# JWT Authentication with Role-Based Authorization in ASP.NET Core Web API

## 1. Introduction

Authentication is the process of verifying the identity of a user, while Authorization determines what an authenticated user is allowed to do.  
  
- Authentication: Confirms who the user is.  
- Authorization: Confirms what the user is allowed to do.  
  
JWT (JSON Web Token) is a popular authentication mechanism in ASP.NET Core Web API that allows secure user authentication and role-based authorization.

## 2. Step-by-Step Guide

### Step 1: Create an ASP.NET Core Web API Project

Run the following command in the terminal to create a new project:

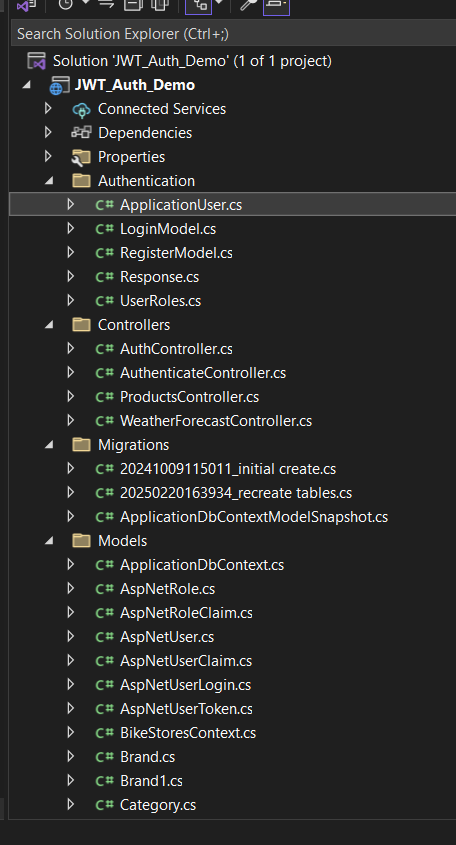
dotnet new webapi -n JwtAuthApi  
cd JwtAuthApi

### Step 2: Install Required NuGet Packages

Run the following command to install necessary dependencies:

dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore  
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer  
dotnet add package Microsoft.EntityFrameworkCore.SqlServer  
dotnet add package Microsoft.EntityFrameworkCore.Tools  
dotnet add package Microsoft.IdentityModel.Tokens  
dotnet add package System.IdentityModel.Tokens.Jwt

**Create Authentication Folder**

****

ApplicationUser.cs

public class ApplicationUser:IdentityUser

{

### }

Explanation

he class ApplicationUser that inherits from IdentityUser is used to **extend** the default ASP.NET Core Identity user model.

**Why Use ApplicationUser Instead of IdentityUser?**

The default IdentityUser class provides basic user-related properties, such as:

* UserName
* Email
* PhoneNumber
* PasswordHash
* SecurityStamp
* TwoFactorEnabled
* and more...

However, if you need to store additional user-related fields in your application (such as First Name, Last Name, Address, etc.), you **must** create a custom ApplicationUser class that inherits from IdentityUser.

**Example Usage: Extending IdentityUser**

public class ApplicationUser : IdentityUser

{

public string FirstName { get; set; }

public string LastName { get; set; }

public DateTime DateOfBirth { get; set; }

}

**Updating the ApplicationDbContext**

To use the custom ApplicationUser instead of IdentityUser, update the ApplicationDbContext:

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)

{

}

}

RegisterModel.cs

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; }

public string Role { get;set; }

}

LoginModel.cs

public class LoginModel

{

[Required(ErrorMessage = "User Name is required")]

public string Username { get; set; }

[Required(ErrorMessage = "Password is required")]

public string Password { get; set; }

}

Response.cs

public class Response

{

public string Status { get; set; }

public string Message { get; set; }

}

UserRoles.cs

public static class UserRoles

{

public const string Admin = "Admin";

public const string User = "User";

}

ApplicationDbContext.cs

### Step 3: Configure Database and Identity

Create a new class `ApplicationDbContext.cs` inside the `Data` folder:

using Microsoft.AspNetCore.Identity;  
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;  
using Microsoft.EntityFrameworkCore;  
  
public class ApplicationDbContext:IdentityDbContext<ApplicationUser>

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options) { }

}

This will create the necessary tables, including ASP.NET Identity tables (AspNetUsers, AspNetRoles, etc.) and your business entities.

