

Construindo uma simples API RESTful com ASP .NET

Olá desenvolvedor, tudo bem?? No artigo de hoje, vou demonstrar como criar uma **SIMPLES** API RESTful utilizando ASP .NET.

Como meu objetivo é demonstrar algo simples, não irei utilizar persistência de dados (banco de dados falando literalmente), portanto irei fazer uma classe que será nosso repositório de dados.

Ao final desse artigo espero que você tenha entendido o porquê de utilizar o padrão REST e como **começar** a desenvolver suas próprias APIs.

Mas afinal O QUE É REST / RESTful?

Antes de sairmos escrevendo código vamos entender como funciona o padrão REST e precisamos entender pra que e porque utilizar uma API.

Primeiro vamos entender o que é uma API.

A sigla API significa **A**pplication **P**rogramming **I**nterface, em geral, ela é um serviço, que vai te ajudar com algo.

Elas são muito utilizadas para acesso a base de dados, por exemplo, tenho um aplicativo mobile e quero que ele pegue todos os clientes que está numa base MySQL, é inseguro e má prática acessar a base de dados diretamente do aplicativo mobile, para isso criamos uma API que acessa os dados e retorna dados para quem requisitou, no caso o aplicativo mobile.

Para poder fazer essa requisição para uma API REST nós utilizamos rotas, onde fazemos um **REQUEST** para essa rota e ela nos retorna um **RESPONSE**, essa response pode ser uma mensagem, um dado ou um arquivo contendo dados (**JSON** ou **XML**).

Agora que já temos claro o que é uma API, vamos entender o padrão REST para podermos começar a desenvolver.

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.**

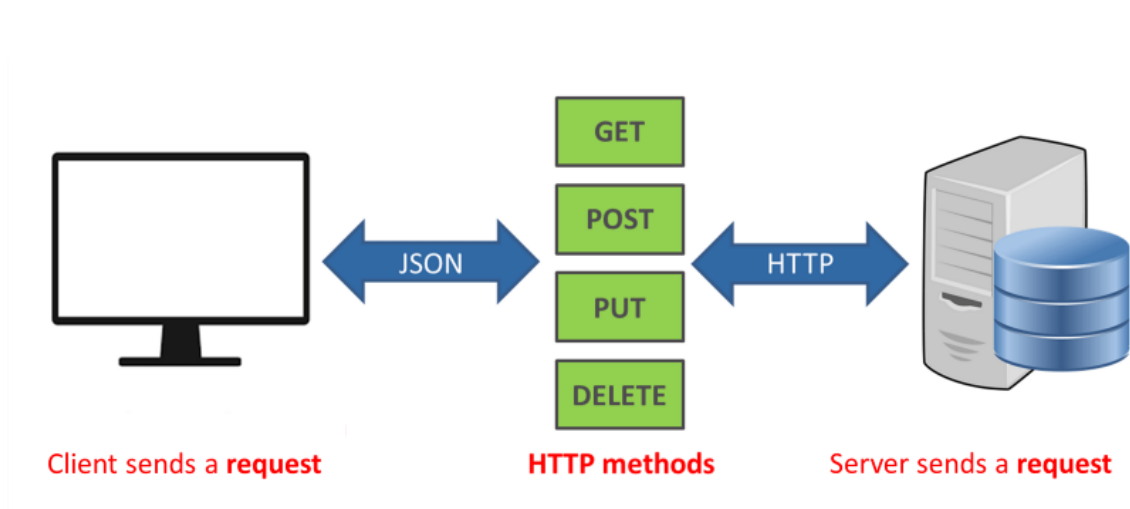
POST: O Post é utilizado quando precisamos enviar dados para algum lugar, pense no post como uma caixa, na qual nossa aplicação coloca todos os seus dados e envia para alguma outra aplicação ou base de dados, deixando assim os dados protegidos e bem estruturados.

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 traze—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>

Agora com os conceitos em mente, para ficar bem claro, vou exemplificar algumas rotas:

“Preciso inserir um usuário”

https://rota/api/usuario/Insert

Perceba que ele acesso ou o domínio da API, (No nosso caso será o localhost), **/api** indica que será um serviço **/usuario** é o **controller** que iremos utilizar e **/Insert** é o método do controller.

