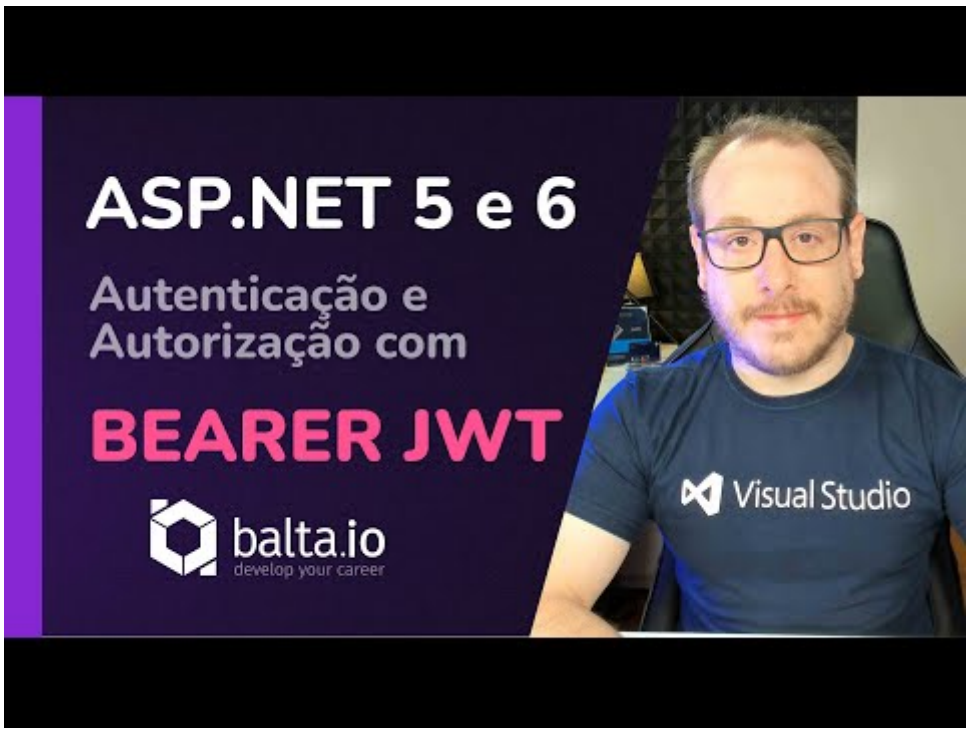
ASP.NET 5 – Autenticação e Autorização com Bearer e JWT

Versão atual

Este artigo atualmente utiliza a versão **5.0.0-rc.1** do ASP.NET/.NET, o que significa que ainda não é uma versão lançada oficialmente. Desta forma, mesmo que pouco provável, ainda podem ocorrer mudanças.

Referências

- ASP.NET Core - Autenticação e Autorização
- ASP.NET Core 3 – Autenticação e Autorização com Bearer e JWT
  - Versão anterior deste artigo com **ASP.NET Core 3**



Preparando o terreno

Bom, para executar a autenticação precisamos de um usuário, senha e seu perfil. Como o intuito deste artigo não é cobrir acesso à dados, vou fixar (Hard Code) os usuários em um repositório.

O esquema para gerar o Token vou separar em um serviço que você poderá reutilizar depois. Basta informar um usuário com um perfil que ele gerará o Token, então estamos totalmente independentes de banco de dados aqui.

Para testar, vamos criar dois itens, o Models/User.cs e o Repositories/UserRepository.cs, que tem a finalidade de modelar o usuário e "recuperá-lo" da fonte de dados respectivamente.

Vamos começar então criando nossa aplicação, executando os seguintes comandos:

```
dotnet new webapi -o Shop
cd Shop
dotnet restore
dotnet add package Microsoft.AspNetCore.Authentication
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Lembrando que para executar este artigo você precisa do .NET 5 instalado.

Em seguida, abra o projeto com o Visual Studio Code e remova os arquivos Weather.* gerado (Tem um na raiz e um na pasta Controllers).

Vamos então criar uma pasta na raiz chamada Models e dentro dela o arquivo User.cs com este código:

```
namespace Shop.Models
{
    public class User
    {
        public int Id { get; set; }
        public string Username { get; set; }
        public string Password { get; set; }
        public string Role { get; set; }
    }
}
```

Nosso próximo item é o repositório, e para isto vamos criar uma pasta na raiz chamada de Repositories e dentro dela o arquivo UserRepository.cs, com este código:

```
using System.Collections.Generic;
using System.Linq;
using Shop.Models;

namespace Shop.Repositories
{
    public static class UserRepository
    {
        public static User Get(string username, string password)
        {
            var users = new List<User>();
            users.Add(new User { Id = 1, Username = "batman", Password = "batman", Role = "
            users.Add(new User { Id = 2, Username = "robin", Password = "robin", Role = "em
            return users.Where(x => x.Username.ToLower() == username.ToLower() && x.Passwor
        }
    }
}
```

Para finalizar precisaremos de uma chave, um segredo, para gerar o Token, esta chave deve ser uma String que só o servidor conhece, ela é chamada de Chave Privada.

Para este cenário, vamos criar um arquivo chamado de Settings.cs na raiz da aplicação com o seguinte código:

```
namespace Shop
{
    public static class Settings
    {
        public static string Secret = "fedaf7d8863b48e197b9287d492b708e";
    }
}
```

Este arquivo tem a finalidade apenas de servir nossa chave, nada além disso. Se quiser deixar nas configurações fica ainda melhor, mas pra facilitar coloquei em um arquivo estático para este artigo.

Gerando um Token

No pacote que adicionamos (Microsoft.Authentication.Jwt) temos uma classe chamada JwtSecurityTokenHandler que é utilizada para gerar um Token baseado em algumas informações que podemos prover.

Estas informações são chamadas de SecurityTokenDescriptor e elas proveem dentre outras opções, o usuário e perfil, tornando-os também disponíveis no ClaimIdentity do ASP.NET.

Ou seja, ele gera o Token e ainda deixa o usuário disponível via User.Identity.Name para nossos Controllers.

Podemos gerar um Token simplesmente chamando o JwtSecurityTokenHandler.CreateToken e informando um SecurityTokenDescriptor, e embora isto possa ser feito dentro de um Controller, eu prefiro externaliza-lo para um serviço e poder chamá-lo onde eu quiser, sem precisar duplicar seu código.

Vamos então criar uma pasta na raiz chamada Services com o arquivo TokenService.cs dentro dela. Este arquivo terá o seguinte código:

```
using System;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using Shop.Models;
using Microsoft.IdentityModel.Tokens;

namespace Shop.Services
{
    public static class TokenService
    {
        public static string GenerateToken(User user)
        {
            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.ToString()),
                    new Claim(ClaimTypes.Role, user.Role.ToString())
                }),
                Expires = DateTime.UtcNow.AddHours(2),
                SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key),
            };
            var token = tokenHandler.CreateToken(tokenDescriptor);
            return tokenHandler.WriteToken(token);
        }
    }
}
```

Acredite ou não, isto é tudo que precisamos para gerar um Token no formato JWT com ASP.NET 5.

Note que na linha quinze utilizamos a chave que criamos no Settings.cs para gerar um Chave Privada, ou seja, só conseguirão desencriptar este Token com esta chave, que só existe no nosso servidor.

Adicionando autenticação e autorização

Até o momento, temos nosso usuário, seu repositório e um serviço que gera um Token e adiciona o usuário ao ClaimPrincipal baseado em um usuário recebido.

Precisamos agora informar para aplicação que queremos trabalhar com autenticação, autorização e que o formato do Token é o JWT. Deste modo ela será capaz de descriptografar nosso Token a cada requisição, e conseguiremos definir quais perfis tem acesso a determinadas ações dos controladores.

O primeiro passo é informar para o ASP.NET que estamos utilizando autenticação, e para isto utilizamos o recurso AddAuthentication no método ConfigureServices do Startup.cs.

```
services.AddAuthentication(x =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
```

Ao informar sobre a autenticação, já informamos também o tipo dela, neste caso JwtBearer e seu modelo de autenticação (Challenge). Neste caso, ambos ficam como padrão do JwtBearer.

Depois de dizer que vamos utilizar JwtBearer, precisamos configurá-lo, e isto é feito no método AddJwtBearer, ainda no ConfigureServices.

Nosso método ConfigureServices final fica com este código:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors();
    services.AddControllers();

    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
        };
    });
}
```

Para fechar, precisamos dizer ao ASP.NET que de fato estamos utilizando autenticação e autorização nesta API e isto é feito no método Configure, adicionando as seguintes linhas:

```
app.UseAuthentication();
app.UseAuthorization();
```

Como resultado, nosso arquivo Startup.cs fica com este código:

```
using System.Text;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.IdentityModel.Tokens;

namespace Shop
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddCors();
            services.AddControllers();

            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, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
```

```
            }

            app.UseHttpsRedirection();

            app.UseRouting();

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

            app.UseAuthentication();
            app.UseAuthorization();


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

Isto é tudo que precisamos informar ao ASP.NET sobre nosso formato de autenticação e autorização.

Autenticando

Para este exemplo, criei um arquivo chamado de HomeController.cs na pasta Controllers, na raiz da API. Vamos explorar toda a autenticação e autorização aqui dentro.

Seguindo este modelo, nosso método para autenticação fica com este código:

```
[HttpPost]
[Route("login")]
public async Task<ActionResult<dynamic>> Authenticate([FromBody]User model)
{
    // Recupera o usuário
    var user = UserRepository.Get(model.Username, model.Password);

    // Verifica se o usuário existe
    if (user == null)
        return NotFound(new { message = "Usuário ou senha inválidos" });

    // Gera o Token
    var token = TokenService.GenerateToken(user);

    // Oculta a senha
    user.Password = "";

    // Retorna os dados
    return new
    {
        user = user,
        token = token
    };
}
```

Vamos então rodar nossa API com o comando dotnet watch run e em seguida fazer uma requisição do tipo POST com o Postman para a URL https://localhost:5001/v1/account/login informando o seguinte JSON no corpo da mesma:

```
{'username':'robin','password':'robin'}
```

Como resultado, teremos o usuário Robin e um Token contendo seus dados e perfil. Agora podemos com este Token fazer requisições a API.

Autorizando

Para ficar legal a brincadeira, vamos criar mais quatro métodos no HomeController, um que qualquer usuário pode acessar, outro que somente usuários autenticados podem acessar e por fim dois métodos que só funcionários e gerentes podem acessar. Estas autorizações são realizadas através dos decoradores:

- AllowAnonymous – Permite acesso anônimo, sem autenticação alguma a página.
- Authorize – Requer que o usuário esteja autenticado, mas não se importa com seu perfil.
- Authorize(Roles=”Role1,Role2”) – Exige que o usuário além de autenticado, faça parte de um dos perfis listados. Note que se quiser restringir por mais de um perfil, basta utilizar “,” para separá-los.

Desta forma, decoraremos nossos métodos com os seguintes:

```
[HttpGet]
[Route("anonymous")]
[AllowAnonymous]
public string Anonymous() => "Anônimo";

[HttpGet]
[Route("authenticated")]
[Authorize]
public string Authenticated() => String.Format("Autenticado - {0}", User.Identity.Name);

[HttpGet]
[Route("employee")]
[Authorize(Roles = "employee,manager")]
public string Employee() => "Funcionário";

[HttpGet]
[Route("manager")]
[Authorize(Roles = "manager")]
public string Manager() => "Gerente";
```

Como resultado final, o HomeController fica com este código:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Shop.Models;
using System;
using Microsoft.AspNetCore.Authorization;
using System.Linq;
using Shop.Services;
using Shop.Repositories;

namespace Shop.Controllers
{
    [Route("v1/account")]
    public class HomeController : Controller
    {
        [HttpPost]
        [Route("login")]
        [AllowAnonymous]
        public async Task<ActionResult<dynamic>> Authenticate([FromBody]User model)
        {
            var user = UserRepository.Get(model.Username, model.Password);

            if (user == null)
                return NotFound(new { message = "Usuário ou senha inválidos" });

            var token = TokenService.GenerateToken(user);
            user.Password = "";
            return new
            {
                user = user,
                token = token
            };
        }

        [HttpGet]
        [Route("anonymous")]
        [AllowAnonymous]
        public string Anonymous() => "Anônimo";

        [HttpGet]
        [Route("authenticated")]
        [Authorize]
        public string Authenticated() => String.Format("Autenticado - {0}", User.Identity.N

        [HttpGet]
        [Route("employee")]
        [Authorize(Roles = "employee,manager")]
        public string Employee() => "Funcionário";

        [HttpGet]
```

```
        [Route("manager")]
        [Authorize(Roles = "manager")]
        public string Manager() => "Gerente";

    }
}
```

Agora é só salvar o arquivo e ir ao Postman fazer quatro requisições GET:

- https://localhost:5001/v1/account/anonymous
- https://localhost:5001/v1/account/authenticated
- https://localhost:5001/v1/account/employee
- https://localhost:5001/v1/account/manager

Não se esqueça de informar o Token na aba Headers das requisições que precisam de autenticação, ela deve ficar no seguinte formato:

```
Chave: Authorization
Valor: Bearer SEUTOKEN
```

Recuperando o usuário logado

Você deve ter percebido que na linha quarenta e três do HomeController.cs nós recuperamos o usuário logado. Isto é feito através da propriedade User.Identity.Name, que é preenchida automaticamente cada vez que um Token é enviado no cabeçalho da requisição.

Em adicional também temos os métodos User.IsInRole e demais do ClaimPrincipal do ASP.NET. Estes valores foram definidos no TokenService, na parte dos Claims.

Eu concordo em receber comunicações.

Me avise, Balta!

Populares

Priority Queue

Priority Queue ou fila prioritária é um tipo de lista inclusa no C# 10 que permite que seus itens...

Implicit Operators no C#

Implicit Operators permitem adicionar comportamentos de conversão a objetos Built In ou complexos...

ASP.NET 5 – Autenticação e Autorização com Bearer e JWT

Este artigo atualmente utiliza a versão 5.0.0-rc.1 do ASP.NET/.NET, o que significa que ainda não...

Clean Code - Guia e Exemplos

Saiba como manter seu código limpo (Clean Code) seguindo algumas práticas sugeridas pelo Robert C...

Git e GitHub - Instalação, Configuração e Primeiros Passos

Git é um sistema de controle de versões distribuídas, enquanto GitHub é uma plataforma que tem o ...

Compartilhe este artigo

Conheça o autor

André Baltieri
Microsoft MVP

Me dedico ao desenvolvimento de software desde 2003, sendo minha maior especialidade o Desenvolvimento Web. Durante esta jornada pude trabalhar presencialmente aqui no Brasil e Estados Unidos, atender remotamente times da ?ndia, Inglaterra e Holanda, receber **8x Microsoft MVP** e realizar diversas consultorias em empresas e projetos de todos os tamanhos.

2.199
Aulas disponíveis

218
horas de conteúdo

45.254
Alunos matriculados

31.417
Certificados emitidos

Comece de graça agora mesmo!

Temos mais de 16 cursos totalmente de graça e todos com certificado de conclusão.

Começar

Prefere algo mais Premium?

Conheça nossos planos

Premium semestral

Compra única, parcelada em até
12x no cartão de crédito

12x R$ 38,33 =R$ 459,90

- ✓ 6 meses de acesso
- ✓ Acesso à todo conteúdo
- ✓ Emissão de Certificado
- ✓ Tira Dúvidas Online
- ✓ 48 cursos disponíveis
- ✓ 4 carreiras disponíveis
- ✓ 161 temas de tecnologia
- ✓ Conteúdo novo todo mês
- ✓ Encontros Premium

Começar agora

Política de privacidade

Premium anual

Compra única, parcelada em até
12x no cartão de crédito

12x R$ **66 ,65**
=R$ 799,90

- ✓ 1 ano de acesso
- ✓ Acesso à todo conteúdo
- ✓ Emissão de Certificado
- ✓ Tira Dúvidas Online
- ✓ 48 cursos disponíveis
- ✓ 4 carreiras disponíveis
- ✓ 161 temas de tecnologia
- ✓ Conteúdo novo todo mês
- ✓ Encontros Premium

Começar agora
Política de privacidade

Precisa de ajuda?

Dúvidas frequentes

Posso começar de graça?

Vou ter que pagar algo?

Por onde devo começar?

Os cursos ensinam tudo que preciso?

O que eu devo estudar?

Estou pronto para estudar no balta.io?

Ainda tem dúvidas?

Assine nosso Newsletter

Receba em primeira mão todas as nossas novidades.

Cadastrar

Sobre

Como funciona?

Seja Premium

Agenda

Blog

Todos os cursos

Cursos

Frontend

Backend

Mobile

Fullstack

Suporte

Termos de uso

Privacidade

Cancelamento

Central de ajuda

Redes Sociais

Telegram

Facebook

Instagram

YouTube

Twitch

LinkedIn

Discord