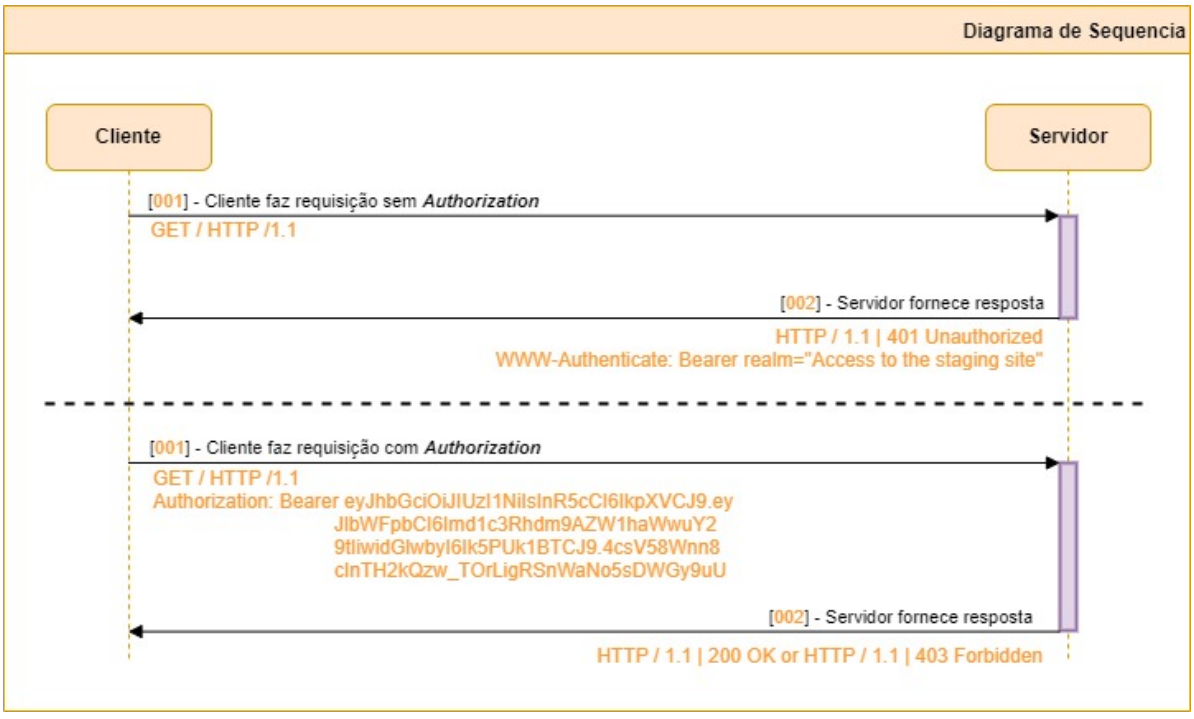
ASPNET - 5 - Blog Pessoal - Proteger API

1. Conceito de segurança em HTTP
2. JSON Web Token (JWT)
3. Protegendo API do Blog Pessoal - Parte 1
4. Protegendo API do Blog Pessoal - Parte 2
5. Protegendo API do Blog Pessoal - Parte 3
6. Protegendo API do Blog Pessoal - Parte 4
7. Protegendo API do Blog Pessoal - Parte 5
8. Testes de integração com Postman

1. Conceito de segurança em HTTP

No mundo da tecnologia, em especial da Internet, nenhuma aplicação em execução na Nuvem pode ficar sem algum tipo de Segurança habilitada, devido aos inúmeros perigos existentes no mundo virtual como ataques Hackers, invasões de Servidores, roubos de dados, entre outros. O *IETF* (*Internet Engineering Task Force*) tem como missão identificar e propor soluções para as questões/problemas relacionados à utilização da Internet, além de propor a padronização das tecnologias e protocolos envolvidos. O mesmo define a estrutura de autenticação *HTTP* que pode ser usada por um servidor para definir uma solicitação do cliente. O servidor responde ao cliente com uma mensagem do tipo *HTTP* Status 401(Não autorizado) e fornece informações de como autorizar com um cabeçalho de resposta *WWW-Authenticate* contendo ao menos uma solicitação. Um cliente que deseja autenticar-se com um servidor pode fazer isso incluindo um campo de cabeçalho de solicitação *WWW-Authenticate* com as credenciais. No Diagrama de Sequência abaixo pode se observar este relacionamento



O fluxo acima consiste em:

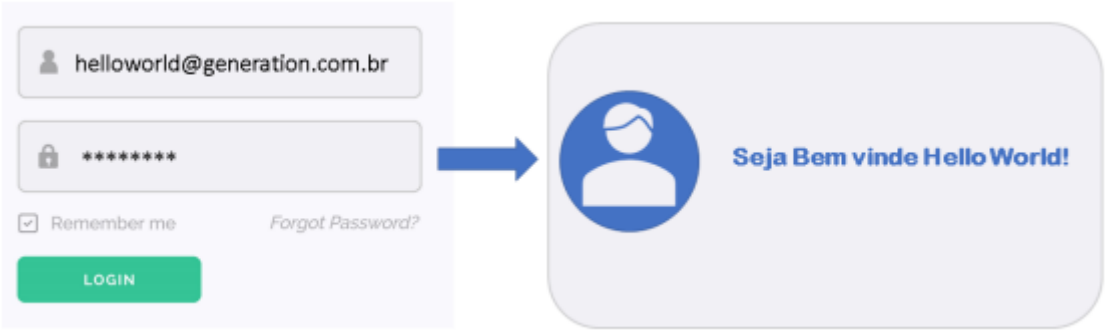
1. O servidor responde a um cliente com um status de resposta 401 (não autorizado) e fornece informações sobre como autorizar com um cabeçalho de resposta *WWW-Authenticate* contendo pelo menos um desafio;

2. Um cliente que deseja se autenticar com o servidor pode fazer incluindo um cabeçalho de solicitação de autorização com as credenciais;
3. Normalmente, um cliente apresentará um prompt de senha ao usuário e, em seguida, emitirá a solicitação, incluindo o cabeçalho correto.