E no **body** nós enviamos os dados do **Usuario** em formato **JSON**.

OBS: Não me aprofundarei nos conceitos de JSON, caso você não saiba o que é um documento JSON ou qual seu formato recomendo que antes de continuar a leitura do artigo, dê uma rápida lida:

“Preciso alterar um usuário”

https://rota/api/usuario/Update

E no **body** nós enviamos os dados do **Usuario** já alterados em **JSON**.

“Preciso deletar um usuário”

https://rota/api/usuario/Delete

E no **body** nós enviamos o ID do **Usuario**.

“Preciso resgatar os dados de um usuário”

https://rota/api/usuario/GetUser

E no **body** nós enviamos o ID do **Usuario**.

Nesse método ele retorna para nós os dados do usuário em formato **JSON**.

“Preciso resgatar todos os dados de TODOS os usuários”

https://rota/api/usuario/GetAllUsers

No **body** não enviamos nada, ele nos retorna um documento contendo todos os dados de todos os usuários em formato **JSON**.

Requisitos

Sem mais delongas, vou apresentar os requisitos mínimos para poder construir a API, então, mãos à obra:

- Visual Studio 2015 ou superior com .NET Framework 4.5
Baixe em:
<https://visualstudio.microsoft.com/>
- **Advanced REST Client** ou qualquer outro software para realizar requisições **HTTP** (Recomendo o **Postman**).
Baixe em:
<https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo>
- Café, muito café, pare agora de ler e vai fazer uma xícara lá, depois continuamos.

Com o café pronto, vamos criar nosso projeto, let's code!

Criando o projeto

Abra o seu Visual Studio 2015 ou superior (estou utilizando o 2019), clique em **Arquivo > Novo > Projeto**.

Procure pelo projeto **Aplicativo Web ASP.NET**, e clique em próximo.

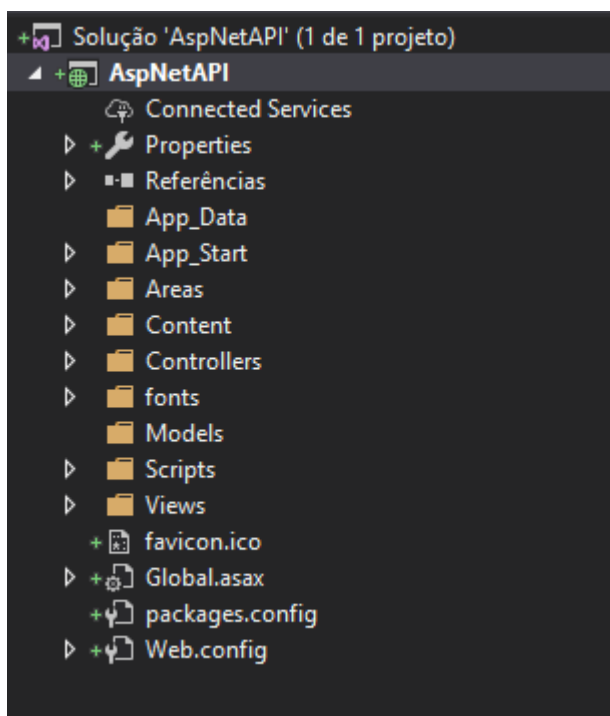
De o nome que preferir ao seu projeto, ao meu deixarei **AspNetAPI**.

Em framework selecione **.NET Framework 4.6 (funciona também em versões superiores do .NET Framework, caso queira fazer em alguma versão superior, sem problemas)**.

Em template selecione **API Web**.

Caso a opção **“Configurar para HTTPS”** estiver selecionada, desmarque-a não precisaremos dela.

E assim o nosso projeto será criado com a seguinte estrutura:



Vamos entender cada pasta e inclusive, apagar algumas que não iremos utilizar.

App_Data: é um ponto de armazenamento para dados em formato de arquivos (TXT, XML, JSON), como não iremos utilizar arquivos para armazenar dados, **pode deletar essa pasta sem problemas**.

