicação > Contexto**.

Fazendo uma simples ilustração, ficaria da seguinte forma:

Aplicação

Request

API

Cliente

Request

Recebimento  
Dos  
Dados

Validação  
Dos  
Dados

Response

Request

Controller

Response

Response

Query no banco através do Entity Framework

Request

Banco de Dados

Contexto

Response

Response

Comunicação direta no banco de dados utilizando o Entity Framework

Qual tipo de dado e qual tabela será consultada

Em resumo, nossa aplicação passará por três principais agentes, o **controller**, a **aplicação** e o **contexto.**

No **controller** ele irá verificar se os dados enviados condizem com o modelo de classe que nós criamos, em resumo ele irá verificar se os dados estão válidos. Caso não esteja, ele irá retornar um aviso dizendo o que não está válido e os dados não passarão pra próxima camada.

A **aplicação** é a 2°camada, que será chamada caso o modelo de dados esteja correto, nela você irá chamar qual método necessita (Inserção, Exclusão, Alteração) e irá enviar os dados recebidos e já validados, caso ocorra algum erro ou exceção, ele irá retornar uma mensagem avisando qual o erro, e não passará para a última camada de contexto.

No **contexto**, é onde o **Entity Framework** entra, ele irá procurar a tabela que queremos inserir, gerar nossa query automaticamente e realizar ela no nosso banco de dados, caso ocorro algum erro de conexão ou algum erro com a base de dados, ele nos retorna uma mensagem dizendo o que estava errado.

Como o **Entity é uma ferramenta que nos auxilia, ele já cria o contexto para nós, junto com seus atributos,** então, só nos resta criar o **controller** e a **aplicação**.

**A camada de aplicação**

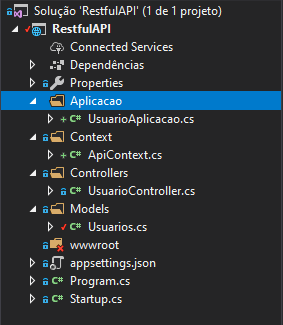
Como já mencionado, a camada de **aplicação** vai receber os dados já validados e enviar para nossa classe de **contexto**, realizar a comunicação com o banco através do Entity.

Agora nós vamos colocar em prática a arquitetura que vimos anteriormente.

Clique com o botão direito no projeto > **Adicionar** > **Nova Pasta** > nomeie a pasta para **Aplicacao**.

Clique com o botão direito na pasta **Aplicacao** > **Adicionar** > **Classe** > nomeie a classe de **UsuarioAplicacao.cs**

A estrutura do seu projeto deve ficar da seguinte forma:



Nessa classe **UsuarioAplicacao** vamos criar os seguintes métodos:

* InsertUser
* UpdateUser
* GetUserByEmail
* GetAllUsers
* DeleteUserByEmail

Utilizaremos o email como identificador principal, para os métodos de **retorno de usuário** e **deletar usuário**, o método de alteração identifica qual linha da tabela modificar através da **primary key**, no nosso caso **IdUsuario.**

Primeiro importe a namespace que iremos utilizar para acessar a classe contexto:

using RestfulAPI.Models;

Agora, adicione esta propriedade e este construtor na nossa classe **UsuarioAplicacao:**

public class UsuarioAplicacao

{

private ApiContext \_contexto;

public UsuarioAplicacao(ApiContext contexto)

{

\_contexto = contexto;

}

}

Para podermos utilizar o Entity Framework precisaremos fazer uma injeção de dependência da classe de **Contexto** e atribui - lá a uma variável já criada dentro da nossa classe **Aplicacao**.

**OBS:** *Injeção de dependência é algo um pouco complicado, então não vou abordar nessa sequência de artigos, caso não saiba o que é ou tenha alguma dúvida, recomendo este vídeo:* [*https://www.youtube.com/watch?v=A\_rPxoNO3-c*](https://www.youtube.com/watch?v=A_rPxoNO3-c)

Com nossas dependências configuradas, vamos agora criar o método de Inserção de usuários, o **InsertUser** fica da seguinte forma:

public string InsertUser(Usuarios usuario)

{

try

{

if(usuario != null)

{

var usuarioExiste = GetUserByEmail(usuario.Email);

if (usuarioExiste == null)

{

\_contexto.Add(usuario);

\_contexto.SaveChanges();

return "Usuário cadastrado com sucesso!";

}

else

{

return "Email já cadastrado na base de dados.";

}

}

else

{

return "Usuário inválido!";

}

}

catch (Exception)

{

return "Não foi possível se comunicar com a base de dados!";

}

}

Primeiro verificamos se o usuário enviado não é nulo (Pode acontecer de acabar recebendo algum usuário nulo apesar das validações) ou exista na base de dados, caso seja nulo ou exista, retornará uma mensagem de erro, senão o usuário é passado para camada de **contexto** e inserido no banco automaticamente (Amém Entity Framework).

