Um termo recorrente nas APIs é a autenticação e autorização, os Roles, Claims, Policies, JWT, oAuth e tudo mais. Embora pareça um assunto complexo, não é, e vamos desmistificar tudo neste guia!

Autenticação

A autenticação diz quem você é. Ela é responsável por identificar o usuário ou quem estiver acessando a API.

É normal hoje em dia vermos autenticação com terceiros, como Facebook, Google e agora até a Apple.

Este mecanismo, interno ou externo, garante que quem está acessando sua API é realmente quem ele diz ser.

Autorização

A autorização por sua vez, diz o que o usuário pode fazer. Ao contrário da autenticação, dificilmente você poderá terceirizar sua autorização, visto que ela faz parte do seu negócio e muitas vezes é específica para uma aplicação.

Roles, Claims e Policies

A autorização se dá por meio de Roles, Claims e Policies, que restringem ou permitem o acesso do usuário a determinada parte da aplicação.

Um Role (Perfil) é um papel, uma função, um cargo, algo que define acesso a um ou um grupo de usuários em partes da aplicação.

Normalmente trabalhamos com Roles em cenários mais simples, onde não necessitamos de muita interação, onde por exemplo, apenas sabendo que o usuário possui o papel de Gerente, já podemos tomar alguma decisão.

Um Claim (Afirmação) é algo mais imperativo em que na maioria dos casos utilizamos pra realizar alguma interação antes de deixar o usuário prosseguir.

Por exemplo, se precisamos validar se o usuário pertence a uma empresa específica, se ele realizou determinada venda ou qualquer outra validação que dependa de banco de dados por exemplo, este é um cenário interessante para os Claims.

As Policies (Políticas) nada mais são que as aplicações de Roles e Claims para determinar contratos de acessos a determinadas áreas de uma aplicação.

Você pode ter vários Roles e Claims compondo uma política que determina acesso a uma parte do seu sistema.

oAuth

Open Authentication nada mais é do que um padrão que especifica como fazemos as autenticações. Com a chegada de APIs abertas há um tempo atrás, foram necessárias algumas especificações mínimas de segurança e o oAuth foi criado.

Hoje ele se consolidou no mercado sendo utilizado por gigantes como Facebook, Twitter, Instagram, Spotify, Netflix dentre outros.

Token

Um token é uma chave encriptada, contendo informações do usuário, que só pode ser desencriptada com outra chave (privada) que fica no servidor.

Pode parecer estranho dizer que temos informações do usuário (Não sensíveis) em uma chave, mas ela garante que não precisamos armazenar estes tokens, tornando possível o que chamamos de Stateless Applications, ou aplicações sem estado.

Isto é fundamental para o escalonamento de qualquer aplicação. Afinal, sempre que uma máquina nova da sua aplicação sobe, sua memória não é copiada.

JWT

É a sigla para Json Web Token e pronuncia-se JOT. O JWT nada mais é que o armazenamento das informações do token no formato JSON.

Antes de prosseguir...

O processo de autenticação inicia-se em uma aplicação (Normalmente frontend) fazendo uma requisição para sua API.

Nesta requisição serão enviados o usuário e senha. Neste momento, nem preciso dizer que suas aplicações PRECISAM trafegar sobre HTTPS, isto é uma exigência do Google para indexar seu site.

Com o usuário e senha, é feito o processo normal de verificação do login, e caso tudo ocorra bem, sua API utiliza uma chave PRIVADA, que só ela tem para gerar um TOKEN contendo as informações que você desejar, no formato JWT.

Neste momento, você tem uma chave enorme, com todas as informações que colocou previamente (Nunca armazene informações sensíveis nos tokens), encriptada com uma chave privada que só a API e no formato JSON.