### Step 4: Configure appsettings.json for JWT Settings

Add the following to `appsettings.json`:

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"Jwt": {

"Key": "MySuperSecretKey12345AlgorithamsRemovedinProduction",

"Issuer": "yourdomain.com",

"Audience": "yourdomain.com"

},

"AllowedHosts": "\*",

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=Auth\_DB\_TestYantra;Trusted\_Connection=true;TrustServerCertificate=true;"

}

}

### Step 5: Configure Identity and JWT Authentication

Modify `Program.cs`:

using JWT\_Auth\_Demo.Authentication;

using JWT\_Auth\_Demo.Models;

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.AspNetCore.Identity;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Configuration;

using Microsoft.IdentityModel.Tokens;

using Microsoft.OpenApi.Models;

using System.Security.Cryptography.Xml;

using System.Text;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

{

builder.Services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(

builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddDbContext<BikeStoresContext>();

// For Identity

builder.Services.AddIdentity<ApplicationUser, IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultTokenProviders();

// Adding Authentication

builder.Services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;

})

// Adding Jwt Bearer

.AddJwtBearer(options =>

{

options.SaveToken = true;

options.RequireHttpsMetadata = false;

options.TokenValidationParameters = new TokenValidationParameters()

{

ValidateIssuer = true,

ValidateAudience = true,

ValidAudience = builder.Configuration["JWT:Audience"],

ValidIssuer = builder.Configuration["JWT:Issuer"],

IssuerSigningKey = new SymmetricSecurityKey

(Encoding.UTF8.GetBytes(builder.Configuration["JWT:Key"]))

};

});

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API", Version = "v1" });

c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme

{

In = ParameterLocation.Header,

Description = "please Enter Token",

Name = "Authorization",

Type = SecuritySchemeType.Http,

BearerFormat = "JWT",

Scheme = "bearer"

});

c.AddSecurityRequirement(new OpenApiSecurityRequirement

{

{

new OpenApiSecurityScheme

{

Reference = new OpenApiReference

{

Type = ReferenceType.SecurityScheme,

Id = "Bearer"

}

},

new string[] { }

}

});

});

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthentication();

app.UseAuthorization();

app.MapControllers();

app.Run();

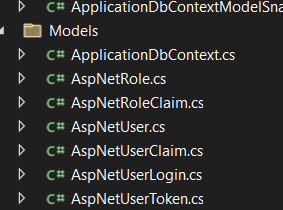
}

Nuget PackageManager Console

Add-Migration ‘tablecreation’

Update-database

Check the the database tables created or not .if created successfully proceed further



### Step 6: Create Authentication Controller

Create a new ` AuthenticateController.cs` inside the `Controllers` folder:  
using JWT\_Auth\_Demo.Authentication;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Mvc;

using Microsoft.IdentityModel.Tokens;

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using System.Text;

namespace JWT\_Auth\_Demo.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, RoleManager<IdentityRole> roleManager, IConfiguration configuration)

{

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, model.Password))

{

var userRoles = await userManager.GetRolesAsync(user);

var authClaims = new List<Claim>

{

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

new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),

};

foreach (var userRole in userRoles)

{

authClaims.Add(new Claim(ClaimTypes.Role, userRole));

}

var authSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_configuration["JWT:Key"]));

var token = new JwtSecurityToken(

issuer: \_configuration["JWT:Issuer"],

audience: \_configuration["JWT:Audience"],

expires: DateTime.Now.AddHours(3),

claims: authClaims,

signingCredentials: new SigningCredentials(authSigningKey,

SecurityAlgorithms.HmacSha256)

);

return Ok(new

{

token = new JwtSecurityTokenHandler().WriteToken(token),

expiration = token.ValidTo

});

}

return Unauthorized();

}

[HttpPost]

[Route("register")]

public async Task<IActionResult> RegisterAdmin([FromBody] RegisterModel model)

{

var userExists = await userManager.FindByNameAsync(model.Username);

if (userExists != null)

return StatusCode(StatusCodes.Status500InternalServerError,

new Response { Status = "Error", Message = "User already exists!" });

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 Response

{

Status = "Error",

Message = "User creation failed! Please check user details and try again."

});

if (model.Role.ToLower() == "user")

{

if (!await roleManager.RoleExistsAsync(UserRoles.User))

await roleManager.CreateAsync(new IdentityRole(UserRoles.User));

if (await roleManager.RoleExistsAsync(UserRoles.User))

{

await userManager.AddToRoleAsync(user, UserRoles.User);

}

}

if (model.Role.ToLower() == "admin")

{

if (!await roleManager.RoleExistsAsync(UserRoles.Admin))

await roleManager.CreateAsync(new IdentityRole(UserRoles.Admin));

if (await roleManager.RoleExistsAsync(UserRoles.Admin))

await userManager.AddToRoleAsync(user, UserRoles.Admin);

}

return Ok(new Response { Status = "Success", Message = "User created successfully!" });

}

}

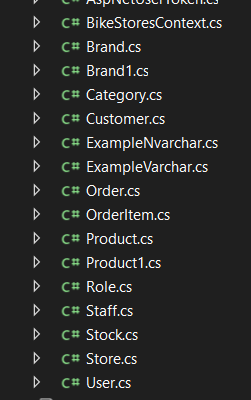
}

Let us add DBFirstApproach as well in the same project to create Controllers for Product,Customers,…