Status Code utilizados

Status	Descrição
401 ou 407	Caso servidor receber credenciais invalidas;
403	Caso servidor receber credenciais válidas que não são adequadas para acessar determinado recurso;
404	Oculta a existência da página a um usuário sem privilégios adequados ou não autenticado corretamente;

Autenticação



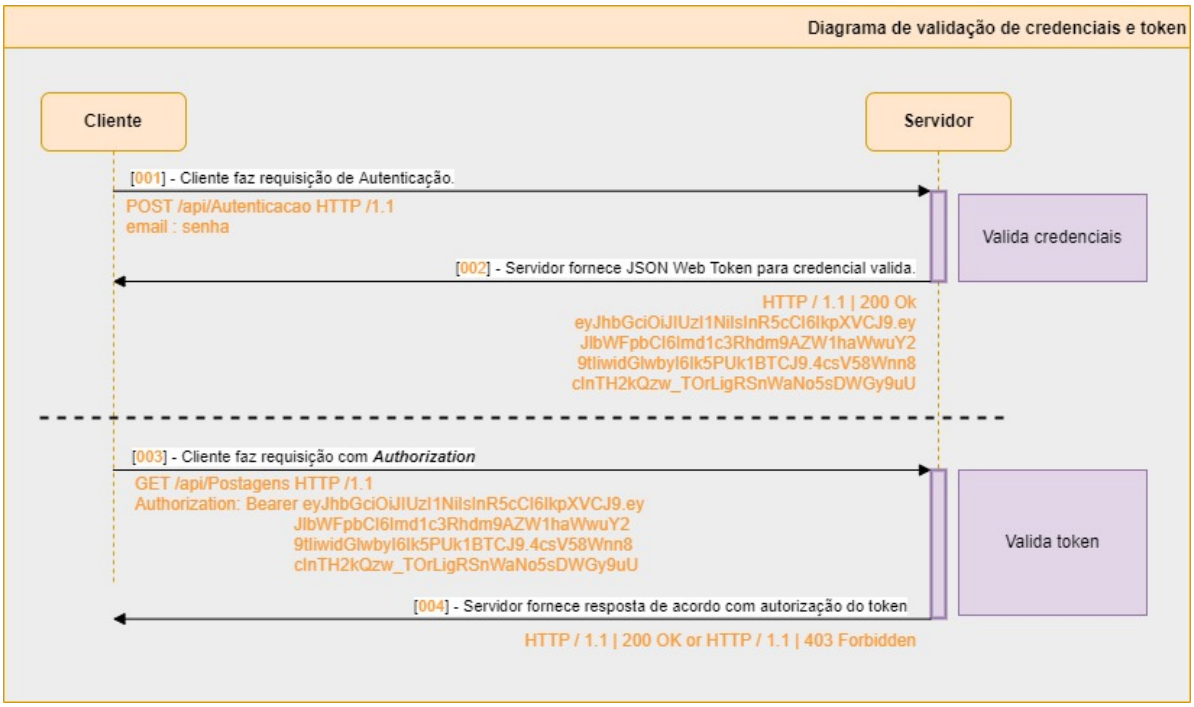
É o primeiro processo da Segurança da Informação, popularmente conhecido como Login no sistema. É o momento em que o usuário informa o seu usuário de acesso (e-mail) e a sua senha (criptografada), e o sistema fará a checagem se estas informações estão corretas.

Autorização

É o segundo processo da Segurança da Informação, popularmente conhecido como Direitos de acesso (Roles) no sistema. É o momento em que o sistema checará o que o usuário pode e não pode fazer no sistema, ou seja, as suas permissões dentro do sistema
(Quais recursos e endpoints podem ser acessados?).

2. JSON Web Token (JWT)

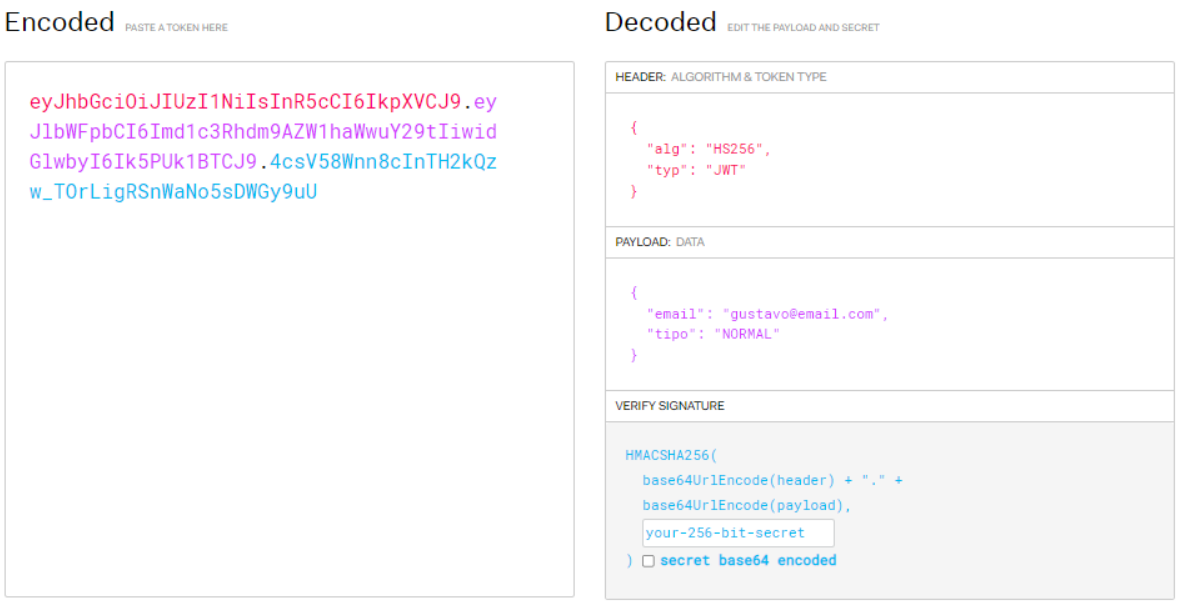
É uma estratégia de autenticação que sua única diferença é como o token é gerado. Veja o processo abaixo:



1. O Navegador (Cliente) faz uma requisição de autenticação para o servidor, passando suas credenciais no caso e-mail e senha;
2. O servidor valida se as credenciais estão corretas, normalmente validando se o usuário existe no banco de dados. Caso existir, gera um token e devolve para o cliente com status 200. Caso contrario um status 401;
3. Cliente faz uma requisição com um parâmetro de *Header Authorization* preenchido com o token fornecido pelo servidor;
4. Se o token for valido e o usuário possuir acesso ao ambiente solicitado e o servidor retorna 200 com o ambiente, caso o token seja valido mas o ambiente não for autorizado para o usuário uma boa pratica é devolver status 403.

Formato JWT

O token é formado por 3 regiões: *header*, *payload* e *signature* (cabeçalho, carga, assinatura). No esquema abaixo é representado cada uma das sessões:



Campo	Descrição
HEADER	Responsável por passar o algoritmo de codificação;
PAYLOAD	Responsável por passar reivindicações (*claims*) a serem validadas. podem ser publicas ou privadas;
SIGNATURE	Responsável por fornecer a assinatura, esta deve conter no servidor para poder validar os dados.

3. Protegendo API do Blog Pessoal - Parte 1

Validar os Pontos

Antes de começar a implementação da interface acima, é necessário validar alguns pontos importantes em algumas classes do sistema:

Criar TipoUsuario (`src/utilidades`):

Criar um arquivo chamado *Enum.cs* no diretório de utilidades (`src/utilidades`), caso não esteja criado, criar. Dentro do arquivo criar o `enum TipoUsuario`, que será responsável por informar se um usuário é `NORMAL` ou `ADMINISTRADOR`. No arquivo *Enum.cs*, escrever o código abaixo:

```
using System.Text.Json.Serialization;

namespace BlogPessoal.src.utilidades
{
    [JsonConverter(typeof(JsonStringEnumConverter))]
    public enum TipoUsuario
    {
        NORMAL,
        ADMINISTRADOR
    }
}
```

O `Enum`, é um tipo de dado que é possível customizar seu formato. Seu papel é enumerar, sendo o primeiro elemento de índice 1 e os demais consecutivamente n + 1. Também é possível visualizar uma notação `[JsonConverter(typeof(JsonStringEnumConverter))]` esta notação converte o valor `int` para seu formato `string` assim facilitando sua escrita literal.

Modificar UsuarioModelo (`src/modelos`):

Em usuário modelo fazer a modificação adicionando o atributo abaixo:

```
[Required]
public TipoUsuario Tipo { get; set; }
```

Seu código deve permanecer como o código abaixo:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;
using BlogPessoal.src.utilidades;

namespace BlogPessoal.src.modelos
{
    [Table("tb_usuarios")]
    public class UsuarioModelo
    {
        [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int Id { get; set; }

        [Required, StringLength(50)]
        public string Nome { get; set; }

        [Required, StringLength(30)]
        public string Email { get; set; }

        [Required, StringLength(30)]
        public string Senha { get; set; }

        public string Foto { get; set; }

        [Required]
        public TipoUsuario Tipo { get; set; }

        [JsonIgnore, InverseProperty("Criador")]
        public List<PostagemModelo> MinhasPostagens { get; set; }
    }
}
```

Observar que o importe de `TipoUsuario` foi efetuado no inicio do documento `using BlogPessoal.src.utilidades;`, e chamado como atributo de usuário. Com essa classe é um modelo do banco, será necessário apagar o banco de dados e rodar a aplicação novamente para se seja construído com o novo atributo.

Modificar classe NovoUsuarioDTO em *UsuarioDTO.cs* (`src/dtos`):

Na classe `NovoUsuarioDTO` inclui atributo de tipo de usuário. Seu papel será que ao criar um novo usuário seja passado o parâmetro para definir qual tipo de usuário esta sendo cadastrado. A classe `NovoUsuarioDTO` deve estar da seguinte maneira:

```
public class NovoUsuarioDTO
{
    [Required, StringLength(50)]
    public string Nome { get; set; }

    [Required, StringLength(30)]
    public string Email { get; set; }

    [Required, StringLength(30)]
    public string Senha { get; set; }

    public string Foto { get; set; }

    [Required]
    public TipoUsuario Tipo { get; set; }

    public NovoUsuarioDTO(string nome, string email, string senha, string foto,
TipoUsuario tipo)
    {
        Nome = nome;
        Email = email;
        Senha = senha;
        Foto = foto;
        Tipo = tipo;
    }
}
```

Assim que for efetuado esta alteração é possível que os testes apresente erro no código, isso se deve a falta do parâmetro `TipoUsuario` ser passado. Para correção basta incluir o campo `TipoUsuario.NORMAL` no final do construtor para que o código funcione.

Modificar método NovoUsuario em *UsuarioRepositorio.cs* (`src/repositorios/implementacoes`):

Incluir no método `NovoUsuario` a atribuição de `Tipo`, que será passado pela *dto*. Visualize o código abaixo:

```
public void NovoUsuario(NovoUsuarioDTO usuario)
{
    _contexto.Usuarios.Add(new UsuarioModelo
        {
            Email = usuario.Email,
            Nome = usuario.Nome,
            Senha = usuario.Senha,
            Foto = usuario.Foto,
            Tipo = usuario.Tipo
        });

    _contexto.SaveChanges();
```

```
}
```

Desta maneira será possível registrar no banco no momento da criação. Caso esse passo não for feito não será possível acessar a coluna no banco.

Criar AutenticacaoDTO.cs (`src/dtos`):

Esta `dto`, é utilizada para fazer autenticação e passar autorização para o usuário cliente. Criar um arquivo chamado *AutenticacaoDTO.cs* dentro de `src/dtos`, seu conteúdo esta definido abaixo:

```
using System.ComponentModel.DataAnnotations;
using BlogPessoal.src.utilidades;

namespace BlogPessoal.src.dtos
{
    public class AutenticarDTO
    {
        [Required]
        public string Email { get; set; }

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

        public AutenticarDTO(string email, string senha)
        {
            Email = email;
            Senha = senha;
        }
    }

    public class AutorizacaoDTO
    {
        public int Id { get; set; }
        public string Email { get; set; }
        public TipoUsuario Tipo { get; set; }
        public string Token { get; set; }

        public AutorizacaoDTO(int id, string email, TipoUsuario tipo, string
token)
        {
            Id = id;
            Email = email;
            Tipo = tipo;
            Token = token;
        }
    }
}
```

Classe	Definição
`AutenticarDTO`	Classe responsável por pegar dados para autenticação de usuário. No Blog Pessoal sera necessario validar o e-mail e a senha;
`AutorizacaoDTO`	Classe responsável por fornecer ao usuário dados para se manter no sistema. É nesta classe que passaremos o token.

Adicionar Secret (*appsettings.json*)

Para que nosso servidor tenha um código de segredo para autenticação, adcionar ao documento *appsettings.json*, o trecho de código abaixo:

```
"Settings":{
    "Secret":"fedaf7d8863b48e197b9287d492b708e"
},
```

O documento irá ficar da seguinte forma:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost; Initial Catalog=db_blogpessoal;
Trusted_Connection=True;"
  },
  "Settings":{
    "Secret":"fedaf7d8863b48e197b9287d492b708e"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Esta variável será acessada pelo sistema nas configurações do nosso token.

Incluir Pacotes

- Autenticação

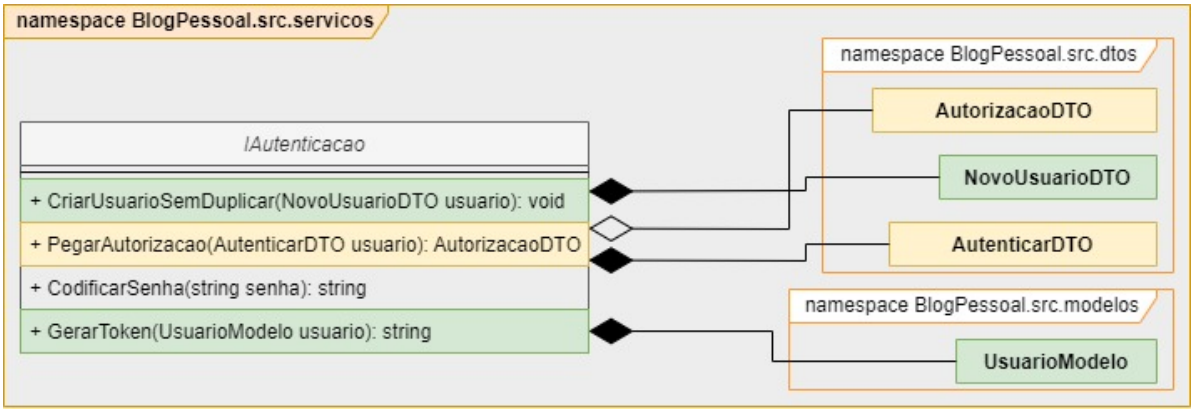
```
dotnet add package Microsoft.AspNetCore.Authentication -v 2.2.0
```

- JWTBearer

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer -v 5.0.13
```

4. Protegendo API do Blog Pessoal - Parte 2

Para proteção do Blog Pessoal criaremos um serviço para autenticar. O mesmo pode ser expressado no diagrama abaixo:



Método	Descrição
`CriarUsuarioSemDuplicar`	Este método deve ser implementado de forma que ao cadastrar um novo usuário, não seja permitido que o cadastro seja duplicado;
`PegarAutorizacao`	Este método tem o papel de devolver a uma *dto* preenchida que contenha dados para acesso do usuário e token JWT;
`CodificarSenha`	Este método é responsável de criptografar a senha do usuário, toda vez que o mesmo tem a intenção de salvar no banco;
`GeraarToken`	Este método é responsável por gerar o token no formato JWT.

Criando interface IAutenticacao

Agora que os ajustes foram colocados, será necessário criar o serviço para autenticação. No diretório `src/servicos` criar documento *IAutenticacao.cs*. Dentro do documento criar a interface seguindo o código abaixo:

```
using BlogPessoal.src.dtos;
using BlogPessoal.src.modelos;

namespace BlogPessoal.src.servicos
{
    public interface IAutenticacao
    {
        string CodificarSenha(string senha);
        void CriarUsuarioSemDuplicar(NovoUsuarioDTO usuario);
        string GerarToken(UsuarioModelo usuario);
        AutorizacaoDTO PegarAutorizacao(AutenticarDTO autenticacao);
    }
}
```

Uma interface existe para criar apenas a assinatura do método. A classe responsável por implementar é a que estende. Notar que a interface esta utilizando recursos que foram atualizados nos passos acima na sessão **validar pontos**.

