ASP.NET Core - True Ultimate Guide

Section 29: JWT & Web API Authentication - Notes

JWT Tokens and How They Work Internally

JWT (JSON Web Token) is an open standard (RFC 7519) used for securely transmitting information between two parties as a JSON object. JWT is widely used for authentication and authorization purposes because it is stateless, compact, and easy to verify.

1. Structure of JWT

A JWT token consists of three parts separated by periods:

- **Header**: Contains metadata about the token, including the type of token (JWT) and the hashing algorithm (e.g., HMAC, SHA256).
- **Payload**: Holds the claims, or information, such as user ID and roles. Claims can be:
    - **Registered Claims**: Predefined claims, e.g., iss (issuer), sub (subject), and exp (expiration).
    - **Public Claims**: Custom claims that are not reserved, e.g., user_id.
    - **Private Claims**: Custom claims that are agreed upon between parties but are unique to that transaction.
- **Signature**: Ensures that the token was not tampered with and verifies the authenticity.

Example JWT:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiIxMjM0Iiwicm9sZSI6ImFkbWluIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Each part of the JWT is **Base64Url-encoded**:

1. **Header**: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
2. **Payload**: eyJ1c2VySWQiOiIxMjM0Iiwicm9sZSI6ImFkbWluIiwiaWF0IjoxNTE2MjM5MDIyfQ
3. **Signature**: SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

2. Steps for JWT Token Generation and Validation

1. **Token Creation**:
    - The server creates a token when a user logs in.
    - It encodes the header and payload, and then hashes these using a secret key and the algorithm specified in the header to create the signature.
2. **Token Transmission**:
    - After generation, the server sends the JWT token to the client (usually in the response header).
    - The client stores the token (typically in localStorage or a cookie).
3. **Token Validation**:
    - When a client makes an authenticated request, it sends the JWT in the Authorization header as a Bearer token.
    - The server decodes the token, checks the signature, and validates claims like expiration (exp).
    - If valid, the server processes the request, otherwise, it rejects it.

Example Flow

1. **User Login**:
    - The client sends login credentials to the server.
    - The server authenticates the user and, if successful, generates a JWT.
    - The server sends the token back to the client.
2. **Requesting Protected Resource**:
    - The client includes the JWT in the request header to access a protected endpoint.
    - The server verifies the JWT and, if valid, allows access.

JWT in the Authorization Header

The JWT token is typically sent in the Authorization header:

```
Authorization: Bearer <JWT_TOKEN>
```

Pros of JWTs for Authentication and Authorization

- **Stateless**: No need to store sessions on the server, making it scalable.
- **Compact**: The Base64Url-encoded format makes JWTs lightweight and fast to transmit.
- **Self-contained**: Contains all necessary information about the user, like roles and permissions.
- **Cross-platform**: Compatible with many languages and frameworks.

JWT Algorithm & How Tokens Are Generated

JWTs can be signed and sometimes encrypted to enhance security. The signing process is essential to ensure that the token was not tampered with after it was created. Here's a detailed breakdown of how JWT tokens are generated and the algorithms used:

1. JWT Algorithms

The **algorithm** defines how the token's header and payload will be signed. JWT supports several algorithms, but the most commonly used ones are:

- **HS256 (HMAC with SHA-256)**: Uses a secret key to create a hash of the header and payload. It's symmetric, meaning the same secret key is used for both signing and verifying.
- **RS256 (RSA Signature with SHA-256)**: Uses a public-private key pair for signing and verification. The private key signs the token, and the public key verifies it. This is asymmetric, making it more secure for distributed applications, as you don't need to share the private key.

The **header** of a JWT specifies the algorithm used, such as:

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

2. Generating JWT Tokens

To generate a JWT, the server:

1. **Creates a Header**: Defines the type (JWT) and algorithm (e.g., HS256).
2. **Creates a Payload**: This can include registered claims like iss (issuer), exp (expiration), custom claims such as user_id, role, etc.
3. **Signs the Token**: Based on the algorithm specified in the header, the server signs the header and payload.

Let’s break down each component in the process with an example.

Example: Generating a JWT Using HS256

Let’s consider a payload with a user_id and role.