App_Start: é onde temos nossos arquivos de configuração de inicialização e build de projeto, como é uma pasta essencial para o funcionamento da API, deixaremos ela do jeito que está. Abra esta pasta para podermos ver o padrão de rotas da **API**.

```
public static void Register(HttpConfiguration config)
{
    // Serviços e configuração da API da Web

    // Rotas da API da Web
    config.MapHttpAttributeRoutes();

    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
}
```

Esse método mostra qual será a URL padrão de acesso para a API, que no caso é “api/{controller}/{id}”

O /api significa que a requisição HTTP será diretamente na API.

O **controller** é onde nós recebemos os dados do usuário e chamamos uma classe aplicação para poder inserir no repositório (vou utilizar uma base de dados fictícia para facilitar o desenvolvimento)

O **id** é o método que vamos chamar para fazer o **request** para a API.

...

Areas: Não faço ideia do porque o template criar uma pasta de áreas em uma API, mas vamos lá, essa pasta fica as outras estruturas de projeto, quando eu quero ter por exemplo, 2 **controllers** rodando em projetos diferentes, se você abri-la, verá que tem outra estrutura de projeto, como nós não precisamos disso, **pode apagar a pasta sem problemas.**

Content: é onde ficam os arquivos CSS, imagens, vídeos e todos os arquivos de conteúdo, por padrão o projeto já vem com o **Bootstrap** instalado, nós não iremos utilizar nada disso, **pode apagar essa pasta sem problemas.**

Controllers: são nossas classes de controle, será lá que criaremos os métodos para receber os dados e repassar para classe de aplicação, ao abri-la verá que já vem

criado por padrão 2 controllers, **HomeController.cs** e **ValuesController.cs**, para melhor entendimento criaremos um controller limpo do zero, **então apague esses 2 anteriores.**

Fonts: nessa pasta vem todos os arquivos de fontes de texto, por padrão ele já vem com a fonte “**glyphicons**”, instalada, como também não iremos utilizar fonte para nada, **pode apagar essa pasta sem problemas.**

Scripts: é onde fica nossos arquivos Javascript, por padrão também já vem com o **Bootstrap** instalado, como também não utilizaremos JS para nada, **pode apagar essa pasta sem problemas.**

Views: é onde fica nossas telas, como nossa API não terá nenhuma tela além da Index (que será uma simples mensagem), **apague todos os arquivos dessa pasta menos Web.Config.**

...

Showwww, limpamos o nosso projeto, deixando-o o mais reduzido e simples possível, isso ajuda muito em questão de desempenho.

Como apagamos alguns arquivos, precisamos também apagar alguns métodos que uso desses arquivos.

Abra a pasta **App_Start** > abra o arquivo **BundleConfig.cs** apague todo o conteúdo do método **RegisterBundles**, deixando-o vazio:

```
public static void RegisterBundles(BundleCollection bundles)
{
    ...
}
```

Agora vamos criar nosso primeiro **controller** que é onde terá nossa simples tela de inicialização.

Clique com o botão direito na pasta **Controllers** > **Adicionar** > **Controlador** > **Controlador MVC 5 – Vazio.**

Clique em **adicionar** e nomeie para **ViewController.cs**

Vamos criar o método de inicialização (A mensagem que vai aparecer quando nós compilarmos), abra a classe **ViewController.cs** e digite o seguinte código:

```
public ActionResult Index()
{
    ViewBag.Mensagem = "medium.com/@lucas.eschechola";
    return View();
}
```

Será o método que a API irá chamar quando nós compilarmos, por padrão a API já vem configurada para rodar no **controller Home**, porém como deletamos ele, precisamos configurar para rodar no **controller View**.

Clique com botão direito na pasta **Views > Adicionar > Nova Pasta > Nomeie de View**.

Clique com o botão direito na pasta **View > Adicionar > Exibir > Nomeie para Index**.