Criando classe AutenticacaoServicos

Esta classe tem como objetivo implementar todos os métodos herdados de `IAutenticacao`. Para inicializar sua implementação criar um documento chamado *AutenticacaoServicos.cs* no diretório `src/servicos/implementacoes`. Dentro deste documento defina a estrutura abaixo:

```
using System;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using BlogPessoal.src.dtos;
using BlogPessoal.src.modelos;
using BlogPessoal.src.repositorios;
using Microsoft.Extensions.Configuration;
using Microsoft.IdentityModel.Tokens;

namespace BlogPessoal.src.servicos.implementacoes
{
    public class AutenticacaoServicos : IAutenticacao
    {
        #region Atributos

        private readonly IUsuario _repositorio;
        public IConfiguration Configuracao { get; }

        #endregion

        #region Construtores

        public AutenticacaoServicos(IUsuario repositorio, IConfiguration
configuration)
        {
            _repositorio = repositorio;
            Configuracao = configuration;
        }

        #endregion


        #region Mátodos

        #endregion
    }
}
```

Em C# é possível definir `#region` onde podemos delimitar aonde ira cada trecho de código digitado. Sempre quando temos uma `#region` aberta, devemos certificar de fecha-lá com `#endregion`. Na `#region Atributos` foi definido a variável `_repositorio` do tipo `IUsuario`, que será utilizada para acessar métodos do repositório caso necessário. Também foi definido a variável `Configuracao` do tipo `IConfiguration`, esta será usada para trazer nossa chave `Secret` que se encontra dentro do documento *appsettings.json*. Esse contexto apenas é possível ser utilizado quando inicializamos o mesmo através de um construtor, isso se deve a injeção de dependências do asp.net.

Método CodificarSenha (#region Métodos):

Este método tem a responsabilidade de codificar a senha do usuário antes de ser armazenada em nosso banco. Ele recebe como parâmetro uma `string` com a senha do usuário, logo recebe os bytes e depois converte para Base64. Veja a implementação abaixo:

```
public string CodificarSenha(string senha)
{
    var bytes = Encoding.UTF8.GetBytes(senha);
    return Convert.ToBase64String(bytes);
}
```

O método parece ser simples, mas sua implementação é fundamental para que a senha somente seja conhecida pelo usuário criador.

Método CriarUsuarioSemDuplicar (#region Métodos):

Este método é implementado de forma que ao cadastrar um novo usuário, não é permitido que o cadastro seja duplicado. O método recebe como parametro um `NovoUsuarioDTO` e não possui retorno, veja sua implementação abaixo:

```
public void CriarUsuarioSemDuplicar(NovoUsuarioDTO dto)
{
    var usuario = _repositorio.PegarUsuarioPeloEmail(dto.Email);

    if (usuario != null) throw new Exception("Este email já está sendo
utilizado");

    dto.Senha = CodificarSenha(dto.Senha);

    _repositorio.NovoUsuario(dto);
}
```

Uma característica interessante deste método é que o mesmo caso pesquisa um usuário no banco que possua o mesmo e-mail, caso encontre dispara uma `Exception`, não permitindo assim que a criação seja feita. Caso o e-mail não exista, a senha é codificada e o repositório de usuário é acionado com o método `NovoUsuario` passando como parâmetro a `dto`.

Método GerarToken (#region Métodos):

O método abaixo parece ser complicado em implementação, mas o mesmo ja foi mencionado em definição. Para compreender validar a tabela abaixo:

Campo	Descrição
HEADER	Responsável por passar o algoritmo de codificação;
PAYLOAD	Responsável por passar reivindicações (*claims*) a serem validadas. podem ser publicas ou privadas;
SIGNATURE	Responsável por fornecer a assinatura, esta deve conter no servidor para poder validar os dados.

A estrutura acima é a que define um token JWT. A estrutura abaixo da características para os campos acima e ao implementar alguns campos é possível notar que o trecho de código que implementa o campo é dado pela tabela abaixo:

Campo	Trecho de código
HEADER	`SigningCredentials`
PAYLOAD	`ClaimsIdentity`
SIGNATURE	`SigningCredentials`

O método recebe como parâmetro um *UsuarioModelo* (de acordo com sua existência no banco de dados) e retorna um token JWT em seu formato `string`, que será passado nas requisições. A implementação do código pode ser vista abaixo:

```
public string GerarToken(UsuarioModelo usuario)
{
    var tokenManipulador = new JwtSecurityTokenHandler();
    var chave = Encoding.ASCII.GetBytes(Configuracao["Settings:Secret"]);
    var tokenDescricao = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(
            new Claim[]
            {
                new Claim(ClaimTypes.Email, usuario.Email.ToString()),
                new Claim(ClaimTypes.Role, usuario.Tipo.ToString())
            }),
        Expires = DateTime.UtcNow.AddHours(2),
        SigningCredentials = new SigningCredentials(
            new SymmetricSecurityKey(chave),
            SecurityAlgorithms.HmacSha256Signature
        )
    };
    var token = tokenManipulador.CreateToken(tokenDescricao);
    return tokenManipulador.WriteToken(token);
}
```

Definições:

var	definição
`tokenManipulador`	Define o tipo de manipulador que será utilizado para construir o token, no caso utilizaremos o JWT;
`chave`	Corresponde a chave segredo do nosso servidor, essa chave passada como parâmetro para construir a assinatura do token;
`tokenDescricao`	Descreve os campos do token: algoritmo de codificação, reivindicações (*claims*) e assinatura;
`token`	Token criado.

Método PegarAutorizacao (#region Métodos):

O método acima e o que define como gerar o token, mas quem de fato ira utilizar será o método `PegarAutorizacao`. Este método recebe uma `AutenticarDTO` para autenticação e retorna uma `AutorizacaoDTO` de autorização. Dentro dessa dto de autorização é passado algumas informações que serão necessárias para manter o usuário acessando o sistema. Sua implementação esta descrita abaixo:

```
public AutorizacaoDTO PegarAutorizacao(AutenticarDTO autenticacao)
{
    var usuario = _repositorio.PegarUsuarioPeloEmail(autenticacao.Email);

    if (usuario == null) throw new Exception("Usuário não encontrado");

    if (usuario.Senha != CodificarSenha(autenticacao.Senha)) throw new
Exception("Senha incorreta");

    return new AutorizacaoDTO(usuario.Id, usuario.Email, usuario.Tipo,
GerarToken(usuario));
}
```

Uma característica bastante interessante neste método é que o mesmo valida se um usuário não for encontrado ou a senha digitada não corresponder ao mesmo. Caso encontre essas falhas dispara uma `Exception` com uma mensagem. Este método é muito importante pois através dele será garantido o acesso externo aos recursos de nossa aplicação.