Payload:

```
{
  "user_id": "12345",
  "role": "admin",
  "exp": 1704067199
}
```

1. **Encoding**: The payload and header are Base64Url-encoded.
2. **Signature Generation**:
    - The header and payload are combined into a single string: header.payload.
    - Using the HS256 algorithm, the server signs the token using a secret key, say my_secret_key.

Signature Creation:

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

The result is a token that looks like:

JhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoiMTIzNDUiLCJyb2xlIjoiYWRtaW4iLCJleHAiOjE3MDQwNjcxOTl9.pxE2T5jzz73YQDd_6YYKTYGFlAlySeyxTWmXaDsX6IM

3. Token Verification

When the client presents this JWT to the server:

1. The server **extracts the header, payload, and signature**.
2. It uses the algorithm defined in the header and the **same secret key** to generate a signature.
3. The server compares this newly generated signature to the one in the JWT.
    - If they match, the token is verified.
    - If they don't match or the token is expired (exp claim), the server rejects it.

Example Code: Generating and Verifying JWT in .NET

Below is an example code snippet that uses System.IdentityModel.Tokens.Jwt to generate and verify JWTs in an ASP.NET Core application.

Generating a JWT:

```
using System;

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using Microsoft.IdentityModel.Tokens;

using System.Text;


public string GenerateJwtToken(string userId, string role, string secretKey)

{

   var claims = new[]

   {

      new Claim(JwtRegisteredClaimNames.Sub, userId),

      new Claim("role", role),

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

   };


   var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey));

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

```
    var token = new JwtSecurityToken(
        issuer: "my_app",
        audience: "my_app",
        claims: claims,
        expires: DateTime.Now.AddMinutes(30),
        signingCredentials: creds);

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

Verifying a JWT:

```
public ClaimsPrincipal ValidateJwtToken(string token, string secretKey)
{
    var tokenHandler = new JwtSecurityTokenHandler();
    var key = Encoding.UTF8.GetBytes(secretKey);
    var validationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(key),
        ValidateIssuer = false,
        ValidateAudience = false
    };

    try
    {
        var principal = tokenHandler.ValidateToken(token, validationParameters, out _);
        return principal;
    }
    catch (Exception)
    {
        return null; // Token is invalid or expired
    }
}
```

In the above example:

- GenerateJwtToken creates a token for a user with a given role.
- ValidateJwtToken verifies the JWT's signature and returns the claims if it's valid, or null if it isn't.

Best Practices and Common Pitfalls of JWT

When working with JWTs, security and efficiency are essential. Here's a guide to the best practices and common pitfalls to avoid:

1. Best Practices

a. Use Strong Secret Keys (for Symmetric Algorithms like HS256)

- Ensure that the secret key used for signing JWTs is long, complex, and stored securely.
- **Avoid using predictable or weak keys**, as this would make it easier for attackers to forge tokens.
- Rotate keys periodically to minimize the risk of token tampering.

b. Use Asymmetric Algorithms (e.g., RS256) for Public Key Verification

- For distributed applications, using asymmetric signing algorithms like RS256 can be more secure.
- In RS256, the private key is used to sign the JWT, and the public key is used to verify it, so you can share the public key with different services without exposing the private key.

c. Limit the Claims in the JWT Payload

- Only include essential information (such as user_id, role, and any custom claims required) in the payload to reduce token size and limit data exposure.
- Avoid storing sensitive data in JWTs, as they are easily readable by anyone who has access to them.

d. Set Expiration Times and Use Short-Lived Tokens

- JWTs should have short expiration times (exp claim) to minimize the window of opportunity for attackers.
- Set a reasonable token lifespan based on the application's security requirements. Typically, tokens are set to expire within minutes or hours for high-security apps.

e. Implement Refresh Tokens for Long-Lived Sessions

- Instead of making JWTs valid for extended periods, use short-lived tokens with refresh tokens to allow the user to obtain a new JWT without re-authenticating.
- Refresh tokens are stored securely on the client side (usually in HTTP-only cookies) and are only exchanged when a new JWT is needed.

f. Store Tokens Securely

- Store tokens in secure, HTTP-only cookies to protect them from client-side JavaScript access, reducing the risk of cross-site scripting (XSS) attacks.
- Avoid storing tokens in localStorage or sessionStorage if possible, as they are vulnerable to XSS attacks.

g. Validate All Claims in the Token

- Verify essential claims like iss (issuer) and aud (audience) to confirm the token was issued by your server and is intended for your application.
- Always validate the exp (expiration) claim to ensure the token hasn’t expired.

h. Monitor for JWT Revocation

- Keep track of tokens that should be invalidated before they expire, such as when a user logs out or when tokens are compromised.
- JWTs are stateless and, by default, cannot be revoked, so implementing a revocation list or managing blacklists in a cache can help mitigate this.