Será criado dentro da pasta **View** um arquivo chamado **Index.cshtml**, o código desse arquivo é:

```
<h2>
    @ViewBag.Mensagem
</h2>
```

Pronto, nossa tela de inicialização já está feita (No .Net Core não precisamos criar isso pois você pode transformar um método com retorno **string** em uma **view**)

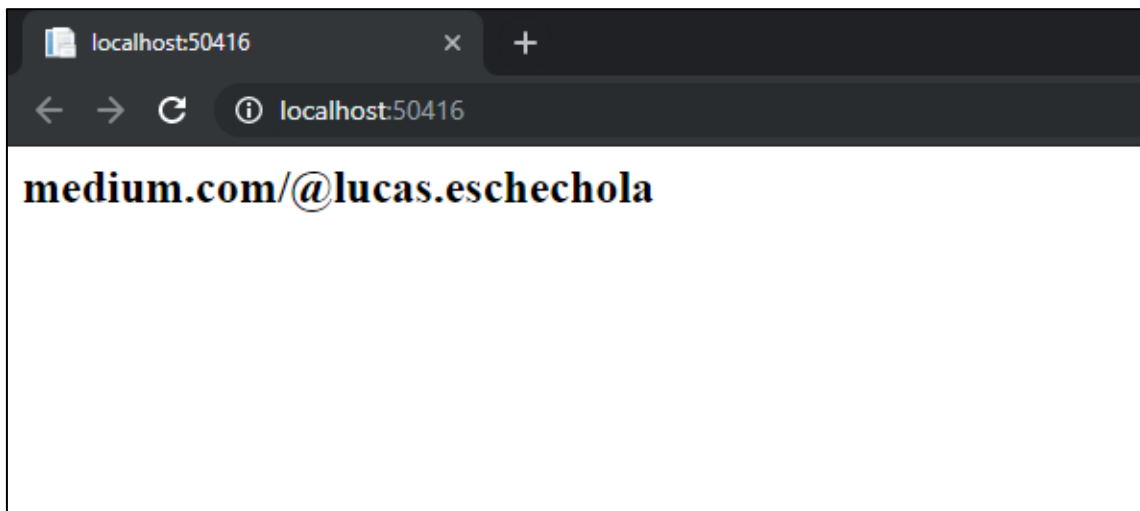
Agora só nos resta configurar a rota de inicialização, abra a pasta **App_Start > RouteConfig.cs** e altere a linha 19 onde está escrito:

Controller = "Home"

Altere para:

Controller = "View"

Aperte F5 ou clique em “IIS Express” para rodar o projeto e a tela que você irá ver será essa:



A mensagem é meramente ilustrativa, você pode colocar a que preferir 😊

Finalmente terminamos de criar e configurar nosso projeto, agora bora pra parte legal, programar!!!!!!

Criando o modelo e a camada de Aplicação

Para começarmos a criar nossa camada de aplicação, precisamos primeiramente criar um modelo de negócio, ou melhor, nosso modelo de dados, como já mencionado acima farei a API baseada em dados de uma pessoa (**Usuario**).

Então vamos lá, clique com o botão direito na pasta **Models > Adicionar > Classe > Nomeie para Usuarios.cs**

Agora precisamos adicionar as propriedades de um usuário, na classe **Usuarios.cs** digite o seguinte código:

```
public class Usuarios
{
    public int Id { get; set; }
    public string Nome { get; set; }
    public string Email { get; set; }
    public string Senha { get; set; }
}
```

Então cada usuário nosso terá esses campos, no qual o campo ID será gerado automaticamente.

Com o nosso modelo de dados já pronto, vamos então criar nossa camada ou melhor, classe de aplicação.

Clique com o **botão direito no projeto > Adicionar > Nova Pasta > Nomeie para Aplicacao**

Clique com o **botão direito dentro dessa pasta > Adicionar > Classe > Nomeie para UsuarioAplicacao.cs**

Será dentro dessa classe que nós iremos colocar nosso banco de dados fictício e todos os métodos que irão manipular esses dados.

