PUBLISHED: OCTOBER 11 2019
LAST UPDATED: APRIL 29 2020

ASP.NET Core 3.1 - JWT Authentication Tutorial with Example API

Tutorial built with **ASP.NET Core 3.1**

Other versions available:

- **ASP.NET:** ASP.NET Core 2.2
- **Node:** Node.js

In this tutorial we'll go through a simple example of how to implement JWT (JSON Web Token) authentication in an ASP.NET Core 3.1 API with C#.

The example API has just two endpoints/routes to demonstrate authenticating with JWT and accessing a restricted route with JWT:

- `/users/authenticate` - public route that accepts HTTP POST requests containing the username and password in the body. If the username and password are correct then a JWT authentication token and the user details are returned.
- `/users` - secure route that accepts HTTP GET requests and returns a list of all the users in the application if the HTTP Authorization header contains a valid JWT token. If there is no auth token or the token is invalid then a 401 Unauthorized response is returned.

The tutorial project is available on GitHub at https://github.com/cornflourblue/aspnet-core-3-jwt-authentication-api.

Update History:

- 29 Apr 2020 - Added `[JsonIgnore]` attribute to password prop on user entity and removed `WithoutPassword()` extension methods, thanks for the comment Héctor Damián Simañuk.
- 13 Dec 2019 - Updated to ASP.NET Core 3.1 (Git commit showing the changes available here)
- 11 Oct 2019 - Built with ASP.NET Core 3.0

Tutorial Contents

Tools required to run the ASP.NET Core 3.1 JWT Example Locally

To develop and run ASP.NET Core applications locally, download and install the following:

- .NET Core SDK - includes the .NET Core runtime and command line tools
- Visual Studio Code - code editor that runs on Windows, Mac and Linux
- C# extension for Visual Studio Code - adds support to VS Code for developing .NET Core applications

Running the ASP.NET Core JWT Authentication API Locally

1. Download or clone the tutorial project code from https://github.com/cornflourblue/aspnet-core-3-jwt-authentication-api
2. Start the api by running `dotnet run` from the command line in the project root folder (where the WebApi.csproj file is located), you should see the message `Now listening on: http://localhost:4000`. Follow the instructions below to test with Postman or hook up with one of the example single page applications available (Angular, React or Vue).

NOTE: You can also start the application in debug mode in VS Code by opening the project root folder in VS Code and pressing F5 or by selecting Debug -> Start Debugging from the top menu. Running in debug mode allows you to attach breakpoints to pause execution and step through the application code.

Testing the ASP.NET Core JWT Auth API with Postman

Postman is a great tool for testing APIs, you can download it at https://www.getpostman.com/.

Below are instructions on how to use Postman to authenticate a user to get a JWT token from the api, and then make an authenticated request with the JWT token to retrieve a list of users from the api.

How to authenticate a user with Postman

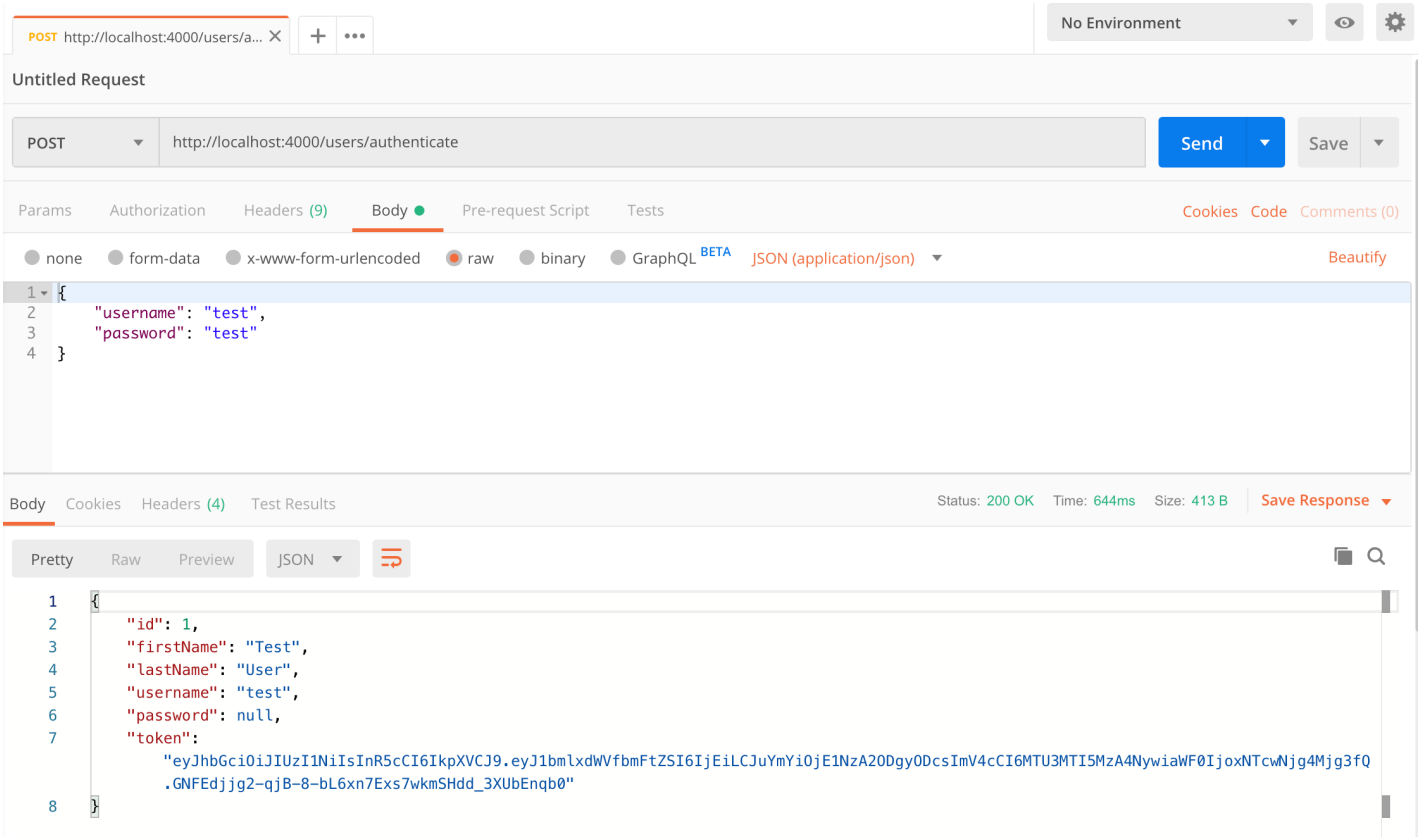
To authenticate a user with the api and get a JWT token follow these steps:

1. Open a new request tab by clicking the plus (+) button at the end of the tabs.
2. Change the http request method to "POST" with the dropdown selector on the left of the URL input field.
3. In the URL field enter the address to the authenticate route of your local API - `http://localhost:4000/users/authenticate`.
4. Select the "Body" tab below the URL field, change the body type radio button to "raw", and change the format dropdown selector to "JSON (application/json)".
5. Enter a JSON object containing the test username and password in the "Body" textarea:

```
{
    "username": "test",
    "password": "test"
}
```

6. Click the "Send" button, you should receive a "200 OK" response with the user details including a JWT token in the response body, make a copy of the token value because we'll be using it in the next step to make an authenticated request.

Here's a screenshot of Postman after the request is sent and the user has been authenticated:

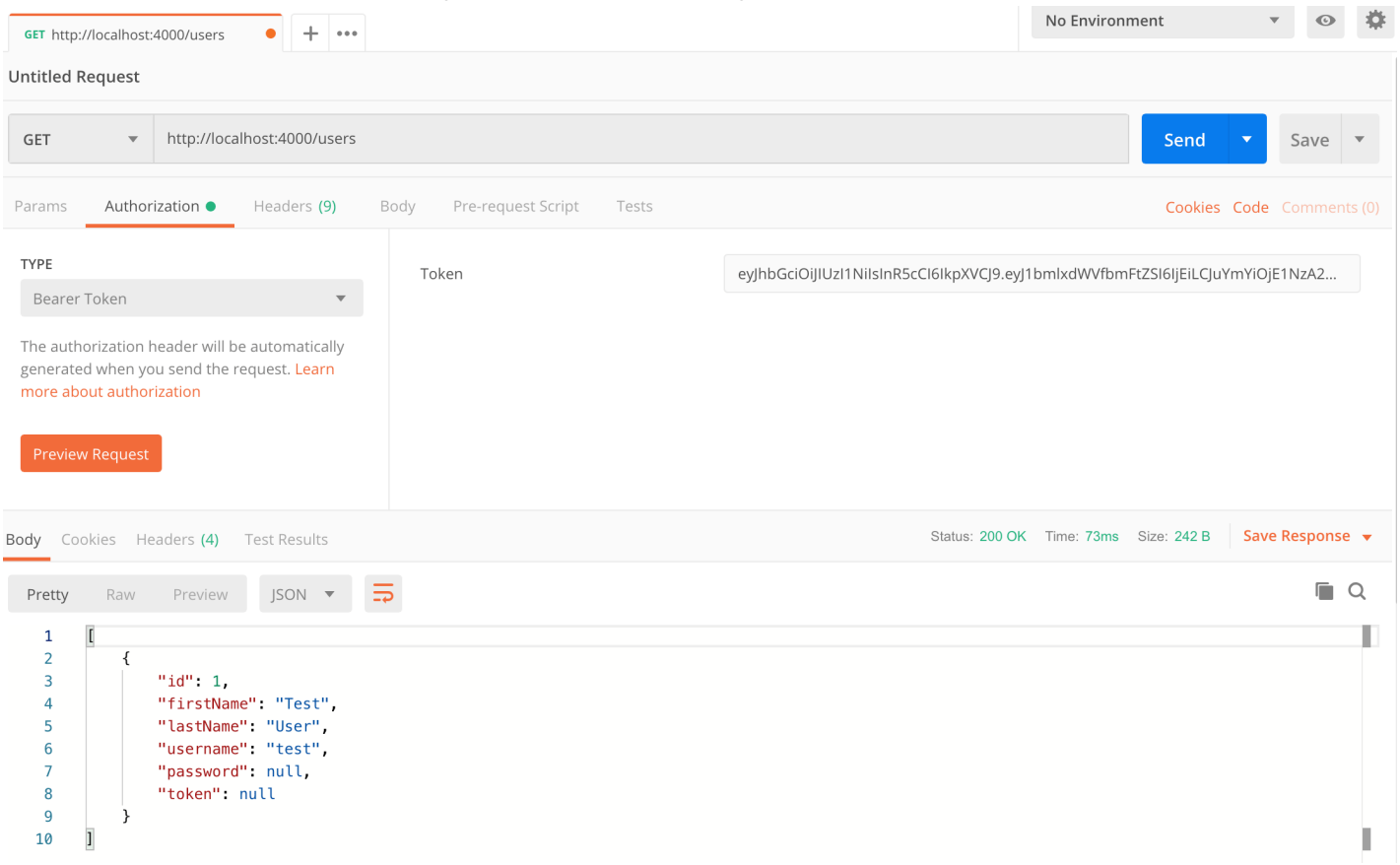


How to make an authenticated request to retrieve all users

To make an authenticated request using the JWT token from the previous step, follow these steps:

1. Open a new request tab by clicking the plus (+) button at the end of the tabs.
2. Change the http request method to "GET" with the dropdown selector on the left of the URL input field.
3. In the URL field enter the address to the users route of your local API - `http://localhost:4000/users`.
4. Select the "Authorization" tab below the URL field, change the type to "Bearer Token" in the type dropdown selector, and paste the JWT token from the previous authenticate step into the "Token" field.
5. Click the "Send" button, you should receive a "200 OK" response containing a JSON array with all the user records in the system (just the one test user in the example).

Here's a screenshot of Postman after making an authenticated request to get all users:



Running an Angular 9 client app with the ASP.NET Core JWT Auth API

For full details about the example Angular 9 application see the post Angular 9 - JWT Authentication Example & Tutorial. But to get up and running quickly just follow the below steps.

1. Download or clone the Angular 9 tutorial code from https://github.com/cornflourblue/angular-9-jwt-authentication-example
2. Install all required npm packages by running `npm install` from the command line in the project root folder (where the package.json is located).
3. Remove or comment out the line below the comment `// provider used to create fake backend` located in the `/src/app/app.module.ts` file.
4. Start the application by running `npm start` from the command line in the project root folder, this will launch a browser displaying the Angular example application and it should be hooked up with the ASP.NET Core JWT Auth API that you already have running.

Running a React client app with the ASP.NET Core JWT Auth API

For full details about the example React application see the post React - JWT Authentication Tutorial & Example. But to get up and running quickly just follow the below steps.

1. Download or clone the React tutorial code from https://github.com/cornflourblue/react-jwt-authentication-example
2. Install all required npm packages by running `npm install` from the command line in the project root folder (where the package.json is located).
3. Remove or comment out the 2 lines below the comment `// setup fake backend` located in the `/src/index.jsx` file.
4. Start the application by running `npm start` from the command line in the project root folder, this will launch a browser displaying the React example application and it should be hooked up with the ASP.NET Core JWT Auth API that you already have running.

Running a Vue.js client app with the ASP.NET Core JWT Auth API

For full details about the example VueJS JWT application see the post Vue.js + Vuex - JWT Authentication Tutorial & Example. But to get up and running quickly just follow the below steps.

1. Download or clone the VueJS tutorial code from https://github.com/cornflourblue/vue-vuex-jwt-authentication-example
2. Install all required npm packages by running `npm install` from the command line in the project root folder (where the package.json is located).
3. Remove or comment out the 2 lines below the comment `// setup fake backend` located in the `/src/index.js` file.
4. Start the application by running `npm start` from the command line in the project root folder, this will launch a browser displaying the VueJS example application and it should be hooked up with the ASP.NET Core JWT Auth API that you already have running.

ASP.NET Core JWT Authentication Project Structure

The tutorial project is organised into the following folders:

Controllers - define the end points / routes for the web api, controllers are the entry point into the web api from client applications via http requests.

Models - represent request and response models for controller methods, request models define the parameters for incoming requests, and response models can be used to define what data is returned.

Services - contain business logic, validation and data access code.

Entities - represent the application data.

Helpers - anything that doesn't fit into the above folders.

Click any of the below links to jump down to a description of each file along with its code:

ASP.NET Core JWT Users Controller

Path: /Controllers/UsersController.cs

The ASP.NET Core users controller defines and handles all routes / endpoints for the api that relate to users, this includes authentication and standard CRUD operations. Within each route the controller calls the user service to perform the action required, this enables the controller to stay 'lean' and completely separated from the business logic and data access code.

The controller actions are secured with JWT using the `[Authorize]` attribute, with the exception of the `Authenticate` method which allows public access by overriding the `[Authorize]` attribute on the controller with `[AllowAnonymous]` attribute on the action method. I chose this approach so any new action methods added to the controller will be secure by default unless explicitly made public.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;
using WebApi.Services;
using WebApi.Models;
using System.Linq;

namespace WebApi.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class UsersController : ControllerBase
    {
        private IUserService _userService;

        public UsersController(IUserService userService)
        {
            _userService = userService;
        }

        [AllowAnonymous]
        [HttpPost("authenticate")]
        public IActionResult Authenticate([FromBody]AuthenticateModel model)
        {
            var user = _userService.Authenticate(model.Username, model.Password);

            if (user == null)
                return BadRequest(new { message = "Username or password is incorrect" });

            return Ok(user);
        }

        [HttpGet]
        public IActionResult GetAll()
        {
            var users = _userService.GetAll();
            return Ok(users);
        }
    }
}
```

Back to top

ASP.NET Core JWT User Entity

Path: /Entities/User.cs

The user entity class represents the data for a user in the application. Entity classes are used to pass data between different parts of the application (e.g. between services and controllers) and can be used to return http response data from controller action methods. If multiple types of entities or other custom data is required to be returned from a controller method then a custom model class should be created in the `Models` folder for the response.

The `[JsonIgnore]` attribute prevents the password property from being serialized and returned in api responses.

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

namespace WebApi.Entities
{
    public class User
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Username { get; set; }

        [JsonIgnore]
        public string Password { get; set; }

        public string Token { get; set; }
    }
}
```

Back to top

ASP.NET Core JWT App Settings

Path: /Helpers/AppSettings.cs

The app settings class contains properties defined in the appsettings.json file and is used for accessing application settings via objects that are injected into classes using the ASP.NET Core built in dependency injection (DI) system. For example the User Service accesses app settings via an `IOptions<AppSettings> appSettings` object that is injected into the constructor.

Mapping of configuration sections to classes is done in the `ConfigureServices` method of the Startup.cs file.

```
namespace WebApi.Helpers
{
    public class AppSettings
    {
        public string Secret { get; set; }
    }
}
```

Back to top

ASP.NET Core JWT Authenticate Model

Path: /Models/AuthenticateModel.cs

The authenticate model defines the parameters for incoming requests to the `/users/authenticate` route of the api, because it is set as the parameter to the `Authenticate` method of the UsersController. When an HTTP POST request is received to the route, the data from the body is bound to an instance of the `AuthenticateModel`, validated and passed to the method.

ASP.NET Core Data Annotations are used to automatically handle model validation, the `[Required]` attribute sets both the username and password as required fields so if either are missing a validation error message is returned from the api.

```
using System.ComponentModel.DataAnnotations;

namespace WebApi.Models
{
    public class AuthenticateModel
    {
        [Required]
        public string Username { get; set; }

        [Required]
        public string Password { get; set; }
    }
}
```

Back to top

ASP.NET Core JWT User Service

Path: /Services/UserService.cs

The user service contains a method for authenticating user credentials and returning a JWT token, and a method for getting all users in the application.

I hardcoded the array of users in the example to keep it focused on JWT authentication, in a production application it is recommended to store user records in a database with hashed passwords. For an extended example that includes support for user registration and stores data with Entity Framework Core check out ASP.NET Core 3.1 - Simple API for Authentication, Registration and User Management.

The top of the file contains an interface that defines the user service, below that is the concrete user service class that implements the interface.

On successful authentication the Authenticate method generates a JWT (JSON Web Token) using the `JwtSecurityTokenHandler` class which generates a token that is digitally signed using a secret key stored in appsettings.json. The JWT token is returned to the client application which must include it in the HTTP Authorization header of subsequent requests to secure routes.

```
using System;
using System.Collections.Generic;
using System.IdentityModel.Tokens.Jwt;
using System.Linq;
using System.Security.Claims;
using System.Text;
using Microsoft.Extensions.Options;
using Microsoft.IdentityModel.Tokens;
using WebApi.Entities;
using WebApi.Helpers;

namespace WebApi.Services
{
    public interface IUserService
    {
        User Authenticate(string username, string password);
        IEnumerable<User> GetAll();
    }

    public class UserService : IUserService
    {
        // users hardcoded for simplicity, store in a db with hashed passwords in production applications
        private List<User> _users = new List<User>
        {
            new User { Id = 1, FirstName = "Test", LastName = "User", Username = "test", Password = "test" }
        };

        private readonly AppSettings _appSettings;

        public UserService(IOptions<AppSettings> appSettings)
        {
            _appSettings = appSettings.Value;
        }

        public User Authenticate(string username, string password)
        {
            var user = _users.SingleOrDefault(x => x.Username == username && x.Password == password);

            // return null if user not found
            if (user == null)
                return null;

            // authentication successful so generate jwt token
            var tokenHandler = new JwtSecurityTokenHandler();
            var key = Encoding.ASCII.GetBytes(_appSettings.Secret);
            var tokenDescriptor = new SecurityTokenDescriptor
            {
                Subject = new ClaimsIdentity(new Claim[]
                {
                    new Claim(ClaimTypes.Name, user.Id.ToString())
                }),
                Expires = DateTime.UtcNow.AddDays(7),
                SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.Hmac
            };
            var token = tokenHandler.CreateToken(tokenDescriptor);
            user.Token = tokenHandler.WriteToken(token);

            return user;
        }

        public IEnumerable<User> GetAll()
        {
            return _users;
        }
    }
}
```

Back to top

ASP.NET Core JWT App Settings (Development)

Path: /appsettings.Development.json
Configuration file with application settings that are specific to the development environment.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

Back to top

ASP.NET Core JWT App Settings

Path: /appsettings.json
Root configuration file containing application settings for all environments.

IMPORTANT: The `"Secret"` property is used by the api to sign and verify JWT tokens for authentication, update it with your own random string to ensure nobody else can generate a JWT to gain unauthorised access to your application.

```
{
  "AppSettings": {
    "Secret": "THIS IS USED TO SIGN AND VERIFY JWT TOKENS, REPLACE IT WITH YOUR OWN SECRET, IT CAN BE ANY STRING"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Back to top

ASP.NET Core JWT Program

Path: /Program.cs
The program class is a console app that is the main entry point to start the application, it configures and launches the web api host and web server using an instance of `IHostBuilder`. ASP.NET Core applications require a host in which to execute.

Kestrel is the web server used in the example, it's a new cross-platform web server for ASP.NET Core that's included in new project templates by default. Kestrel is fine to use on it's own for internal applications and development, but for public facing websites and applications it should sit behind a more mature reverse proxy server (IIS, Apache, Nginx etc) that will receive HTTP requests from the internet and forward them to Kestrel after initial handling and security checks.

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;

namespace WebApi
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>()
                        .UseUrls("http://localhost:4000");
                });
    }
}
```

Back to top

ASP.NET Core JWT Startup

Path: /Startup.cs

The startup class configures the request pipeline of the application and how all requests are handled.

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

namespace WebApi
{
    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 services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddCors();
            services.AddControllers();

            // configure strongly typed settings objects
            var appSettingsSection = Configuration.GetSection("AppSettings");
            services.Configure<AppSettings>(appSettingsSection);

            // configure jwt authentication
            var appSettings = appSettingsSection.Get<AppSettings>();
            var key = Encoding.ASCII.GetBytes(appSettings.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
                };
            });

            // configure DI for application services
            services.AddScoped<IUserService, UserService>();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            app.UseRouting();

            // global cors policy
            app.UseCors(x => x
                .AllowAnyOrigin()
                .AllowAnyMethod()
                .AllowAnyHeader());
```

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

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

Back to top

ASP.NET Core JWT Web Api csproj

Path: /WebApi.csproj

The csproj (C# project) is an MSBuild based file that contains target framework and NuGet package dependency information for the application.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="3.1.0" />
    <PackageReference Include="System.IdentityModel.Tokens.Jwt" Version="5.6.0" />
  </ItemGroup>
</Project>
```

Back to top

Subscribe or Follow Me For Updates

Subscribe to my YouTube channel or follow me on Twitter or GitHub to be notified when I post new content.

- Subscribe on YouTube at https://www.youtube.com/channel/UCc46Wo9z8S3xSDhw9vdvxtg/
- Follow me on Twitter at https://twitter.com/jason_watmore
- Follow me on GitHub at https://github.com/cornflourblue
- Feeds formats available: RSS, Atom, JSON

Tags: ASP.NET Core, C#, Authentication and Authorization, Security, JWT

Share:

More ASP.NET Core Posts

- Vue.js + ASP.NET Core on Azure with SQL Server - How to Deploy a Full Stack App to Microsoft Azure
- React + ASP.NET Core on Azure with SQL Server - How to Deploy a Full Stack App to Microsoft Azure
- Angular + .NET Core + SQL on Azure - How to Deploy a Full Stack App to Microsoft Azure
- ASP.NET Core - EF Core Migrations for Multiple Databases (SQLite and SQL Server)
- ASP.NET Core - Automatic EF Core Migrations to SQL Database on Startup
- ASP.NET Core 3.1 - Basic Authentication Tutorial with Example API
- ASP.NET Core 3.1 - Role Based Authorization Tutorial with Example API
- ASP.NET Core 3.1 - Simple API for Authentication, Registration and User Management
- ASP.NET Core 2.2 - Role Based Authorization Tutorial with Example API
- C# - Pure Pagination Logic in C# / ASP.NET
- ASP.NET Core Razor Pages - Pagination Example
- ASP.NET Core 2.2 - Basic Authentication Tutorial with Example API
- ASP.NET Core 2.2 - JWT Authentication Tutorial with Example API
- ASP.NET Core 2.2 - Simple API for Authentication, Registration and User Management