

Table of Contents

1. Introduction.....	2
2. Library Structure.....	2
3. Model Requirements and Generic Structure	3
4. Installation and Configuration.....	4
5. Usage Scenarios.....	5
6. Interface Methods — Brief Explanations	6
7. Conclusion	7

1. Introduction

I realized that I repeat the same things every time in my projects. JWT services, interface definitions, cookie settings, refresh token management... Instead of starting with "copy-pasting" on each new project, I wanted to bring it all under one simple, organized, and extensible structure.

Yunus.JwtKit was born out of this need. Standard JWT solutions are generally limited to Access Token generation. However, a true identity verification process is not just about generating tokens.

JwtKit offers both a *secure* and *flexible* authentication infrastructure thanks to Refresh Token management, cookie-based session support and ASP.NET Identity compatibility.

The library is built with Clean Code and Plug & Play: You don't need to write services or interfaces from scratch for every project.

The goal is for the developer to be able to easily run the program flow without having to deal with setting up a token system from scratch .

Supported environments:

- Local (Swagger, Postman)
- Cross-Origin (React, Next.js, Angular)
- Production (HTTPS, HttpOnly, Secure cookie)

Properties:

- Access and Refresh token generation
- Cookie or Header based token migration
- Cookie security against XSS and CSRF
- ASP.NET Identity compatibility (but not mandatory)
- Plug & Play installation: active in 5 minutes

2. Library Structure

The library consists of 2 main components:

a) Interface – `IJwtTokenService<TUser, TUserInfo>`

It defines all basic methods such as token generation, renewal, cookie transactions. This interface presents the "contract" to the outside. It reduces project dependency.

b) Service – `JwtService<TUser, TUserInfo>`

It is the implementation of the Interface.

It generates tokens, verifies them, records refresh tokens, and manages cookie transactions.

It works completely dynamically through AppSettings and UserManager structure.

Generic types (TUser, TUserInfo) are the strongest part of the package.

3. Model Requirements and Generic Structure

JwtService requests two generic types:

1- TUser

It is the user model.

To hold the refresh token and its duration, you must implement the following interface:

```
public interface IHasRefreshToken
{
    string? RefreshToken { get; set; }
    DateTime? RefreshTokenExpiryTime { get; set; }
}
```

◆ If you're using ASP.NET Identity:

```
public class AppUser : IdentityUser, IHasRefreshToken
{
    public string? RefreshToken { get; set; }
    public DateTime? RefreshTokenExpiryTime { get; set; }

    public string? FirstName { get; set; }
    public string? LastName { get; set; }
}
```

◆ If you are not using Identity:

```
public class User : IHasRefreshToken
{
    public int Id { get; set; }
    public string Email { get; set; } = string.Empty;
    public string PasswordHash { get; set; } = string.Empty;

    public string? RefreshToken { get; set; }
    public DateTime? RefreshTokenExpiryTime { get; set; }
}
```

2-TUserInfo

The user information (DTO) that will be returned in the token response. This means that whatever information you want to send outside the API contains only them.

```
public class JwtResponse
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public string? Email { get; set; }
}
```

Here's what the token response looks like:

```
{
    "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJvYWIjoiVGVzdC1sInN1YiI6IjEyMzQ1Njc4OTIa",
    "refreshToken": "U9xZp4mKk5vT3sF1p2rQ9ls4jM7bH6dP3tG8qV1xE5yN0sC2dJ5oF0aG8uW6rQ2l=",
    "tokenType": "Bearer",
    "expiresIn": 3600,
    "issuedAt": "2025-11-07T13:00:00Z",
    "user": {
        "firstName": "Test",
        "lastName": "User",
        "email": "test@example.com"
    }
}
```

4. Installation and Configuration

Setup via NuGet

```
dotnet add package Yunus.JwtKit
```

appsettings.json

```
"JwtSettings": {
    "SecretKey": "your_super_secret_key",
    "Issuer": "your_app",
    "Audience": "your_users",
    "AccessTokenExpiryMinutes": 60,
    "RefreshTokenExpiryDays": 7
}
```