2. Common Pitfalls

a. Overly Long Token Expiration Times

- Avoid issuing tokens with excessively long lifespans, as this increases the risk if a token is compromised. Keep token expiration times short and utilize refresh tokens for long sessions.

b. Including Sensitive Data in the Payload

- Avoid putting sensitive information like passwords, credit card numbers, or private keys in the JWT payload. JWTs can be easily decoded without the secret key, exposing any information inside the payload.

c. Lack of Signature Validation

- Always verify the JWT's signature to ensure it was issued by a trusted source. Skipping this step opens the application to forged tokens and security vulnerabilities.

d. Using Weak Signing Algorithms

- Some algorithms (e.g., none) don't sign the JWT at all, making it easy to modify and reissue. Always use secure signing algorithms like HS256 or RS256.

e. Not Using HTTPS

- When transmitting JWTs over HTTP, the tokens can be intercepted. Always use HTTPS to encrypt token transmission, protecting against man-in-the-middle (MITM) attacks.

3. Example Code: JWT Best Practices in ASP.NET Core

In the following example, we generate and validate JWTs using HS256, set a short expiration time, and demonstrate the importance of HTTPS:

```
using System;

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using Microsoft.IdentityModel.Tokens;

using System.Text;


public class JwtHelper

{

    private readonly string _secretKey;

    private readonly int _expiryMinutes;


    public JwtHelper(string secretKey, int expiryMinutes)
```

```
{
   _secretKey = secretKey;
   _expiryMinutes = expiryMinutes;
}

// Generate a short-lived JWT with minimal claims
public string GenerateToken(string userId, string role)
{
   var claims = new[]
   {
      new Claim(JwtRegisteredClaimNames.Sub, userId),
      new Claim("role", role),
      new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
   };

   var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_secretKey));
   var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

   var token = new JwtSecurityToken(
      issuer: "my_app",
      audience: "my_app",
      claims: claims,
      expires: DateTime.UtcNow.AddMinutes(_expiryMinutes),
      signingCredentials: creds);

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

// Validate the JWT by checking expiration, issuer, and signature
public ClaimsPrincipal ValidateToken(string token)
{
        var tokenHandler = new JwtSecurityTokenHandler();
        var key = Encoding.UTF8.GetBytes(_secretKey);

        var validationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(key),
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidIssuer = "my_app",
            ValidAudience = "my_app",
            ClockSkew = TimeSpan.Zero
        };

        try
        {
            return tokenHandler.ValidateToken(token, validationParameters, out _);
        }
        catch
        {
            return null; // Invalid token
        }
    }
}
```

In this code:

- **GenerateToken** method creates a JWT with a user ID and role, setting a short expiration time (expiryMinutes) and signing it using the HS256 algorithm.
- **ValidateToken** method validates the token, checking the signature, issuer, and audience, which helps enforce claims and ensure secure token handling.

4. Token Revocation Example

You could store revoked tokens in a cache like Redis with an expiration equal to the token’s expiration time, then check each token against this list when validating. Although this approach adds state management, it effectively addresses early revocation requirements.

JWT Authentication and Authorization with JWT in ASP.NET Core Web API

This section provides a step-by-step guide on implementing JWT authentication and authorization in an ASP.NET Core Web API. This integration ensures that users can securely access resources based on their identity and roles.