**Caso ocorra algum erro** no processo de receber os dados, verificar se eles são nulos e enviar para a classe **contexto,** ele retornará uma mensagem de erro e não vai fazer alteração alguma na nossa base de dados.

Seguindo a mesma lógica do método anterior, vamos agora criar o método **UpdateUser**, seu código fica da seguinte forma:

public string UpdateUser(Usuarios usuario)

{

try

{

if (usuario != null)

{

\_contexto.Update(usuario);

\_contexto.SaveChanges();

return "Usuário alterado com sucesso!";

}

else

{

return "Usuário inválido!";

}

}

catch (Exception)

{

return "Não foi possível se comunicar com a base de dados!";

}

}

O método de alteração segue a mesma lógica que o de inserção, com a diferença que, ele utiliza o método **Update()** ao invés de Add() , passando o usuário com os dados já alterados.

**OBS:** *O Entity altera os dados do usuário pela primary key, então sim, você passa o usuário com todos os dados, mesmo os que não foram alterados. Sem mudar o valor da primary key.*

Agora vamos criar o método **GetUserByEmail** que vai nos retornar um **usuário**, diferente dos outros dois métodos o tipo de retorno dele vai ser um **objeto** e não uma **string.**

Caso ocorra algum erro ou ele não encontre o usuário, ele irá retornar nulo.

O código fica da seguinte forma:

public Usuarios GetUserByEmail(string email)

{

Usuarios primeiroUsuario = new Usuarios();

try

{

if (email == string.Empty)

{

return null;

}

var cliente = \_contexto.Usuarios.Where(x => x.Email == email).ToList();

primeiroUsuario = cliente.FirstOrDefault();

if (primeiroUsuario != null)

{

return primeiroUsuario;

}

else

{

return null;

}

}

catch (Exception)

{

return null;

}

}

Para encontrar o usuário com o **email** correspondente, ele utiliza uma expressão **lambda**, que diz que só quer os usuários que tem o campo **email** igual ao valor passado.

Logo irá retornar uma lista com um usuário, para pegar esse usuário único, nós utilizamos o método **FirstOrDefault().**

Agora nós vamos criar o **GetAllUsers** um método que irá nos retornar **uma lista** com todos os **Usuarios**.

Caso ocorra algum erro na comunicação com a classe contexto ou com a base de dados, ele irá retornar nulo.

O código fica da seguinte forma:

public List<Usuarios> GetAllUsers()

{

List<Usuarios> listaDeUsuarios = new List<Usuarios>();

try

{

listaDeUsuarios = \_contexto.Usuarios.Select(x => x).ToList();

if (listaDeUsuarios != null)

{

return listaDeUsuarios;

}

else

{

return null;

}

}

catch (Exception)

{

return null;

}

}

Para pegarmos todos os usuários utilizamos a expressão lambda **(x => x)** que no caso diz pro Entity que, queremos todas as linhas do banco de dados nas quais os dados tenham o formato igual ao do objeto usuário, em resumo **TODOS OS DADOS DO NOSSO BANCO.**

E por fim vamos adicionar o método **DeleteUserByEmail** que vai deletar um usuário através do **email** passado.

public string DeleteUserByEmail(string email)

{

try

{

if (email == string.Empty)

{

return "Email inválido! Por favor tente novamente.";

}

else

{

var usuario = GetUserByEmail(email);

if (usuario != null)

{

\_contexto.Usuarios.Remove(usuario);

\_contexto.SaveChanges();

return "Usuário " + usuario.Nome + " deletado com sucesso!";

}

else

{

return "Usuário não cadastrado!";

}

}

}

catch (Exception)

{

return "Não foi possível se comunicar com a base de dados!";

}

}

Aqui nós pegamos o usuário que tem o email informado através do método **GetUserByEmail** e passamos para o método **Remove** do Entity, que excluir os dados do usuário através da **primary key.**