O código dessa classe ficará da seguinte forma:

```
public class UsuarioAplicacao
{
    //criando o banco de dados fictio utilizando o modelo Usuario e adicionando 4 usuarios
    private List<Usuarios> usuarios = new List<Usuarios>
    {
        new Usuarios{ Id = 0, Nome = "Lucas Gabriel", Email = "lucas.gabriel@eu.com", Senha="12345"},
        new Usuarios{ Id = 1, Nome = "João Vitor", Email = "joao.vitor@eu.com", Senha="12345"},
        new Usuarios{ Id = 2, Nome = "Ana Maria", Email = "ana.maria@eu.com", Senha="12345"},
        new Usuarios{ Id = 3, Nome = "Isabella Silva", Email = "isabella.silva@eu.com", Senha="12345"},
    };

    //método para adicionar um usuário na lista
    public bool Adicionar(Usuarios usuarioRecebido)
    {
        //atribui os dados a uma nova variavel
        var usuarioInserir = usuarioRecebido;
        //pega o valor do ultimo id cadastrado precisa do -1 pois começa em zero
        var ultimoIdCadastrado = usuarios[usuarios.Count - 1].Id;
        //adiciona +1 pois o id nao pode ser igual ao anterior
        usuarioInserir.Id = ultimoIdCadastrado + 1;

        try
        {
            usuarios.Add(usuarioInserir);
            return true;
        }
        catch (Exception)
        {
            return false;
        }
    }

    //método para remover um usuário na lista através do Id utilizando LINQ
    public bool Remover(int Id)
    {
        try
        {
            usuarios.Remove(usuarios.Where(x => x.Id == Id).ToList()[0]);
            return true;
        }
        catch (Exception)
        {
            return false;
        }
    }
}
```

```

//método para alterar os dados de um usuario
public bool Alterar(Usuarios usuarioRecebido)
{
    //como o id é automatico, caso o id passado não exista
    //joga uma exceção que é tratada no controller
    try
    {
        usuarios[usuarioRecebido.Id].Nome = usuarioRecebido.Nome;
        usuarios[usuarioRecebido.Id].Senha = usuarioRecebido.Senha;
        usuarios[usuarioRecebido.Id].Email = usuarioRecebido.Email;
        return true;
    }
    catch (Exception)
    {
        return false;
    }
}

//método para exibir todos os usuarios
public List<Usuarios> ExibirTodos()
{
    return usuarios;
}

//método que busca um usuario através do id e retorna.
public Usuarios ExibirUsuario(int id)
{
    try
    {
        return usuarios.Where(x => x.Id == id).ToList()[0];
    }
    catch (Exception)
    {
        return null;
    }
}
}

```

Criamos uma Lista de usuários, que é o nosso repositório fictício já com alguns dados inseridos, e após isso criamos 5 métodos para acessarem esses dados, o famoso CRUD, caso ocorra algum erro ele vai retornar **false** ou **null** que será tratado no **controller**.

Criando nosso controller e nossas rotas (Primeira requisição REST!)

Agora nós vamos desenvolver a última parte da nossa API, nosso **controller**, onde conterá nossas rotas com suas funcionalidades, mas antes nós precisamos realizar uma última (prometo) configuração de rota.

Abra a pasta **App_Start WebApiConfig.cs**

Na linha **19** altere de:

routeTemplate: "api/{controller}/{id}",

para:

routeTemplate: "api/{controller}/{action}/{id}",

Show, com nossa rota já configurada, na pasta **Controllers > UsuarioController.cs**.