1. Setting Up JWT Authentication in ASP.NET Core

To enable JWT-based authentication, start by configuring the authentication service within the Startup.cs file (or Program.cs/AppSettings.json in newer versions of ASP.NET Core):

Step 1: Install the Necessary NuGet Packages

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Step 2: Add JWT Settings to appsettings.json

In your appsettings.json file, define the JWT options (such as Issuer, Audience, and SecretKey):

```
"JwtSettings": {
  "SecretKey": "Your_Secret_Key_Here",
  "Issuer": "my_app",
  "Audience": "my_app_audience"
}
```

Step 3: Configure Authentication in Program.cs

Add the JWT authentication scheme in the ConfigureServices method:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;

var builder = WebApplication.CreateBuilder(args);

// Retrieve settings from appsettings.json
var jwtSettings = builder.Configuration.GetSection("JwtSettings");
var secretKey = jwtSettings.GetValue<string>("SecretKey");
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = jwtSettings.GetValue<string>("Issuer"),
        ValidAudience = jwtSettings.GetValue<string>("Audience"),
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey))
    };
});

builder.Services.AddAuthorization();
```

2. Protecting Endpoints with JWT Authorization

After setting up authentication, you can protect specific endpoints by adding the [Authorize] attribute, which restricts access to authenticated users only.

Example Controller with Secured Endpoints

Here’s an example of a controller with secured actions:

```
using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;


[ApiController]

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

public class SecureDataController : ControllerBase

{

   // Only authenticated users can access this endpoint

   [HttpGet("secure-info")]

   [Authorize]

   public IActionResult GetSecureInfo()

   {

      return Ok("This is a secure endpoint only accessible to authenticated users.");

   }


   // Role-based authorization: Only users with the "Admin" role can access

   [HttpGet("admin-data")]

   [Authorize(Roles = "Admin")]

   public IActionResult GetAdminData()

   {

      return Ok("This is an admin-protected endpoint.");

   }

}
```

In this code:

- The [Authorize] attribute ensures that the user must be authenticated to access GetSecureInfo.
- The [Authorize(Roles = "Admin")] attribute restricts access to users with the Admin role, enforcing role-based authorization.

3. Generating JWT Tokens for Users

For authentication to work, you'll need a way to issue JWT tokens. Typically, this is handled by an AuthController where users authenticate with credentials and receive a token upon successful login.

Creating an AuthController for Login and Token Generation

In AuthController, implement a login method to verify user credentials and generate a JWT:

```
using Microsoft.AspNetCore.Mvc;

using System;

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using Microsoft.IdentityModel.Tokens;

using System.Text;


[ApiController]

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

public class AuthController : ControllerBase

{

   private readonly IConfiguration _configuration;


   public AuthController(IConfiguration configuration)

   {

      _configuration = configuration;

   }


   [HttpPost("login")]

   public IActionResult Login([FromBody] LoginModel model)
```

```
{
   // Simplified example: In practice, you would validate against a user database
   if (model.Username == "testuser" && model.Password == "password")
   {
      var token = GenerateJwtToken(model.Username);
      return Ok(new { Token = token });
   }

   return Unauthorized("Invalid credentials");
}

private string GenerateJwtToken(string username)
{
   var jwtSettings = _configuration.GetSection("JwtSettings");
   var secretKey = jwtSettings.GetValue<string>("SecretKey");

   var claims = new[]
   {
      new Claim(JwtRegisteredClaimNames.Sub, username),
      new Claim("role", "User"),
      new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
   };

   var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey));
   var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

   var token = new JwtSecurityToken(
      issuer: jwtSettings.GetValue<string>("Issuer"),
      audience: jwtSettings.GetValue<string>("Audience"),
      claims: claims,
      expires: DateTime.UtcNow.AddMinutes(30),
            signingCredentials: creds);


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

In this example:

- The Login action checks credentials, and if they’re correct, calls GenerateJwtToken.
- GenerateJwtToken generates a JWT with claims such as sub (subject, or username) and role.
- JwtSecurityTokenHandler.WriteToken(token) serializes the JWT, making it ready to send as a response.

Testing JWT Authentication and Authorization

1. **Authenticate** by calling api/auth/login with valid credentials to receive a JWT.
2. **Access Protected Endpoints** by passing the JWT in the Authorization header with the prefix Bearer, as shown below:

```
GET /api/SecureData/secure-info HTTP/1.1

Authorization: Bearer {JWT_TOKEN}
```

Refresh Tokens

JWTs are often short-lived to minimize risks in case a token is compromised. However, short expiration times can cause frequent interruptions for users. Refresh tokens help maintain user sessions smoothly by allowing clients to request a new JWT without requiring the user to re-authenticate.

1. What is a Refresh Token?

- A **refresh token** is a long-lived token issued alongside the JWT, allowing the client to request a new JWT when it expires.
- Unlike the JWT, which is sent on every request to the API, the refresh token is only sent to the authorization server when renewing the JWT.