Por fim nossa classe de aplicação **UsuarioAplicacao.cs** fica da seguinte forma:

public class UsuarioAplicacao

{

private readonly ApiContext \_contexto;

public UsuarioAplicacao(ApiContext contexto)

{

\_contexto = contexto;

}

public string InsertUser(Usuarios usuario)

{

try

{

if(usuario != null)

{

var usuarioExiste = GetUserByEmail(usuario.Email);

if (usuarioExiste == null)

{

\_contexto.Add(usuario);

\_contexto.SaveChanges();

return "Usuário cadastrado com sucesso!";

}

else

{

return "Email já cadastrado na base de dados.";

}

}

else

{

return "Usuário inválido!";

}

}

catch (Exception)

{

return "Não foi possível se comunicar com a base de dados!";

}

}

public string UpdateUser(Usuarios usuario)

{

try

{

if (usuario != null)

{

\_contexto.Update(usuario);

\_contexto.SaveChanges();

return "Usuário alterado com sucesso!";

}

else

{

return "Usuário inválido!";

}

}

catch (Exception)

{

return "Não foi possível se comunicar com a base de dados!";

}

}

public Usuarios GetUserByEmail(string email)

{

Usuarios primeiroUsuario = new Usuarios();

try

{

if (email == string.Empty)

{

return null;

}

var cliente = \_contexto.Usuarios.Where(x => x.Email == email).ToList();

primeiroUsuario = cliente.FirstOrDefault();

if (primeiroUsuario != null)

{

return primeiroUsuario;

}

else

{

return null;

}

}

catch (Exception)

{

return null;

}

}

public List<Usuarios> GetAllUsers()

{

List<Usuarios> listaDeUsuarios = new List<Usuarios>();

try

{

listaDeUsuarios = \_contexto.Usuarios.Select(x => x).ToList();

if (listaDeUsuarios != null)

{

return listaDeUsuarios;

}

else

{

return null;

}

}

catch (Exception)

{

return null;

}

}

public string DeleteUserByEmail(string email)

{

try

{

if (email == string.Empty)

{

return "Email inválido! Por favor tente novamente.";

}

else

{

var usuario = GetUserByEmail(email);

if (usuario != null)

{

\_contexto.Usuarios.Remove(usuario);

\_contexto.SaveChanges();

return "Usuário " + usuario.Nome + " deletado com sucesso!";

}

else

{

return "Usuário não cadastrado!";

}

}

}

catch (Exception)

{

return "Não foi possível se comunicar com a base de dados!";

}

}

}

**A camada de controle**

Como já explicado anteriormente, na camada de controle é onde iremos receber os dados e verificar se eles são válidos, já temos nosso arquivo **UsuarioController.cs** criado dentro da pasta **Controllers**.

Na pasta Controllers abra o arquivo **UsuarioController.cs** e digite o seguinte código:

//variavel de contexto para acesso as utilidades do entity  
private ApiContext \_contexto;

public UsuarioController(ApiContext contexto)

{

\_contexto = contexto;

}

Essa variável **\_contexto** será a dependência que iremos injetar na camada de aplicação. Nossa primeira **rota,** será a rota pra inserir um **usuário**, o código ficará da seguinte forma:

using RestfulAPI.Models;

[HttpPost]

[Route("InsertUser")]

public IActionResult InsertUser([FromBody]Usuarios usuarioEnviado)

{

try

{

if (!ModelState.IsValid || usuarioEnviado == null)

{

return BadRequest("Dados inválidos! Tente novamente.");

}

else

{

var resposta = new UsuarioAplicacao(\_contexto).InsertUser(usuarioEnviado);

return Ok(resposta);

}

}

catch (Exception)

{

return BadRequest("Erro ao comunicar com a base de dados!");

}

}

Antes de colocarmos o nome do método, falamos que ele aceita requisições do tipo **POST** usando a anotação [HttpPost]

E também indicamos qual será o nome da rota na URL utilizando [Route("InsertUser")].

Então nossa rota de acesso ao método **InsertUser,** será:

*https://link/api/Usuario/****InsertUser***

**link** será onde hospedarmos nossa API, por enquanto iremos testa – lá localmente.

**/api** é a rota padrão que definimos no **launchsetting.json** no anteriormente.

**/Usuario** é o nome do nosso **controller**.

E **/InsertUser** é o método no qual queremos enviar os dados.

