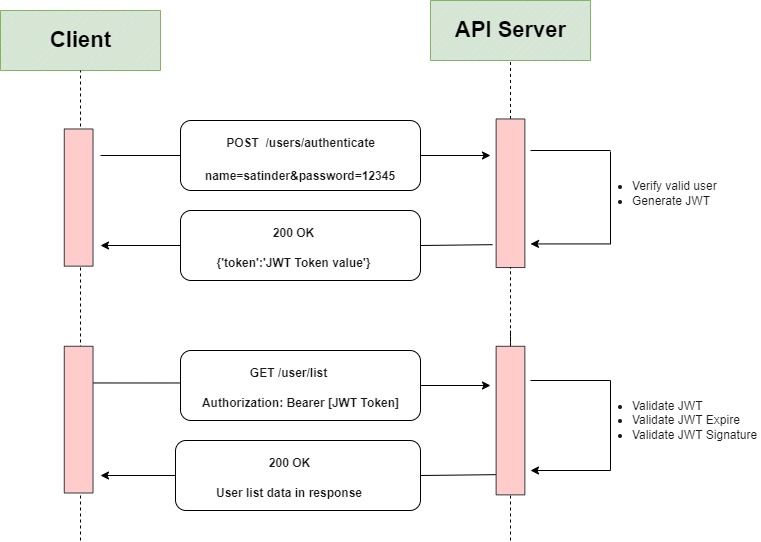
# Authentication and Authorization in ASP.NET Web API:

* Authentication is knowing the identity of the user. For example, Alice logs in with her username and password, and the server uses the password to authenticate Alice.
* Authorization is deciding whether a user is allowed to perform an action. For example, Alice has permission to get a resource but not create a resource.

# JWT in ASP.NET Core

[JWT (JSON web token)](https://www.c-sharpcorner.com/article/introduction-to-jwt/)has become more and more popular in web development. It is an open standard which allows transmitting data between parties as a JSON object in a secure and compact way. The data transmitting using JWT between parties are digitally signed so that it can be easily verified and trusted.



**What is the JSON Web Token structure?**

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.

xxxxx.yyyyy.zzzzz

Let's break down the different parts.

**Header**

The header *typically* consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

For example:

{

"alg": "HS256",

"typ": "JWT"

}

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

**Payload**

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: *registered*, *public*, and *private* claims.

* [**Registered claims**](https://tools.ietf.org/html/rfc7519#section-4.1): These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: **iss** (issuer), **exp** (expiration time), **sub** (subject), **aud** (audience), and [others](https://tools.ietf.org/html/rfc7519#section-4.1).
* [**Public claims**](https://tools.ietf.org/html/rfc7519#section-4.2): These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the [IANA JSON Web Token Registry](https://www.iana.org/assignments/jwt/jwt.xhtml) or be defined as a URI that contains a collision resistant namespace.
* [**Private claims**](https://tools.ietf.org/html/rfc7519#section-4.3): These are the custom claims created to share information between parties that agree on using them and are neither *registered* or *public* claims.

An example payload could be:

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

The payload is then **Base64Url** encoded to form the second part of the JSON Web Token.

Do note that for signed tokens this information, though protected against tampering, is readable by anyone. Do not put secret information in the payload or header elements of a JWT unless it is encrypted.

**Signature**

To create the signature part, you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

HMACSHA256(

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret)

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

1. Install Microsoft.AspNetCore.Authentication.JwtBearer on nuget compatible with .net 6.
2. Create User Model With user data:

public class UserModel

{

public string? DisplayName { get; set; }

public string? UserName { get; set; }

public string? Email { get; set; }

public string? Password { get; set; }

public DateTime? CreatedDate { get; set; }

}

Create Login Controller: Create a LoginController and Login method which is responsible to generate the JWT.

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

[ApiController]

public class UsersController : ControllerBase

{

private IConfiguration \_configuration;

public UsersController(IConfiguration options)

{

\_configuration = options;

}

// GET: api/<UsersController>

[HttpGet]

public IEnumerable<string> Get()

{

return new string[] { "value1", "value2" };

}

// GET api/<UsersController>/5

[HttpGet("{id}")]

public string Get(int id)

{

return "value";

}

// POST api/<UsersController>

[HttpPost("login")]

public IActionResult Login([FromBody] UserModel user)

{

if (user != null && user.Name !=null && user.Password !=null)

{

var token = GenerateJwtToken(user.Name);

return Ok(new { token });

}

return Unauthorized();

}

private string GenerateJwtToken(string username)

{

var claims = new[]

{

new Claim(JwtRegisteredClaimNames.Sub, username),

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

};

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

var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

var token = new JwtSecurityToken(

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

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

claims: claims,

expires: DateTime.Now.AddMinutes(30),

signingCredentials: creds);

return new JwtSecurityTokenHandler().WriteToken(token);

}

Set service in the program::

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

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

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

builder.Services.Configure<UnitOptions>(builder.Configuration.GetSection("Units"));

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseLazyLoadingProxies().UseSqlServer(builder.Configuration.GetConnectionString("DbConnection")));

builder.Services.AddScoped<IBlogRepository, BlogRepository>();

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

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

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

IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))

};

});

builder.Services.AddAuthorization();

builder.Services.AddSwaggerGen(c => {

c.SwaggerDoc("v1", new OpenApiInfo

{

Title = "JWTToken\_Auth\_API",

Version = "v1"

});

c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme()

{

Name = "Authorization",

Type = SecuritySchemeType.ApiKey,

Scheme = "Bearer",

BearerFormat = "JWT",

In = ParameterLocation.Header,

Description = "JWT Authorization header using the Bearer scheme. \r\n\r\n Enter 'Bearer' [space] and then your token in the text input below.\r\n\r\nExample: \"Bearer 1safsfsdfdfd\"",

});

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

In the above code, we configured authorization middleware in the startup. Here we have passed the security key when creating the token and enabled validation of Issuer and Audience. Also, we have set “SaveToken” to true, which stores the bearer token in HTTP Context. So we can use the token later in the controller.

Now open “**appsetting.json**” and add the below code to the file

"Jwt": {

"Key": "Yh2k7QSu4l8CZg5p6X3Pna9L0Miy4D3Bvt0JVr87UcOj69Kqw5R2Nmf4FWs03Hdx",

"Issuer": "JWTAuthenticationServer",

"Audience": "JWTServicePostmanClient",

"Subject": "JWTServiceAccessToken"

}

Now we add the authorization attribute to the controller, so all the APIs under this controller will be secured with the token.

[Authorize]

marking a method with the [AllowAnonymous] attribute will bypass the authentication.

Use Authorization in swagger:

Add Authorization button to swagger ui:

builder.Services.AddSwaggerGen(c => {

c.SwaggerDoc("v1", new OpenApiInfo

{

Title = "JWTToken\_Auth\_API",

Version = "v1"

});

c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme()

{

Name = "Authorization",

Type = SecuritySchemeType.ApiKey,

Scheme = "Bearer",

BearerFormat = "JWT",

In = ParameterLocation.Header,

Description = "JWT Authorization header using the Bearer scheme. \r\n\r\n Enter 'Bearer' [space] and then your token in the text input below.\r\n\r\nExample: \"Bearer 1safsfsdfdfd\"",

});

c.AddSecurityRequirement(new OpenApiSecurityRequirement {

{

new OpenApiSecurityScheme {

Reference = new OpenApiReference {

Type = ReferenceType.SecurityScheme,

Id = "Bearer"

}

},

new string[] {}

}

});

});

Call Authorization method with existing user

Copy returned bearer

Go to Authorize button and paste it there