Authentication And Authorization In ASP.NET Core Web API With JSON Web Tokens

Sarathlal Saseendran Updated date Sep 19, 2020

456.3k 40 61

JWTAuthentication.zip

Introduction

Authentication is the process of validating user credentials and authorization is the process of checking privileges for a user to access specific modules in an application. In this article, we will see how to protect an ASP.NET Core Web API application by implementing JWT authentication. We will also see how to use authorization in ASP.NET Core to provide access to various functionality of the application. We will store user credentials in an SQL server database and we will use Entity framework and Identity framework for database operations.

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**.

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

- Header
- Payload
- Signature

Therefore, a JWT typically looks like the following.

xxxx.yyyy.zzzz

Please refer to below link for more details about JSON Web Tokens.

https://jwt.io/introduction/

Create ASP.NET Core Web API using Visual Studio 2019

We can create an API application with ASP.NET Core Web API template.

We must install below libraries using NuGet package manager.

- **Microsoft.EntityFrameworkCore.SqlServer**
- **Microsoft.EntityFrameworkCore.Tools**
- **Microsoft.AspNetCore.Identity.EntityFrameworkCore**
- **Microsoft.AspNetCore.Identity**
- **Microsoft.AspNetCore.Authentication.JwtBearer**

We can modify the appsettings.json with below values.

appsettings.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "ConnStr": "Data Source=
(localdb)\\MSSQLLocalDB;Initial Catalog=SarathlalDB;Integrated Security=True;Ap
  },
  "JWT": {
    "ValidAudience": "http://localhost:4200",
    "ValidIssuer": "http://localhost:61955",
    "Secret": "ByYM000OLlMQG6VVVp1OH7Xzyr7gHuw1qvUC5dcGt3SNM"
  }
}
```

FEATURED ARTICLES
How To Upgrade to Windows 11
Exploring Subject <T> In Reactive Extensions For .Net
Micro Frontends With Webpack
What's New In iPhone 13
Understanding Synchronization Context Task.ConfigureAwait In Action
View All

TRENDING UP
01 The Best VS Code Extensions For Remote Working
02 Getting Started With .NET 6.0
03 How To Post Data In ASP.NET Core Using Ajax
04 Visual Studio 2022 Installation Step By Step
05 New Features In Blazor With .NET 6
06 Debugging The Hottest Release Of Visual Studio With Code Demos
07 Implementing Smart Contract in ASP.NET Application
08 The Best VS Code Extensions To Supercharge Git
09 Getting Started With ASP.NET Core 6.0
10 7 Minutes to better Selling Podcast Ep. 9
View All

We have added a database connection string and also added valid audience, valid issuer and secret key for JWT authentication in above settings file.

Create an “ApplicationUser” class inside a new folder “Authentication” which will inherit the IdentityUser class. IdentityUser class is a part of Microsoft Identity framework. We will create all the authentication related files inside the “Authentication” folder.

ApplicationUser.cs

```
using Microsoft.AspNetCore.Identity;

namespace JWTAuthentication.Authentication
{
    public class ApplicationUser: IdentityUser
    {
    }
}
```

We can create the “ApplicationDbContext” class and add below code.

ApplicationDbContext.cs

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace JWTAuthentication.Authentication
{
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> opti
        {

        }
        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
        }
    }
}
```

Create a static class “UserRoles” and add below values.

UserRoles.cs

```
namespace JWTAuthentication.Authentication
{
    public static class UserRoles
    {
        public const string Admin = "Admin";
        public const string User = "User";
    }
}
```

We have added two constant values “Admin” and “User” as roles. You can add many roles as you wish.

Create class “RegisterModel” for new user registration.

RegisterModel.cs

```
using System.ComponentModel.DataAnnotations;

namespace JWTAuthentication.Authentication
{
    public class RegisterModel
    {
        [Required(ErrorMessage = "User Name is required")]
        public string Username { get; set; }

        [EmailAddress]
        [Required(ErrorMessage = "Email is required")]
        public string Email { get; set; }

        [Required(ErrorMessage = "Password is required")]
        public string Password { get; set; }

    }
}
```

Create class “LoginModel” for user login.

LoginModel.cs

```
using System.ComponentModel.DataAnnotations;

namespace JWTAuthentication.Authentication
{
    public class LoginModel
    {
        [Required(ErrorMessage = "User Name is required")]
        public string Username { get; set; }

        [Required(ErrorMessage = "Password is required")]
        public string Password { get; set; }
    }
}
```

We can create a class “Response” for returning the response value after user registration and user login. It will also return error messages, if the request fails.

Response.cs

```
namespace JWTAuthentication.Authentication
{
    public class Response
    {
        public string Status { get; set; }
        public string Message { get; set; }
    }
}
```

We can create an API controller “AuthenticateController” inside the “Controllers” folder and add below code.

AuthenticateController.cs

```
using JWTAuthentication.Authentication;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.IdentityModel.Tokens;
using System;
using System.Collections.Generic;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using System.Threading.Tasks;

namespace JWTAuthentication.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AuthenticateController : ControllerBase
    {
        private readonly UserManager<ApplicationUser> userManager;
        private readonly RoleManager<IdentityRole> roleManager;
        private readonly IConfiguration _configuration;

        public AuthenticateController(UserManager<ApplicationUser> userManager,
        {
            this.userManager = userManager;
            this.roleManager = roleManager;
            _configuration = configuration;
        }

        [HttpPost]
        [Route("login")]
        public async Task<IActionResult> Login([FromBody] LoginModel model)
        {
            var user = await userManager.FindByNameAsync(model.Username);
            if (user != null && await userManager.CheckPasswordAsync(user, mode
            {
                var userRoles = await userManager.GetRolesAsync(user);

                var authClaims = new List<Claim>
                {
                    new Claim(ClaimTypes.Name, user.UserName),
                    new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToStr
                };

                foreach (var userRole in userRoles)
                {
                    authClaims.Add(new Claim(ClaimTypes.Role, userRole));
                }

                var authSigningKey = new SymmetricSecurityKey(Encoding.UTF8.Get

                var token = new JwtSecurityToken(
                    issuer: _configuration["JWT:ValidIssuer"],
                    audience: _configuration["JWT:ValidAudience"],
                    expires: DateTime.Now.AddHours(3),
                    claims: authClaims,
                    signingCredentials: new SigningCredentials(authSigningKey,
                    );

                return Ok(new
                {
                    token = new JwtSecurityTokenHandler().WriteToken(token),
                    expiration = token.ValidTo
                });
            }
            return Unauthorized();
        }

        [HttpPost]
        [Route("register")]
        public async Task<IActionResult> Register([FromBody] RegisterModel mode
        {
            var userExists = await userManager.FindByNameAsync(model.Username);
            if (userExists != null)
                return StatusCode(StatusCodes.Status500InternalServerError, new

            ApplicationUser user = new ApplicationUser()
            {
                Email = model.Email,
```

```
                SecurityStamp = Guid.NewGuid().ToString(),
                UserName = model.Username
            };
            var result = await userManager.CreateAsync(user, model.Password);
            if (!result.Succeeded)
                return StatusCode(StatusCodes.Status500InternalServerError, new

            return Ok(new Response { Status = "Success", Message = "User create
        }

        [HttpPost]
        [Route("register-admin")]
        public async Task<IActionResult> RegisterAdmin([FromBody] RegisterModel
        {
            var userExists = await userManager.FindByNameAsync(model.Username);
            if (userExists != null)
                return StatusCode(StatusCodes.Status500InternalServerError, new

            ApplicationUser user = new ApplicationUser()
            {
                Email = model.Email,
                SecurityStamp = Guid.NewGuid().ToString(),
                UserName = model.Username
            };
            var result = await userManager.CreateAsync(user, model.Password);
            if (!result.Succeeded)
                return StatusCode(StatusCodes.Status500InternalServerError, new

            if (!await roleManager.RoleExistsAsync(UserRoles.Admin))
                await roleManager.CreateAsync(new IdentityRole(UserRoles.Admin)
            if (!await roleManager.RoleExistsAsync(UserRoles.User))
                await roleManager.CreateAsync(new IdentityRole(UserRoles.User))

            if (await roleManager.RoleExistsAsync(UserRoles.Admin))
            {
                await userManager.AddToRoleAsync(user, UserRoles.Admin);
            }

            return Ok(new Response { Status = "Success", Message = "User create
        }
    }
}
```

We have added three methods “login”, “register”, and “register-admin” inside the controller class. Register and register-admin are almost same but the register-admin method will be used to create a user with admin role. In login method, we have returned a JWT token after successful login.

We can make below changes in “ConfigureServices” and “Configure” methods in “Startup” class as well.

Startup.cs

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

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

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add servi
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllers();

            // For Entity Framework
            services.AddDbContext<ApplicationDbContext>
(options => options.UseSqlServer(Configuration.GetConnectionString("ConnStr")))

            // For Identity
            services.AddIdentity<ApplicationUser, IdentityRole>()
                .AddEntityFrameworkStores<ApplicationDbContext>()
                .AddDefaultTokenProviders();

            // Adding Authentication
            services.AddAuthentication(options =>
            {
                options.DefaultAuthenticateScheme = JwtBearerDefaults.Authentic
                options.DefaultChallengeScheme = JwtBearerDefaults.Authenticati
                options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
            })

            // Adding Jwt Bearer
            .AddJwtBearer(options =>
```

```
            {
                options.SaveToken = true;
                options.RequireHttpsMetadata = false;
                options.TokenValidationParameters = new TokenValidationParamete
                {
                    ValidateIssuer = true,
                    ValidateAudience = true,
                    ValidAudience = Configuration["JWT:ValidAudience"],
                    ValidIssuer = Configuration["JWT:ValidIssuer"],
                    IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.G
                };
            });
        }

        // This method gets called by the runtime. Use this method to configure
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseRouting();

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

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

We can add “Authorize” attribute inside the “WeatherForecast” controller.

We must create a database and required tables before running the application. As we are using entity framework, we can use below database migration command with package manger console to create a migration script.

“add-migration Initial”

Use below command to create database and tables.

“update-database”

If you check the database using SQL server object explorer, you can see that below tables are created inside the database.

Above seven tables are used by identity framework to manage authentication and authorization.

We can run the application and try to access get method in weatherforecast controller from Postman tool.

We have received a 401 unauthorized error. Because, we have added Authorize attribute to entire controller. We must provide a valid token via request header to access this controller and methods inside the controller.

We can create a new user using register method in authenticate controller.

We can use above user credentials to login and get a valid JWT token.

We have received a token after successful login with above credentials.

We can pass above token value as a bearer token inside the authorization tab and call get method of weatherforecast controller again.

This time, we have successfully received the values from controller.

We can modify the weatherforecast controller with role-based authorization.

Now, only users with admin role can access this controller and methods.

We can try to access the weatherforecast controller with same token again in Postman tool.

We have received a 403 forbidden error now. Even though, we are passing a valid token we don't have sufficient privilege to access the controller. To access this controller, user must have an admin role permission. Current user is a normal user and do not have any admin role permission.

We can create a new user with admin role. We already have a method "register-admin" in authenticate controller for the same purpose.

We can login with this new user credentials and get a new token and use this token instead of old token to access the weatherforecast controller.

We have again received the values from weatherforecast controller successfully.

We can see the token payload and other details using **jwt.io** site.

Inside the payload section, you can see the user name, role and other details as claims.

Conclusion

In this post, we have seen how to create a JSON web token in ASP.NET Core Web API application and use this token for authentication and authorization. We have created two users, one without any role and one with admin role. We have applied the authentication and authorization in controller level and saw the different behaviors with these two users.

ASP.NET Core Web API | Authentication in ASP.NET Core | Authorization in ASP.NET Core | Json Web Tokens | JWT | JWT Tokens | Web API

Next Recommended Reading
ASP.NET Core Web API 5.0 Authentication Using JWT(JSON BASE TOKEN)


Sarathlal Saseendran TOP 500
115 5.6m 3
A passionate human being loves to learn new things always.
https://www.linkedin.com/in/sarathlal-saseendran/

Hello, Thanks for the article. The code "var userExists = await userManager.FindByNameAsync(model.Username);" in file authenticate controller does not return null even though user is not present instead it returns task with status faulted. Can you guide me why this could be the case?

Rehman Ali

1974 101 0

Aug 24, 2021

0 0 Reply

Good Article.....

Amit Gautam

1888 187 0

Aug 21, 2021

0 0 Reply

Great job.. How can I implement refresh token in this example??

Abhijit Das

2047 28 0

Jul 26, 2021

0 0 Reply

It is a very good and clear article. To help other with some tips, in IIS you must open the port 61955 in web site's bindings in order to do calls with postman. Moreover, if someone apply that instuction in his applicatation then in order to create successfully the authentication tables the (-context ApplicationDbContext) must be used in both Package manager Console commands. e.g. (add-migration Initial -Context ApplicationDbContext) and (update-database -Context ApplicationDbContext). I hope it help someone from losing time!

Κωνσταντίνος Μαλλιαρίδης

2067 8 0

Jul 15, 2021

1 0 Reply

Great Article, Thanks For Share

Ajay Kumar

1474 602 52.3k

May 09, 2021

0 0 Reply

Nice Article

Mayank Mishra

1978 97 0

May 03, 2021

0 0 Reply