Nos parâmetros nós pedimos um objeto do tipo **Usuarios** mas antes colocamos a tag [FromBody] para indicar que queremos receber os dados através de um documento e não pelo link.

Após isso verificamos se o modelo não é nulo e está valido, caso tudo esteja certo ele manda os dados para classe de **aplicação.**

Agora vamos testar nossa primeira rota, aperte **F5** para poder rodar a **API.**

**OBS:** *Verifique se seu servidor de banco de dados está ligado, caso não esteja, não irá funcionar.*

Para podermos testa nossa API utilizarei recomendo o **Advanced Rest Client,** mas caso preferir pode utilizar qualquer outro software para testar requisições HTTP, recomendo o **Postman**.

No campo **Request** **Url** cole a **Url** que abriu no seu navegador na hora que você rodou a aplicação e adicione **/InsertUser**:



**OBS**: *Não se esqueça de deixar o campo* ***Method*** *como* ***POST****.*

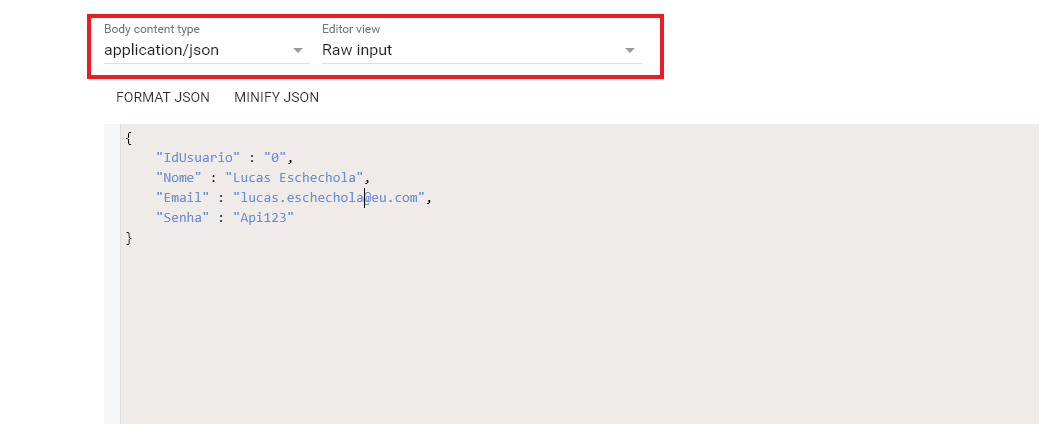
Agora nós precisamos enviar os dados em um documento **json** para a API, para isso vá na guia **Body** e digite os dados da seguinte forma:

{

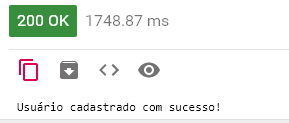
“IdUsuario” : ”0”,  
 “Nome” : ”Lucas Eschechola”,  
 “Email” : ”lucas.eschechola@eu.com”,  
 “Senha” : ”Api123”

}

**OBS:** *Recomendo esse código, você digitar, pois se você copia -ló diretamente terá que recolocar as aspas, pois elas virão invertidas.*

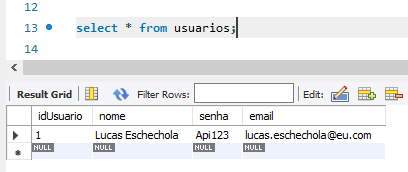


Agora é só clicar no botão **Send**  que o ARC irá fazer a requisição a nossa API e nos retornar uma resposta logo abaixo.



Caso houvesse algum erro ou o usuário já estivesse cadastrado ou o **modelo estivesse inválido,** ele iria nos retornar uma mensagem de erro que nós já colocamos anteriormente.

Vamos verificar no nosso SGBD se realmente foi adicionado um novo registro:



Agora vamos criar nosso método de alteração no controller, o código fica da seguinte forma:

[HttpPut]

[Route("UpdateUser")]

public IActionResult UpdateUser([FromBody]Usuarios usuarioEnviado)

{

try

{

if (!ModelState.IsValid || usuarioEnviado == null)

{

return BadRequest("Dados inválidos! Tente novamente.");

}

else

{

var resposta = new UsuarioAplicacao(\_contexto).UpdateUser(usuarioEnviado);

return Ok(resposta);

}

}

catch (Exception)

{

return BadRequest("Erro ao comunicar com a base de dados!");

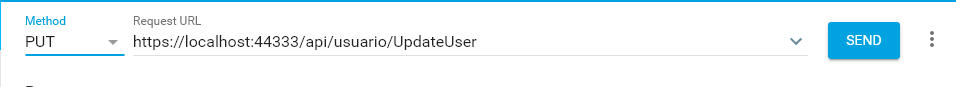
}

}

Funciona da mesma forma que o **InsertUser,** com a diferença que você tem que enivar a primary key na qual quer **alterar**.