5. Protegendo API do Blog Pessoal - Parte 3

Após implementar os *serviços* é necessário criar o escopo para futuras injeções de dependência e utilização do sistema. Também é necessário adicionar o serviço de autenticação as configurações. Para essa ação é necessário acessar o documento *Startup.cs* e adicionar o trecho de código abaixo:

```
// Configuração de Serviços
services.AddScoped<IAutenticacao, AutenticacaoServicos>();

// Configuração do Token Autenticação JWTBearer
var chave = Encoding.ASCII.GetBytes(Configuration["Settings:Secret"]);
services.AddAuthentication(a =>
    {
        a.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
        a.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    }).AddJwtBearer(b =>
    {
        b.RequireHttpsMetadata = false;
        b.SaveToken = true;
        b.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(chave),
            ValidateIssuer = false,
            ValidateAudience = false
        };
    }
);
```

O código acima é responsável por injetar o escopo de serviços contendo a interface `IAutenticacao` e vinculando a classe `AutenticacaoServicos`. Também esta sendo feita a configuração do token de autenticação *JWTBearer*. Nesta sessão é possível adicionar o serviço de autenticação que incluem a configuração que passa ao sistema o esquema de autenticação e o que será feito com o token.

Seu código ira ficar assim no método `ConfigureServices` :

```
public void ConfigureServices(IServiceCollection services)
{
    // Configuração Banco de Dados
    services.AddDbContext<BlogPessoalContexto>(
        opt =>
opt.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"])
    );

    // Configuração Repositorios
    services.AddScoped<IUsuario, UsuarioRepositorio>();
    services.AddScoped<ITema, TemaRepositorio>();
    services.AddScoped<IPostagem, PostagemRepositorio>();

    // Configuração de Controladores
    services.AddCors();
    services.AddControllers();

    // Configuração de Serviços
    services.AddScoped<IAutenticacao, AutenticacaoServicos>();

    // Configuração do Token Autenticação JWTBearer
    var chave = Encoding.ASCII.GetBytes(Configuration["Settings:Secret"]);
    services.AddAuthentication(a =>
        {
            a.DefaultAuthenticateScheme =
JwtBearerDefaults.AuthenticationScheme;
            a.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
        }).AddJwtBearer(b =>
        {
            b.RequireHttpsMetadata = false;
            b.SaveToken = true;
            b.TokenValidationParameters = new TokenValidationParameters
            {
                ValidateIssuerSigningKey = true,
                IssuerSigningKey = new SymmetricSecurityKey(chave),
                ValidateIssuer = false,
                ValidateAudience = false
            };
        }
    );
}
```

Também é necessário modificar o método `Configure` localizado no documento *Startup.cs*, adcionar o trecho de código abaixo:

```
// Autenticação e Autorização
app.UseAuthentication();
app.UseAuthorization();
```

Este trecho de código é necessário que se localize entre o código `app.UseRouting` e `app.UseEndpoints`. Esta configuração informa que sua aplicação agora utilizara autenticação e autorização. Seu código ira ficar assim no método `Configure` :

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
BlogPessoalContexto contexto)
{
    // Ambiente de Desenvolvimento
    if (env.IsDevelopment())
    {
        contexto.Database.EnsureCreated();
        app.UseDeveloperExceptionPage();
    }

    // Ambiente de produção

    // Rotas
    app.UseRouting();

    app.UseCors(c => c
        .AllowAnyOrigin()
        .AllowAnyMethod()
        .AllowAnyHeader()
    );

    // Autenticação e Autorização
    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Desta maneira sua aplicação estará pronta para ser testada!.

6. Protegendo API do Blog Pessoal - Parte 4

Validar os Pontos

Modificar método NovoUsuario e AtualizarUsuario em *UsuarioControlador.cs* (`src/controladores`):

Para registrar um novo usuário no sistema é necessário que seja alterado o método `NovoUsuario` localizado em `src/controladores` dentro do documento *UsuarioControlador.cs*. O método atual permite o cadastro duplicado, pois esta utilizando através do repositório. Para que funcione corretamente injetar o serviços de `IAutenticacao` na classe `UsuarioControlador`. O trecho de código abaixo já esta definido como ira ficar as regiões:

```
#region Atributos

private readonly IUsuario _repositorio;
private readonly IAutenticacao _servicos;
```

```
#endregion


#region Construtores

public UsuarioControlador(IUsuario repositorio, IAutenticacao servicos)
{
    _repositorio = repositorio;
    _servicos = servicos;
}

#endregion
```

Desta maneira é possível acessar não somente o repositório de usuário, como também é possível acessar os serviços de autenticação. Agora é importante efetuar a alteração do método `NovoUsuario`. O mesmo ira ficar definido da seguinte maneira:

```
[HttpPost]
[AllowAnonymous]
public IActionResult NovoUsuario([FromBody] NovoUsuarioDTO usuario)
{
    if(!ModelState.IsValid) return BadRequest();

    try
    {
        _servicos.CriarUsuarioSemDuplicar(usuario);
        return Created($"api/Usuarios/email/{usuario.Email}", usuario);
    }
    catch (Exception ex)
    {
        return Unauthorized(ex.Message);
    }
}
```

Nosso novo método agora não somente cria um usuário, se não que valida sua existência no banco, caso não exista o método codifica a senha e salva o novo usuário no banco. Caso usuário já exista no banco uma exceção é lançada como reposta e nosso método informa com um status 401. Vale notar que o método esta notado um `[AllowAnonymous]`, essa notação permite que mesmo protegido esse endpoint esta liberado para acesso de qualquer usuario.