Abaixo um exemplo de chave:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bmlxdWVfbmFtZSI6ImFuZHJlYmFsdGllcmkiLCJTdG9yZSI6InVzZXIiLCJuYmYiOjE1NjE2MDI2NjYsImV4cCI6MTU2MjIwNzQ2NiwiaWF0IjoxNTYxNjAyNjY2fQ.y2cvZ02PLJG-BFPUAWn1fChTvS4rPOv\\_iEEsLPd7eBg

Essa chave decodificada tem o seguinte conteúdo

{'alg':'HS256','typ':'JWT'}{'unique\\_name':'andrebaltieri','Store':'user','nbf':1561602666,'exp':1562207466,'iat':1561602666}

No caso, além do nome de usuário e perfil, também informo a data/hora de criação e expiração da mesma, mas isto é uma particularidade de código.

Conforme comentei anteriormente, nós não ficamos “logados” na API, então após o Login, este token é retornado e deve ser enviado novamente a cada requisição.

Configurando sua API

A primeira coisa que precisamos fazer em nossa API é definir uma chave, um segredo, algo que só a API terá acesso.

Esta chave será utilizada para encriptar o token e afins posteriormente. Não vou focar em COMO, mas sim em O QUE você precisa fazer, e neste cenário, eu criei uma classe estática, chamada Settings.cs com minha chave.

Poderia utilizar o AppSettings dentre várias outras coisas, mas o cenário mais simples é este, e como nossa chave não tende a mudar, isto resolve o problema.

***Importante:****Se você mudar a chave, todos os tokens ainda ativos perderão a validade, já que será impossível desencriptá-los com uma chave diferente da original.*

namespace store

{

public static class Settings

{

public static string Secret = "43e4dbf0-52ed-4203-895d-42b586496bd4";

}

}

Com nossa chave definida, precisamos gerar nossa chave, e este processo pode ser feito de diferentes maneiras também, com RSA e afins, mas um dos mais simples e que funciona é este abaixo:

var key = Encoding.ASCII.GetBytes(Settings.Secret);

Adicionando Autenticação

O primeiro passo é adicionar autenticação a nossa aplicação, e para isto vamos utilizar o método **AddAuthentication** do ASP.NET Core, que traz uma implementação padrão ótima para o que precisamos.

services.AddAuthentication(x =>

{

x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

Note que definimos o **AuthenticationScheme** como JWT, o qual desejamos utilizar aqui. Além disso, há um **DefaultChallenge** que definimos também, mas ele não nos interessa no momento.

Depois de definido nosso esquema, precisamos dizer como nossos tokens vão funcionar, quais configurações irão ter, e isto é feito no método **AddJwtBearer**, conforme abaixo.

.AddJwtBearer(x =>

{

x.RequireHttpsMetadata = false;

x.SaveToken = true;

x.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuerSigningKey = true,

IssuerSigningKey = new SymmetricSecurityKey(key),

ValidateIssuer = false,

ValidateAudience = false

};

});

O que mais importa aqui são os **TokenValidationParameters**, que neste caso, informamos nossa chave, chamando a função **SymmetricSecurityKey** do .NET para gerar a chave segura.

Precisamos também, informar ao ASP.NET Core para utilizar a autenticação, e isto é feito no método Configure.

app.UseAuthentication();

Por fim, é importante sempre utilizar o **SetCompatibilityVersion** para garantir que nada quebrará em versões futuras. O resultado final da classe Startup.cs é este abaixo, mas calma que ele ainda irá mudar.

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddCors();

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

var key = Encoding.ASCII.GetBytes(Settings.Secret);

services.AddAuthentication(x =>

{

x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(x =>

{

x.RequireHttpsMetadata = false;

x.SaveToken = true;

x.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuerSigningKey = true,

IssuerSigningKey = new SymmetricSecurityKey(key),

ValidateIssuer = false,

ValidateAudience = false

};

});

}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

app.UseCors(x => x

.AllowAnyOrigin()

.AllowAnyMethod()

.AllowAnyHeader());

app.UseAuthentication();

app.UseMvc();

}

}

Adicionando Autorização

Se utilizamos o **AddAuthentication** para autenticação, o **AddAuthorization** vai para autorização. Intuitivo né.