Vamos alterar os dados do nosso usuário inserido, a primary key dele é 0.

No campo **Request** **Url** cole a **Url** que abriu no seu navegador na hora que você rodou a aplicação e adicione **/UpdateUser**:



O **body,** fica da seguinte forma:

{

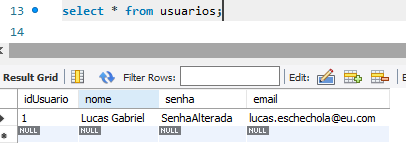
“IdUsuario” : ”1”,  
 “Nome” : ”Lucas Gabriel”,  
 “Email” : ”lucas.alterado@eu.com”,  
 “Senha” : ”SenhaAlterada”

}

A resposta:



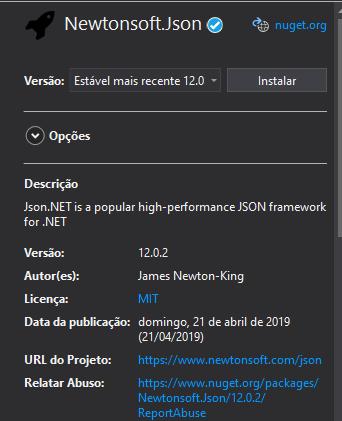
Verificando se foi alterado:



Agora vamos criar nosso método de GetUserByEmail no **controller**, mas antes precisamos instalar um pacote para Serializazção do usuário que iremos mandar.

Clique com o botão direito no projeto > **Gerenciar Pacotes do NuGet.**

Pesquise por **Newtonsoft.Json** e instale esse pacote.



Com esse pacote já instalado, vá no **controller** e inclua a seguinte **namespace:**

using Newtonsoft.Json;

O código do nosso método **GetUserByEmail** fica da seguinte forma:

[HttpPost]

[Route("GetUserByEmail")]

public IActionResult GetClienteByEmail([FromBody]string email)

{

try

{

if (email == string.Empty)

{

return BadRequest("Email inválido! Tente novamente.");

}

else

{

var resposta = new UsuarioAplicacao(\_contexto).GetUserByEmail(email);

if (resposta != null)

{

var usuarioResposta = JsonConvert.SerializeObject(resposta);

return Ok(usuarioResposta);

}

else

{

return BadRequest("Usuário não cadastrado!");

}

}

}

catch (Exception)

{

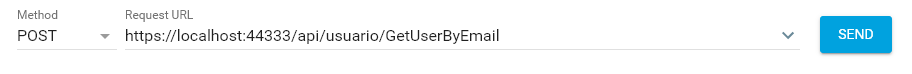
return BadRequest("Erro ao comunicar com a base de dados!");

}

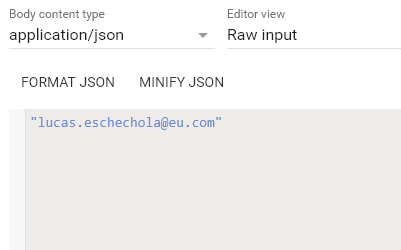
}

No **body** nós apenas enviamos o email como **string** e se caso ele achar o usuário, vai nos retornar um **JSON** com os dados deste usuário.

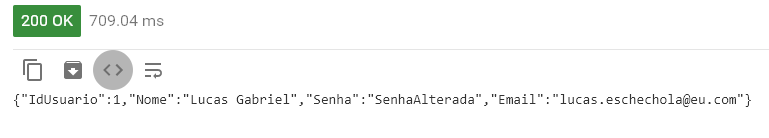
No campo **Request** **Url** cole a **Url** que abriu no seu navegador na hora que você rodou a aplicação e adicione **/GetUserByEmail**:



No **Body** fica somente o email:



A resposta:



Agora vamos criar o método **GetAllUsers,** mas antes, para melhor visualização, insira 3 novos usuários através do método **InsertUser,**  da mesma forma que foi mostrado anteriormente.