Para atualizar um usuário é necessário validar a codificação da senha. Para isso incluir deixar o código da maneira que esta definida abaixo:

```
[HttpPut]
[Authorize(Roles = "NORMAL,ADMINISTRADOR")]
public IActionResult AtualizarUsuario([FromBody] AtualizarUsuarioDTO usuario)
{
    if(!ModelState.IsValid) return BadRequest();

    usuario.Senha = _servicos.CodificarSenha(usuario.Senha);

    _repositorio.AtualizarUsuario(usuario);
    return Ok(usuario);
}
```

Vale destacar que antes de chamar o método de atualização do repositório é feito a codificação da senha no método `_servicos.CodificarSenha(usuario.Senha);`, isso garante que nossa aplicação não falhe depois que um usuário for alterado. Ressaltando que para este método tanto usuário NORMAL quanto ADMINISTRADOR estão autorizados para utilização;

Criar controlador AutenticacaoControlador (`src/controladores`):

Este controlador possuirá apenas 1 método, seu nome será `Autenticar`, seu papel é fornecer a autorização através da autenticação passada como parâmetro. Sua implementação pode ser vista no trecho abaixo de código:

```
using System;
using BlogPessoal.src.dtos;
using BlogPessoal.src.servicos;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BlogPessoal.src.controladores
{
    [ApiController]
    [Route("api/Autenticacao")]
    [Produces("application/json")]
    public class AutenticacaoControlador : ControllerBase
    {
        #region Atributos

        private readonly IAutenticacao _servicos;

        #endregion


        #region Construtores

        public AutenticacaoControlador(IAutenticacao servicos)
        {
            _servicos = servicos;
        }

        #endregion


        #region Métodos

        [HttpPost]
        [AllowAnonymous]
        public IActionResult Autenticar([FromBody] AutenticarDTO autenticacao)
        {
            if(!ModelState.IsValid) return BadRequest();

            try
            {
                var autorizacao = _servicos.PegarAutorizacao(autenticacao);
                return Ok(autorizacao);
            }
            catch (Exception ex)
            {
                 return Unauthorized(ex.Message);
```

```
            }
        }

        #endregion
    }
}
```

O método definido como `Autenticar` utiliza nosso serviço `PegarAutorizacao` para retornar as credenciais validas para acessar os endpoints. Caso na autenticação um e-mail ou senha forem inválidos, nosso método retornara um status 401 com mensagem de exceção.

7. Protegendo API do Blog Pessoal - Parte 5

Para finalizar a implementação é necessário fazer com que todos os *endpoints* estejam datados com notações de autorização. Veja abaixo exemplos:

Notação	Quando utilizar
`[AllowAnonymous]`	Quando endpoint for acessado sem precisar de autorização;
`[Authorize(Roles = "NORMAL,ADMINISTRADOR")]`	Quando usuários do tipo `NORMAL` e `ADMINISTRADOR`, puderem acessar;
`[Authorize(Roles = "ADMINISTRADOR")]`	Quando somente usuários do tipo `ADMINISTRADOR` puderem acessar;
`[Authorize]`	Quando qualquer tipo de usuário credenciado puder acessar.

Proteger os métodos do Blog Pessoal da seguinte maneira:

Controlador	Método	Notação
`AutenticacaoControlador`	`Autenticar`	`[AllowAnonymous]`
`UsuarioControlador`	`PegarUsuarioPeloId`	`[Authorize(Roles = "NORMAL,ADMINISTRADOR")]`
`UsuarioControlador`	`PegarUsuariosPeloNome`	`[Authorize(Roles = "NORMAL,ADMINISTRADOR")]`
`UsuarioControlador`	`PegarUsuarioPeloEmail`	`[Authorize(Roles = "NORMAL,ADMINISTRADOR")]`
`UsuarioControlador`	`NovoUsuario`	`[AllowAnonymous]`
`UsuarioControlador`	`AtualizarUsuario`	`[Authorize(Roles = "NORMAL,ADMINISTRADOR")]`
`UsuarioControlador`	`DeletarUsuario`	`[Authorize(Roles = "ADMINISTRADOR")]`
`TemaControlador`	`PegarTodosTemas`	`[Authorize]`
`TemaControlador`	`PegarTemaPeloId`	`[Authorize]`
`TemaControlador`	`PegarTemasPelaDescricao`	`[Authorize]`
`TemaControlador`	`NovoTema`	`[Authorize]`
`TemaControlador`	`AtualizarTema`	`[Authorize(Roles = "ADMINISTRADOR")]`
`TemaControlador`	`DeletarTema`	`[Authorize(Roles = "ADMINISTRADOR")]`
`PostagemControlador`	`PegarPostagemPeloId`	`[Authorize]`
`PostagemControlador`	`PegarTodasPostagens`	`[Authorize]`
`PostagemControlador`	`PegarPostagensPorPesquisa`	`[Authorize]`
`PostagemControlador`	`NovaPostagem`	`[Authorize]`
`PostagemControlador`	`AtualizarPostagem`	`[Authorize]`
`PostagemControlador`	`DeletarPostagem`	`[Authorize]`

8. Testes de integração com Postman

Visão geral de Testes

Testes Funcionais

São testes que avaliam o comportamento da aplicação (funcionalidades, regras de negócio, etc) e são eles:

1. Unitário:
    - Também conhecidos como testes de unidade. Tem por objetivo testar a menor parte testável do sistema, geralmente um método;

    - Não utilizam banco, mas podem possuir *mocks*. *Mocks* são objetos "falsos" que simulam o comportamento de uma classe ou objeto real.
2. Integração:
    - Tem por objetivo testar a interação entre componentes de software, normalmente duas partes do software. É o processo de verificar se os componentes do sistema, juntos, trabalham conforme esperado;
    - Podem utilizar banco e *mocks*;
    - Costumam testar a integração entre classes, chamadas a *WebServices* (normalmente com *mocks*), entre outras coisas.
3. Sistema:
    - Tem como objetivo garantir que o sistema funciona como um todo. Se no teste de integração, o acesso ao banco, aos serviços web, etc, são testados individualmente, nos testes de sistema o sistema é testado como um todo;
    - São comumente chamados de testes de caixa-preta, pois o sistema é testado com tudo ligado: banco de dados, serviços web, batch jobs, e etc;
    - Usando como exemplo uma aplicação Web, os testes de sistema seriam responsáveis por fazer os passos que um usuário poderia fazer, então um teste de sistema poderia abrir um browser, se logar na aplicação e conferir se tudo foi executado com sucesso, da mesma forma que pode conferir como a tela da aplicação reage caso o usuário digite um dado inválido.
4. Aceitação:
    - Tem como objetivo verificar se o que foi implementado atende corretamente ao que o cliente esperava, ou seja, validar o sistema do ponto de vista do cliente. Normalmente, isso é feito através de testes de sistema.