Program.cs – Service Registration

```

builder.Services.AddScoped<IJwtTokenService<AppUser, JwtResponse>, JwtService<AppUser, JwtResponse>>(
    serviceBuilder =>
{
    var service = new JwtService<AppUser, JwtResponse>(
        sp.GetRequiredService<UserManager<AppUser>>(),
        sp.GetRequiredService<ILogger<JwtService<AppUser, JwtResponse>>>(),
        sp.GetRequiredService< IConfiguration>()
    );

    // 🔔 Local (Swagger / Postman) ortamı için (⚙️ Default değerler)
    // CookieHttpOnly = false
    // CookieSecure = false
    // CookieSameSite = SameSiteMode.Lax

    // 🔔 Cross-Origin (React / Next.js) ortamı için (örnek konfigürasyon)
    // service.CookieHttpOnly = true;
    // service.CookieSecure = true;
    // service.CookieSameSite = SameSiteMode.None;

    return service;
}
);

```

5. Usage Scenarios

5.1 Local Development (Swagger/Postman) Cookies are not mandatory.

The token can be moved via body or header.

- HttpOnly = false
- Secure = false
- SameSite = Lax

```
{
    "success": true,
    "data": {
        "accessToken": "eyJhbGciOiJUzTESTaccesstoken123...",
        "refreshToken": "sd9f8sdfTESTrefreshtoken456...",
        "user": {
            "firstName": "Test",
            "lastName": "User",
            "email": "testuser@example.com"
        }
    }
}
```

5.2 Cross-Origin (React/Next.js)

If a cookie-based login is to be opened on the React side:

- HttpOnly = true
- Secure = true
- SameSite = None

In this case, JWT does not appear in the browser (XSS security). Cookies are automatically sent in requests (withCredentials: true).

5.3 Token Sending via Header

An alternative for those who do not want to use cookies.

6. Interface Methods — Brief Explanations

Method	Explanation
Task<string> GenerateAccessTokenAsync(TUser user, Func<TUser, IEnumerable<Claim>>? extraClaims = null)	Generates JWT Access Token belonging to the given user. Optionally, extra claims can be added.
string GenerateRefreshToken()	It randomly generates a 64-byte long Refresh Token (in Base64 format).
ClaimsPrincipal GetPrincipalFromExpiredToken(string token)	It extracts user information (claims) from an expired token. It is used for refresh operations.
Task<bool> ValidateRefreshTokenAsync(TUser user, string refreshToken)	It verifies whether the given refresh token belongs to the user and has not expired.
Task SaveRefreshTokenAsync(TUser user, string refreshToken)	It records the user's refresh token and expiration date in the database.
Task RevokeRefreshTokenAsync(TUser user)	Resets the user's refresh token (used during the logout process).
string GetUserIdFromToken(string token)	Returns the user ID from the NamelIdentifier (or sub) claim within the token.
bool IsTokenValid(string token)	It verifies whether the token is valid (signature, duration, issuer, audience, etc.).
String? GetTokenFromRequest(HttpRequest request)	It searches for the token in the incoming request in the following order: Authorization Header, Cookie, Query Param.
void SetAccessTokenCookie(HttpResponse response, string token, int expiryMinutes)	Writes the Access token to the cookie. Expiry time is minutes. HttpOnly can be used against XSS.
void SetRefreshTokenCookie(HttpResponse response, string refreshToken, int expiryDays)	Writes the refresh token to the cookie. Expiry time is days. Ideal for browser sessions.

String? GetTokenFromCookie(HttpContext request, string cookieName)	Reads the Access or Refresh token from within the cookie.
void ClearTokenCookies(HttpContext response)	Overrides Access and Refresh token cookies (called after logout).
Task<JwtTokenResponse<TUserInfo>> CreateTokenResponseAsync(...)	Combine the Access and Refresh tokens to create a response object (JwtTokenResponse) to be sent to the client.

After these methods, you can use the package you have integrated into your project in systems with all Jwt operations by calling it in your service structure or in the controller.

7. Conclusion and Closing

I wrote this library because I was bored of rebuilding the same JWT structure in every project. My aim was to establish a system that was "simple enough for everyone to understand" but "professional".

Jodolph.JwtKit is more than just a token service—it's a non-repetitive, understandable, and extensible infrastructure philosophy.

Use it, examine it, and if there is anything missing, I am always open to feedback to improve myself.

We hope you use it in your projects.