O código desse **controller** ficará o seguinte:

```
public class UsuarioController : ApiController
{
    //variavel estatica, para não resetar os dados quando nós fizermos as rotas
    static private readonly UsuarioAplicacao usuarioAplicacao = new UsuarioAplicacao();

    //rota para adicionar um novo usuário, recebe um usuario serializado em JSON
    [HttpPost]
    public IHttpActionResult Adicionar([FromBody]Usuarios usuario)
    {
        try
        {
            //chama a camada de aplicação para adicionar o usuário
            var sucesso = usuarioAplicacao.Adicionar(usuario);

            if (sucesso)
            {
                return Ok("Usuário inserido com sucesso.");
            }
            else
            {
                return BadRequest("Não conseguimos inserir o usuário. Por favor tente novamente");
            }
        }
        catch (Exception)
        {
            return BadRequest("Ocorreu algum erro, por favor tente novamente.");
        }
    }
}
```

```
//rota para alterar um usuário existente, recebe um usuario serializado em JSON
[HttpPut]
public IHttpActionResult Alterar([FromBody]Usuarios usuario)
{
    try
    {
        //chama a camada de aplicação para alterar o usuário
        var sucesso = usuarioAplicacao.Alterar(usuario);

        if (sucesso)
        {
            return Ok("Usuário atualizado com sucesso.");
        }
        else
        {
            return BadRequest("Não conseguimos atualizar o usuário. Por favor tente novamente");
        }
    }
    catch (Exception)
    {
        return BadRequest("Ocorreu algum erro, por favor tente novamente.");
    }
}
```

```
//rota para deletar um usuário existente, recebe o id do usuario que será deletado
[HttpDelete]
public IHttpActionResult Remover([FromBody]int idUsuario)
{
    try
    {
        //chama a camada de aplicação para deletar o usuário
        var sucesso = usuarioAplicacao.Remover(idUsuario);

        if (sucesso)
        {
            return Ok("Usuário removido com sucesso.");
        }
        else
        {
            return BadRequest("Não conseguimos remover o usuário. Por favor tente novamente");
        }
    }
    catch (Exception)
    {
        return BadRequest("Ocorreu algum erro, por favor tente novamente.");
    }
}
```

```
//rota para exibir um usuário existente, recebe o id do usuario que será retornado
[HttpPost]
public IHttpActionResult ExibirUsuario([FromBody]int idUsuario)
{
    try
    {
        //chama a camada de aplicação para retornar um usuário
        var usuarioRetornar = usuarioAplicacao.ExibirUsuario(idUsuario);
        if (usuarioRetornar != null)
        {
            //caso o usuário exista, ele transforma o usuário em um documento json e o retorna
            var usuarioSerializado = JsonConvert.SerializeObject(usuarioRetornar);
            return Ok(usuarioSerializado);
        }
        else
        {
            return BadRequest("Nenhum usuário foi encontrado com esse ID, por favor, tente novamente.");
        }
    }
    catch (Exception)
    {
        return BadRequest("Ocorreu algum erro, por favor tente novamente.");
    }
}

[HttpGet]
public IHttpActionResult ExibirTodos()
{
    try
    {
        //pega TODOS os usuário da camada aplicação
        var usuarios = usuarioAplicacao.ExibirTodos();

        if (usuarios != null)
        {
            //se ele conseguir pegar todos os usuário ele transforma essa lista de usuarios em JSON e retorna
            var usuariosSerializados = JsonConvert.SerializeObject(usuarios);
            return Ok(usuariosSerializados);
        }
        else
        {
            return BadRequest("Nenhum usuário cadastrado!");
        }
    }
    catch (Exception)
    {
        return BadRequest("Ocorreu algum erro, por favor tente novamente.");
    }
}
}
```


Vamos entender os tipos de retorno:

BadRequest()

Retornar um status 400, utilizado quando algo inesperado ou errado ocorre no processamento da rota.

Ok()

Retorna um status 200, utilizado quando tudo ocorre corretamente e conforme o esperado.

OBS: *Existem outros status de retorno http, aqui apenas abordei os 2 que utilizaremos, lembrando essa é uma API simples, para iniciantes, caso queira saber como fazer uma API de grande porte e escalável, você pode adquirir meu ebook gratuitamente no site:*

<https://eschechola-com.umbler.net/>

Com tudo certinho, lets bora testar essa bagaça.

Abra o **Advanced Rest Client** ou qualquer outro software para realizar requisições HTTP.

Vamos primeiramente testar a rota:

OBS: Lembre – se que a porta em localhost:XXXX pode alterar, onde XXXX é a porta, use a que abrir no navegador.

- Adicionar

/api/usuario/Adicionar





Em Request URL coloque:

Method	Request URL
POST	http://localhost:50416/api/usuario/Adicionar

No **Body**:

Headers	Body
Body content type application/json	Editor view Raw input
FORMAT JSON MINIFY JSON	
<pre>{ "Id": "0", "Nome": "Isabella Cristina", "Email": "isabella.cristina@eu.com", "Senha": "isabella123" }</pre>	

Clique em **Send** e a resposta será:

200 OK	252.73 ms
   	
<pre>"Usuário inserido com sucesso."</pre>	

OBS: Lembre-se que o Id é automático, então você não precisa saber qual o ID certo pra inserir, isso a API já faz automaticamente na camada de aplicação.

- Alterar

/api/usuario/Alterar

Vamos alterar o usuário de ID 1, então nessa rota eu preciso passar o **ID correto** de quem eu quero alterar.