**OBS:** *Lembre – se que nossa API não cadastra e-mails iguais, então mude eles quando for cadastrar.*

Com os 3 novos usuários já cadastrados, vamos criar agora nosso método.

O código do método **GetAllUsers**, fica da seguinte forma:

[HttpGet]

[Route("GetAllUsers")]

public IActionResult GetAllClientes()

{

try

{

var listaDeUsuarios = new UsuarioAplicacao(\_contexto).GetAllUsers();

if (listaDeUsuarios != null)

{

var resposta = JsonConvert.SerializeObject(listaDeUsuarios);

return Ok(resposta);

}

else

{

return BadRequest("Nenhum usuário cadastrado!");

}

}

catch (Exception)

{

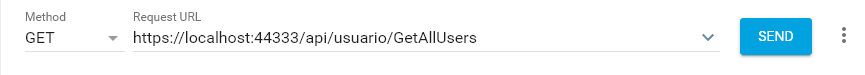
return BadRequest("Erro ao comunicar com a base de dados!");

}

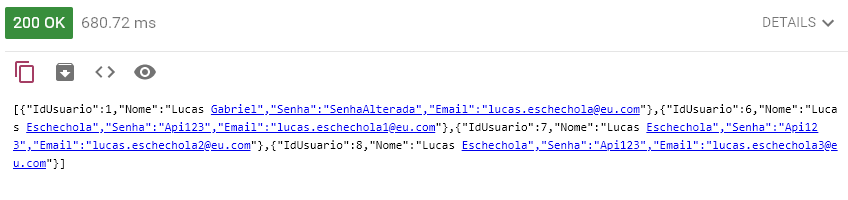
}

Quando utilizamos o método **GET** não existe body, esse método é apenas para leitura de dados, então não enviamos nada para nossa API, apenas fazemos a requisição.

No campo **Request** **Url** cole a **Url** que abriu no seu navegador na hora que você rodou a aplicação e adicione **/GetAllUsers**:



A resposta vem como uma lista de usuários no formato **JSON:**



E finalmente (me perdoem o artigo longo hehe) vamos criar o último método, o **DeleteUserByEmail.**

O código fica da seguinte forma:

[HttpDelete]

[Route("DeleteUserByEmail")]

public IActionResult DeleteUserByEmail([FromBody]string email)

{

try

{

if (email == string.Empty)

{

return BadRequest("Email inválido! Tente novamente.");

}

else

{

var resposta = new UsuarioAplicacao(\_contexto).DeleteUserByEmail(email);

return Ok(resposta);

}

}

catch (Exception)

{

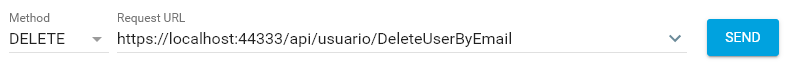
return BadRequest("Erro ao comunicar com a base de dados!");

}

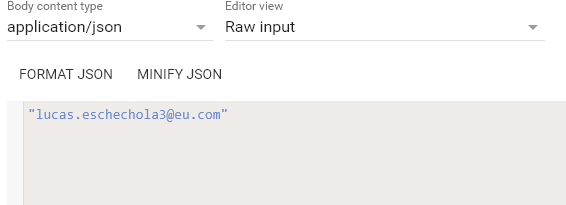
}

Seguindo o mesmo padrão dos métodos de Inserir e Atualizar, o método deletar verifica se o email foi inserido e manda para a classe **Aplicacao.**

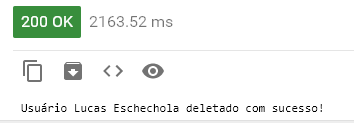
No campo **Request** **Url** cole a **Url** que abriu no seu navegador na hora que você rodou a aplicação e adicione **/DeleteUserByEmail**:



No **body** enviamos apenas o email:



A resposta:



Boaaaaaa, finalmente chegamos ao fim da nossa 3° parte da nossa sequência de artigos, na próxima etapa vou ensinar como colocar **autenticação via JWT,** espero que você tenha conseguido entender o funcionamento de uma API Rest, o porquê usarmos o Entity Framework para facilitar nossas **querys** e principalmente, **a importância de definir previamente uma arquitetura para facilitar o desenvolvimento e entendimento do próprio código.**