Testes não Funcionais

São testes que verificam atributos de um componente de sistema que não se relacionam com a funcionalidade (confiabilidade, eficiência, usabilidade, manutenabilidade e portabilidade). Se dividem em muitos tipos, mas aqui vão os principais ou mais utilizados:

1. Carga:
    - Tem como objetivo testar o sistema simulando múltiplos usuários acessando ao mesmo tempo para ver como o sistema se comporta e aguenta. Está diretamente associado aos próximos dois testes.
2. Performance:
    - Tem como objetivo testar rapidez: tempo de resposta. Pode estar sobre condições que seriam próximas às reais para verificar a performance do sistema.
3. Estresse:
    - Tem como objetivo testar os limites da aplicação. A ideia é sobrecarregar o sistema com muitos usuários para ver até onde o sistema para e como (e se) ele se recupera.

Ferramentas

- No C#, as ferramentas para testes funcionais costumam ser: **MSTest**, **xUnit** e **NUnit**, utilizado com o **Selenium** (para testes de sistema) e o uso de **Specflow** para de comportamento(BDD);
- Para testes não-funcionais, independente de linguagem, o mais conhecido é o **JMeter**, mas possuem outros como **httperf** (para testes de performance) e **locust** (para todos os tipos).

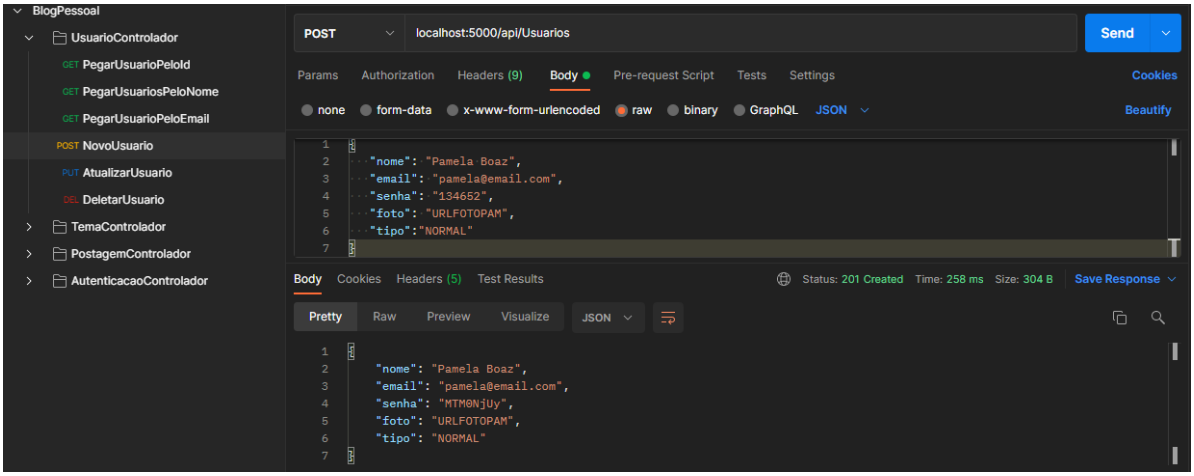
Modelo Ideal de téste

- Os testes unitários testando praticamente todos os métodos da aplicação;
- Os testes de integração testando apenas as partes das pequenas integrações;
- Os testes de sistema (e aceitação) testando o sistema no geral.

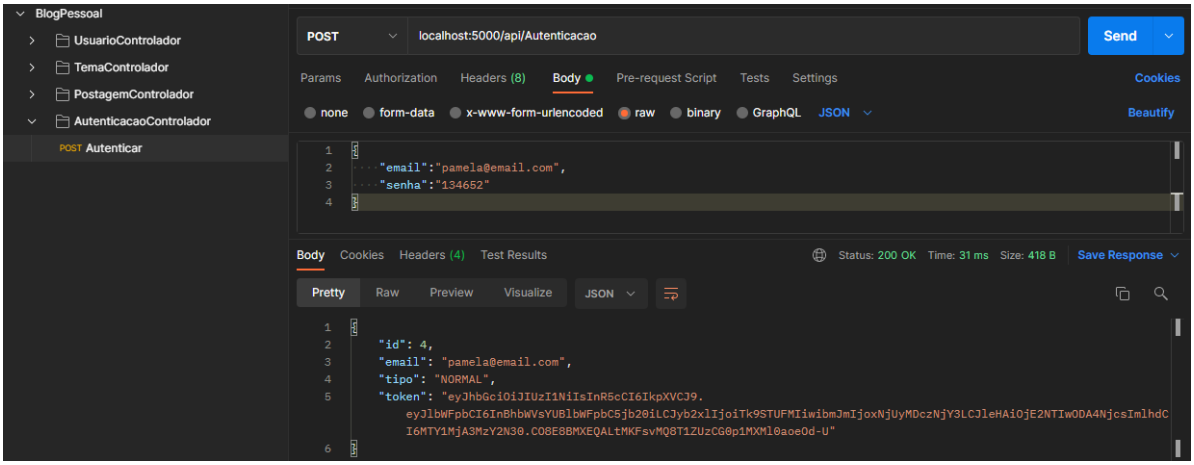
Testando autorização e autenticação com Postman

Para testar com postman, fazer download da ferramenta no Link. Sempre que quiser testar a aplicação é necessário executar o projeto.

Criar um usuário:



Autenticar usuário:



Teste o acesso com token: