



Lecturer

Rokas Slaboševičius

Json Web Token (JWT)

Data



Today you will learn

01

What is JWT?



What is JWT?

Technical definition:

What is JSON Web Token?

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**.



What is JWT?

In simple terms, a JWT is a three-part string containing information about the client, which is used by the server to decide what functionality the client can and cannot use.

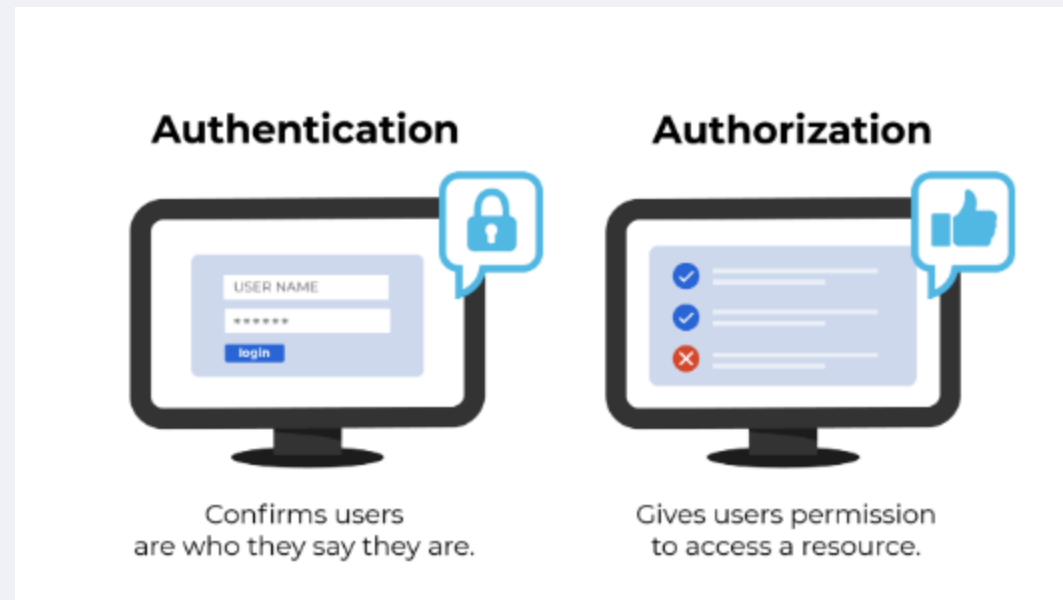
For example: a simple User type user can view photos on the website, while an Admin type user can edit, delete and add new ones.



Terminology

Authentication - Making sure a user is who they say they are.

Authorisation - granting rights depending on the type of user (usually the next step after authentication).





Terminology

	Authentication	Authorisation
What does it do?	Verifies credentials	Gives/takes away rights
How does it work?	Passwords, biometrics, one-time pins or other apps	System configurations
Is this visible to the customer?	Yes	No
Is it controlled by client?	Partly	No



Jwt structure

Jwt consists of three parts separated by the dot `.`:

1. Header
2. Payload
3. Signature

It therefore looks like this:

xxxxx.yyyy.zzzzz



Header

The header contains information about the type of token (in this case it is always JWT) and the algorithm used to encrypt the token, e.g. HMAC SHA256 or RSA.

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```




Payload

Claims are stored in the Payload section.

Claims are the properties of the client from whom the request was received.

Claims are divided into three types:

1. Registered
2. Public
3. Private



Claims

What do you mean by interoperability?

Interoperability (pronounced IHN- tuhr -AHP- uhr -uh-BIHL- ih -tee) is the ability of different systems, devices, applications or products to connect and communicate in a coordinated way, without effort from the end user.

1. Registered

- a. Registered claims are optional but predefined interoperable information such as **iss**(issuer), **exp**(expiration time), **sub**(subject), **aud**(audience) etc.

2. Public

- a. These claims can be created by programmers, but care must be taken with the names so as not to spoil the structure of the JSON structure. You can read more about creating public claims here [IANA JSON Web Token Registry](#)

3. Private

- a. Claims that are used by both client and server (by mutual agreement) but are neither public nor registered are called Private



Payload example

PAYLOAD: DATA

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}
```



Signature

To create the signature part, you need to take the encoded header, payload, **secret** (a randomly generated long string stored in a location invisible to the client), the algorithm mentioned in the header and **sign** everything.

If we used the HMAC SHA 256 algorithm, it would look similar:

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  secret)
```



Signature

The Signature part is used to ensure the security of the information and to make sure that jwt has not been modified.

Why is this necessary?

The JWT is usually stored in cookies, which can be seen by any user in their browser and they can modify the JWT value without being disturbed, but if the server sees the signature part, it will realise that the token is not the original one and will reject the requests.



Jwt structure

This is what the generated JWT looks like

You can experiment more for yourself at www.jwt.io

Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

Signature Verified

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  "alg": "HS256",  "typ": "JWT"}
```

PAYLOAD: DATA

```
{  "sub": "1234567890",  "name": "John Doe",  "iat": 1516239822}
```

VERIFY SIGNATURE

```
HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),  your-256-bit-secret)
```




☐ secret base64 encoded

SHARE JWT



Jwt Asp net core API

Packages:

	Microsoft.AspNetCore.Authentication.JwtBearer  by Microsoft, 216M downloads	5.0.17
	ASP.NET Core middleware that enables an application to receive an OpenID Connect bearer token.	6.0.6



Creating login functionality

First step we create a model

```
public class Account
{
    public Guid Id { get; set; }
    public string Username { get; set; } = string.Empty;
    public byte[] PasswordHash { get; set; }
    public byte[] PasswordSalt { get; set; }
}
```




Creating login functionality

Step 2: Create
registration functionality

```
public Account SignupNewAccount(string username, string password)
{
    var account = CreateAccount(username, password);
    _applicationDbContext.SaveAccount(account);
    return account;
}

private Account CreateAccount(string username, string password)
{
    CreatePasswordHash(password, out byte[] passwordHash, out byte[] passwordSalt);
    var account = new Account
    {
        Username = username,
        PasswordHash = passwordHash,
        PasswordSalt = passwordSalt
    };

    return account;
}

private void CreatePasswordHash(string password, out byte[] passwordHash, out byte[] passwordSalt)
{
    using var hmac = new HMACSHA512();
    passwordSalt = hmac.Key;
    passwordHash = hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));
}
```



Creating login functionality

Step 3: Create
login functionality

```
public bool Login(string username, string password)
{
    var account = _applicationDbContext.GetAccount(username);
    if (VerifyPasswordHash(password, account.PasswordHash, account.PasswordSalt))
        return true;

    return false;
}

private bool VerifyPasswordHash(string password, byte[] passwordHash, byte[] passwordSalt)
{
    using var hmac = new HMACSHA512(passwordSalt);
    var computedHash = hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));

    return computedHash.SequenceEqual(passwordHash);
}
```



Creating Jwt functionality

We start with the appsettings.development.json configurations

```
"Jwt": {  
  "Key": "7e070bdece42bd55243ea423aa9d024ae655d611bc8fab598fb543e5f3545224133bf7befc9112dab34c4535d50c107551b580660c353c83476cd1e4f87ccaed",  
  "Issuer": "https://localhost:44338/",  
  "Audience": "https://localhost:44338/"  
}
```



Creating Jwt functionality

We can sign up for the Jwt service

```
public class JwtService : IJwtService
{
    private readonly IConfiguration _configuration;

    public JwtService(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    public string GetJwtToken(string username)
    {
        List<Claim> claims = new List<Claim>
        {
            new Claim(ClaimTypes.Name, username)
        };

        var secretToken = _configuration.GetSection("Jwt:Key").Value;
        var key = new SymmetricSecurityKey(System.Text.Encoding.UTF8.GetBytes(secretToken));

        var cred = new SigningCredentials(key, SecurityAlgorithms.HmacSha512Signature);

        var token = new JwtSecurityToken(
            issuer: "https://localhost:44338/",
            audience: "https://localhost:44338/",
            claims: claims,
            expires: DateTime.Now.AddDays(1),
            signingCredentials: cred);

        return new JwtSecurityTokenHandler().WriteToken(token);
    }
}
```



Creating Jwt functionality

Settings in Startup.cs configurations

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = Configuration["Jwt:Issuer"],
            ValidAudience = Configuration["Jwt:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(Configuration["Jwt:Key"]))
        };
    });
```



Building Jwt functionality

Settings in Startup.cs configurations

add a Swagger login page

```
services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo { Title = "UnitTestPracticeApplication", Version = "v1" });
    options.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        In = ParameterLocation.Header,
        Description = "Please enter a valid token",
        Name = "Authorization",
        Type = SecuritySchemeType.Http,
        BearerFormat = "JWT",
        Scheme = "Bearer"
    });
    options.AddSecurityRequirement(new OpenApiSecurityRequirement
    {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference
                {
                    Type=ReferenceType.SecurityScheme,
                    Id="Bearer"
                }
            },
            new string[]{}
        }
    });
});
```



Creating Jwt functionality

Now we have an Authorize button where we can enter the JWT token to access the closed endpoints

Demo API v1 OAS3
<https://localhost:7029/swagger/v1/swagger.json>

Authentication

POST	/api/Authentication/signup	✓	🔒
POST	/api/Authentication/login	✓	🔒
GET	/api/Authentication	✓	🔒



Task 1

- Connect login/signup with JWT functionality to the third .NET lecture API