Neste cenário teremos dois Claims, um para usuário e outro para administrador, e estes são configurados neste formato, também no método **ConfigureServices** da classe Startup.cs.

services.AddAuthorization(options =>

{

options.AddPolicy("user", policy => policy.RequireClaim("Store", "user"));

options.AddPolicy("admin", policy => policy.RequireClaim("Store", "admin"));

});

No código acima eu crio duas Policies e dois Claims, uma para cada cenário. Em adicional, sempre definimos uma “chave” para eles, no caso utilizei o nome da aplicação, “Store”.

Deste modo definimos as políticas e afirmações, mas ainda não estamos utilizando elas, veremos isto nos Controllers logo mais.

Políticas Globais

Particularmente, em diversos cenários acho mais fácil restringir TODA a API e liberar só o que preciso, conforme a demanda.

Desta forma, podemos definir políticas globais, dizendo que TODO endpoint da nossa API requer autenticação.

Isto é feito também no **ConfigureServices**, no método AddMvc, desta forma.

services.AddMvc(config =>

{

var policy = new AuthorizationPolicyBuilder()

.RequireAuthenticatedUser()

.Build();

config.Filters.Add(new AuthorizeFilter(policy));

}).SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

O resultado final da nossa classe Startup.cs pode ser conferido abaixo.

using Microsoft.AspNetCore.Builder; using Microsoft.AspNetCore.Hosting; using Microsoft.Extensions.DependencyInjection; using Microsoft.IdentityModel.Tokens; using System.Text; using Microsoft.AspNetCore.Authentication.JwtBearer; using Microsoft.AspNetCore.Mvc; using Microsoft.AspNetCore.Authorization; using Microsoft.AspNetCore.Mvc.Authorization;

namespace shop

{

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddCors();

services.AddMvc(config =>

{

var policy = new AuthorizationPolicyBuilder()

.RequireAuthenticatedUser()

.Build();

config.Filters.Add(new AuthorizeFilter(policy));

}).SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

services.AddAuthorization(options =>

{

options.AddPolicy("user", policy => policy.RequireClaim("Store", "user"));

options.AddPolicy("admin", policy => policy.RequireClaim("Store", "admin"));

});

var key = Encoding.ASCII.GetBytes(Settings.Secret);

services.AddAuthentication(x =>

{

x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(x =>

{

x.RequireHttpsMetadata = false;

x.SaveToken = true;

x.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuerSigningKey = true,

IssuerSigningKey = new SymmetricSecurityKey(key),

ValidateIssuer = false,

ValidateAudience = false

};

});

}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

app.UseCors(x => x

.AllowAnyOrigin()

.AllowAnyMethod()

.AllowAnyHeader());

app.UseAuthentication();

app.UseMvc();

}

}

}

Gerando o Token

Para este exemplo, criei um serviço para geração do Token, para não ficar tudo no Controller, mas sinta-se à vontade em fazer de forma diferente.

Na verdade, tudo que precisamos fazer aqui é buscar um usuário do banco de dados, gerar um token e aplicar as políticas ao usuário. Não vou cobrir como buscar o usuário do banco, OK, não é o foco deste artigo!

O ASP.NET Core possui um **JwtSecurityTokenHandler**, que vai fazer todo o trabalho duro para nós, precisamos apenas passar para ele um **TokenDescriptor**, que nada mais é do que a definição do token a ser gerado, com os Claims a data de expiração e as credenciais de acesso.

Neste cenário, para não criar muitas classes também, adicionei uma propriedade Token no User, apenas para poder retornar o User direto para tela, sem precisar converter ele para uma ViewModel. Não faça isto em casa!

var tokenHandler = new JwtSecurityTokenHandler();

var key = Encoding.ASCII.GetBytes(Settings.Secret);

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new ClaimsIdentity(new Claim[]

{

new Claim(ClaimTypes.Name, user.Username),

new Claim("Store", user.Role)

}),

Expires = DateTime.UtcNow.AddDays(7),

SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)

};

var token = tokenHandler.CreateToken(tokenDescriptor);

user.Token = tokenHandler.WriteToken(token);

No **TokenDescriptor**, temos o Subject, que define o Claim principal, e seus Roles. Neste caso também deixei apenas um Role no usuário para facilitar. Se tiver mais de um Role, faz um ForEach e tá tudo certo.

Movendo adiante temos o Expires que é o tempo de vida do Token, como é uma API de exemplo e não quero ficar fazendo Refresh do Token, utilizei 7 dias, mas normalmente utilizo duas horas.

Por fim temos as credenciais de acesso, novamente utilizando nossa chave. O resultado final do nosso UserService é este.

using System;

using System.IdentityModel.Tokens.Jwt;

using System.Linq;

using System.Security.Claims;

using System.Text;

using Microsoft.IdentityModel.Tokens;

using shop.Database;

using store.Models;

namespace store.Services

{

public class UserService

{

public User Authenticate(string username, string password)

{

var user = Context.Users.Where(x => x.Username == username && x.Password == password).FirstOrDefault();

if (user == null)

return null;

var tokenHandler = new JwtSecurityTokenHandler();

var key = Encoding.ASCII.GetBytes(Settings.Secret);

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new ClaimsIdentity(new Claim[]

{

new Claim(ClaimTypes.Name, user.Username),

new Claim("Store", user.Role)

}),

Expires = DateTime.UtcNow.AddDays(7),

SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)

};

var token = tokenHandler.CreateToken(tokenDescriptor);

user.Token = tokenHandler.WriteToken(token);

user.Password = null;

return user;

}

}

}

Permitindo Acesso Anônimo

Quando adicionamos as políticas globais, nós restringimos todo acesso à nossa API, porém pelo menos o Login precisa ser liberado, e para isto, utilizamos o decorador **AllowAnonymous** nas ações dos nossos controladores.

[HttpPost("login")]

[AllowAnonymous]

public IActionResult Authenticate([Fro.....

Então, sempre que quiser liberar acesso a uma determinada ação do seu controlador, basta utilizar este decorador.

O resultado final do nosso AccountController fica desta forma.

using System; using Microsoft.AspNetCore.Authorization; using Microsoft.AspNetCore.Mvc; using store.Models; using store.Services;

namespace store.Controllers

{

[Route("v1/account")]

public class AccountController : Controller

{

[HttpPost("login")]

[AllowAnonymous]

public IActionResult Authenticate([FromBody]User model)

{

try

{

var service = new UserService();

var user = service.Authenticate(model.Username, model.Password);

if (user == null)

return BadRequest(new { message = "Usuário ou senha inválidos" });

return Ok(user);

}

catch (Exception ex)

{

return BadRequest(ex);

}

}

}

}

Restringindo Acesso por Perfil

Toda nossa API está restrita, porém a restrição é apenas para usuários anônimos, ou seja, se o usuário estiver autenticado, ele já tem acesso.

Nós criamos duas políticas de acesso para a API, uma para usuário (user) e outra para administrador (admin), definidas lá no Startup.cs.

Para utilizar estas políticas e restringir o acesso por perfil, basta utilizar o decorador **Authorize**, informando o perfil que deve ter acesso. É importante lembrar que as Strings definidas são Case Sensitive, então cuidado com isto.

[HttpGet]

[Route("")]

[Authorize("User")]

public IList<Category> Get() => Database.Context.Categories();

[HttpGet]

[Route("")]

[Authorize("Admin")]

public IList<Category> Get() => Database.Context.Categories();

Pronto! Temos uma API com autenticação e autorização com Claims e Policies de uma forma simples e fácil!