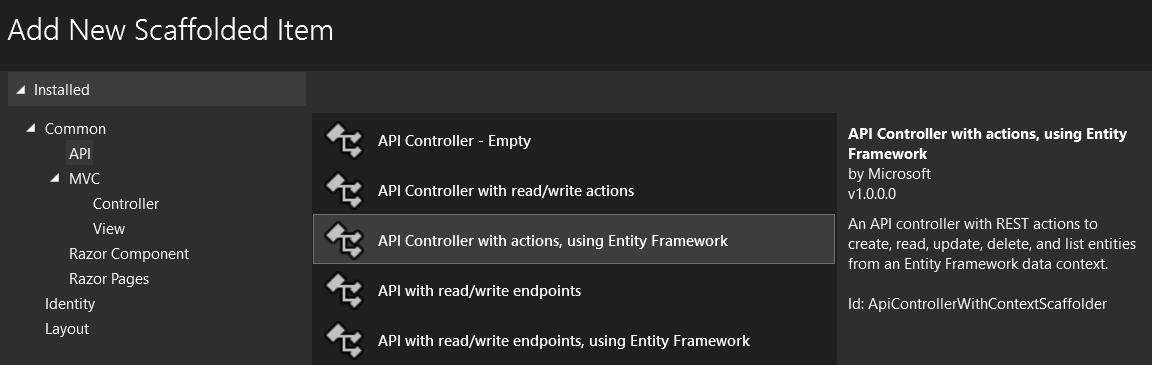
Package Manager Console

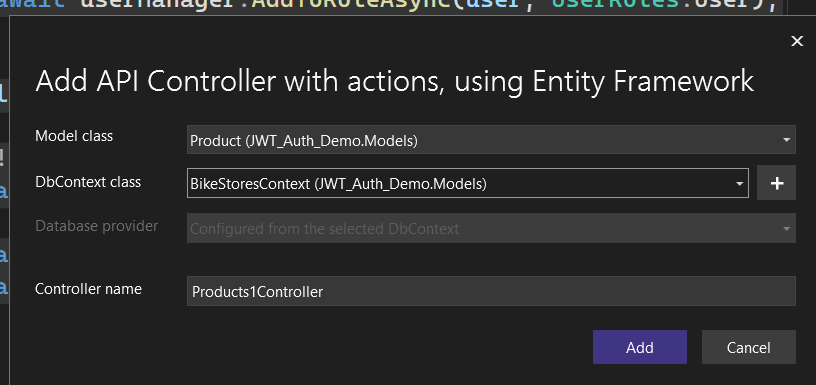
Scaffold-DbContext "Server=(localdb)\MSSQLLocalDB;Database=BikeStores;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models

It will create below model from BikeStores Db



Right click on controllers Add new Controller using MVC EntityFramework (API)





ProductsController.cs

Add Security using Authorize and Roles as below

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using JWT\_Auth\_Demo.Models;

using Microsoft.AspNetCore.Authorization;

namespace JWT\_Auth\_Demo.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class ProductsController : ControllerBase

{

private readonly BikeStoresContext \_context;

public ProductsController(BikeStoresContext context)

{

\_context = context;

}

[Authorize(Roles ="User,Admin")]

// GET: api/Products

[HttpGet]

public async Task<ActionResult<IEnumerable<Product>>> GetProducts()

{

return await \_context.Products.ToListAsync();

}

// GET: api/Products/5

[HttpGet("{id}")]

public async Task<ActionResult<Product>> GetProduct(int id)

{

var product = await \_context.Products.FindAsync(id);

if (product == null)

{

return NotFound();

}

return product;

}

// PUT: api/Products/5

// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754

[HttpPut("{id}")]

public async Task<IActionResult> PutProduct(int id, Product product)

{

if (id != product.ProductId)

{

return BadRequest();

}

\_context.Entry(product).State = EntityState.Modified;

try

{

await \_context.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

if (!ProductExists(id))

{

return NotFound();

}

else

{

throw;

}

}

return NoContent();

}

[Authorize(Roles="Admin")]

// POST: api/Products

// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754

[HttpPost]

public async Task<ActionResult<Product>> PostProduct(Product product)

{

\_context.Products.Add(product);

try

{

await \_context.SaveChangesAsync();

}

catch (DbUpdateException)

{

if (ProductExists(product.ProductId))

{

return Conflict();

}

else

{

throw;

}

}

return CreatedAtAction("GetProduct", new { id = product.ProductId }, product);

}

// DELETE: api/Products/5

[HttpDelete("{id}")]

public async Task<IActionResult> DeleteProduct(int id)

{

var product = await \_context.Products.FindAsync(id);

if (product == null)

{

return NotFound();

}

\_context.Products.Remove(product);

await \_context.SaveChangesAsync();

return NoContent();

}

private bool ProductExists(int id)

{

return \_context.Products.Any(e => e.ProductId == id);

}

}

}