Em Request URL coloque:

Method	Request URL
PUT	http://localhost:50416/api/usuario/Adicionar

No **Body**:

Headers	Body
Body content type application/json	Editor view Raw input
FORMAT JSON MINIFY JSON	
<pre>{ "Id": "1", "Nome": "Lucas Pedro", "Email": "lucas.pedro@eu.com", "Senha": "lucas123" }</pre>	

Clique em **Send** e a resposta será:

200 OK	7.85 ms
   	
<pre>"Usuário atualizado com sucesso."</pre>	

Foram alterados os dados do usuário com o ID 1, que foi o que nós informamos.

- Remover

/api/usuario/Remover

Vamos deletar o usuário de ID 2, então nessa rota eu preciso passar somente o ID do usuário que eu quero deletar.





Em Request URL coloque:

Method	Request URL
DELETE ▼	http://localhost:50416/api/usuario/Remover

No **Body**:

Headers	Body
Body content type application/json ▼	Editor view Raw input ▼
FORMAT JSON MINIFY JSON	
<pre>"2"</pre>	

Clique em **Send** e a resposta será:

200 OK	15.43 ms
   	
<pre>"Usuário removido com sucesso."</pre>	

Foi deletado o usuário que continha o ID 2, que foi o que nós informamos.

Agora vamos ver se nossas alterações anteriores foram realmente realizadas na base fictícia de dados:

- Exibir Todos

/api/usuario/ExibirTodos

Vamos mostrar todos os usuários, como o método utilizado é o GET, não precisamos passar nada no **body**.

Em Request URL coloque:

Method	Request URL
GET	http://localhost:50416/api/usuario/ExibirTodos

Clique em **Send** e a resposta será:

200 OK	25.13 ms
	
<pre>"[{\"Id\":0,\"Nome\":\"Lucas Gabriel\",\"Email\":\"lucas.gabriel@eu.com\",\"Senha\":\"12345\"},{\"Id\":1,\"Nome\":\"Lucas Pedro\",\"Email\":\"lucas.pedro@eu.com\",\"Senha\":\"lucas123\"},{\"Id\":3,\"Nome\":\"Isabella Silva\",\"Email\":\"isabella.silva@eu.com\",\"Senha\":\"12345\"},{\"Id\":4,\"Nome\":\"Isabella Cristina\",\"Email\":\"isabella.cristina@eu.com\",\"Senha\":\"isabella123\"}]\"</pre>	

Perceba que:

Foi retornado um documento JSON com os dados TODOS serializados.

O usuário de ID 1, foi alterado;

O usuário de ID 2, foi deletado;

E foi adicionado um usuário novo com o ID 4.

Então nossos métodos anteriores deram certo e alteraram nossos dados, por fim, vamos testar o método de retorno de um usuário.

- Exibir Um Usuario

/api/usuario/ExibirUsuario

Vamos retornar um usuário, para isso precisamos passar o ID do usuário que queremos.

Em Request URL coloque:

Method	Request URL
POST	http://localhost:50416/api/usuario/ExibirUsuario

No **Body**:

Headers	Body
Body content type application/json	Editor view Raw input
FORMAT JSON	MINIFY JSON
<pre>"4"</pre>	

Clique em **Send** e a resposta será:

200 OK	7.29 ms
<pre>{"Id":4,"Nome":"Isabella Cristina","Email":"isabella.cristina@eu.com","Senha":"isabella123"}</pre>	

Todos os dados do usuário com ID 4 em formato JSON.

Extras

Dependendo do navegador, ou plataforma que você use, a API pode acabar retornando os dados todos no formato **XML**.

Para forçar a requisição retornar todos os dados em JSON vá em: **“App_Start” > “WebApiConfig.cs”** e digite:

```
var    formatters    =    GlobalConfiguration.Configuration.Formatters;  
formatters.Remove(formatters.XmlFormatter);
```

Assim toda a requisição feita a API, será retornada em JSON.

Conclusão

O padrão REST é o padrão mais utilizado para comunicação entre aplicações do **MUNDO**, então utilize—o e pratique para poder fixar os conhecimentos.

No artigo eu manipulei dados de uma base de dados “fictícia”, mas, para utilizar uma base de dados real, bastaria apenas trocar os métodos da classe `UsuarioAplicacao` para fazer suas queries no banco ou utilizar uma ferramenta de ORM, o que torna o código da API reutilizável.

Agradeço a todos pela atenção e peço desculpas por qualquer erro de digitação e ou didática (Meu primeiro artigo!).