2. How Refresh Tokens Work in JWT Authentication

Here's a simplified flow:

1. **User Authenticates**: Upon successful login, the server issues both a JWT and a refresh token to the client.
2. **Using JWT**: The client uses the JWT to access protected resources until it expires.
3. **Token Expiry and Refresh**:
   - When the JWT expires, the client sends the refresh token to a secure endpoint to obtain a new JWT.
   - If the refresh token is valid, the server issues a new JWT and, optionally, a new refresh token.
4. **Logout**: When a user logs out, the refresh token is invalidated to prevent further access.

3. Implementing Refresh Tokens in ASP.NET Core

To implement refresh tokens, you need:

- A storage solution to store and verify refresh tokens (often a database).
- An endpoint for issuing new tokens when the JWT expires.

Step 1: Extend AuthController with Refresh Token Logic

Model for Refresh Token: Define a model to represent a refresh token in your database. This model typically includes properties such as Token, UserId, ExpiryDate, and IsRevoked.

```
public class RefreshToken
{
   public string Token { get; set; }
   public string UserId { get; set; }
   public DateTime ExpiryDate { get; set; }
   public bool IsRevoked { get; set; }
}
```

Generate Refresh Token: Extend the GenerateJwtToken function to also create and return a refresh token.

```
private string GenerateRefreshToken()
{
   var randomNumber = new byte[32];
```

```
    using (var rng = RandomNumberGenerator.Create())
    {
        rng.GetBytes(randomNumber);
        return Convert.ToBase64String(randomNumber);
    }
}
```

Extend Login Response to Include Refresh Token: When a user logs in, return both the JWT and a refresh token.

```
[HttpPost("login")]
public IActionResult Login([FromBody] LoginModel model)
{
    if (model.Username == "testuser" && model.Password == "password")
    {
        var jwtToken = GenerateJwtToken(model.Username);
        var refreshToken = GenerateRefreshToken();

        // Store refreshToken in database associated with user
        SaveRefreshTokenToDatabase(model.Username, refreshToken);

        return Ok(new { Token = jwtToken, RefreshToken = refreshToken });
    }
    return Unauthorized("Invalid credentials");
}
```

Step 2: Implement Refresh Token Endpoint

Create an endpoint in AuthController to allow clients to refresh their JWT using the refresh token:

```
[HttpPost("refresh-token")]
public IActionResult RefreshToken([FromBody] RefreshTokenRequest request)
{
    var savedToken = GetStoredRefreshToken(request.RefreshToken);
    if (savedToken == null || savedToken.IsRevoked || savedToken.ExpiryDate < DateTime.UtcNow)
    {
        return Unauthorized("Invalid or expired refresh token.");
    }

    // Issue new JWT and refresh token
    var newJwtToken = GenerateJwtToken(savedToken.UserId);
    var newRefreshToken = GenerateRefreshToken();

    // Update refresh token in the database
    UpdateStoredRefreshToken(savedToken, newRefreshToken);

    return Ok(new { Token = newJwtToken, RefreshToken = newRefreshToken });
}
```

In this endpoint:

- The refresh token is validated against the database.
- If valid, a new JWT and refresh token are issued and returned.
- The previous refresh token is updated or revoked as per security policy.

Step 3: Store and Validate Refresh Tokens in Database

To manage refresh tokens securely:

- **Save Tokens**: Store tokens in a secure database table with fields such as Token, UserId, ExpiryDate, and IsRevoked.
- **Expire and Revoke**: Set expiration dates for refresh tokens, and revoke tokens on logout to prevent unauthorized access.

4. Security Best Practices for Refresh Tokens

- **Store Refresh Tokens Securely**: Keep them in a secure HTTP-only cookie or local storage on the client side.
- **Rotate Tokens**: Issue a new refresh token each time a JWT is refreshed to reduce the risk of token reuse.
- **Limit Token Scope**: Only allow the refresh token to request new JWTs, not access resources directly.
- **Short Expiration for JWT**: Keep JWTs short-lived, and use refresh tokens for session persistence.
- **Implement Logout**: Invalidate refresh tokens on user logout to prevent further use.

Key Points to Remember for JWT Authentication and Refresh Tokens

1. **JWT Structure**: Understand the structure (header, payload, signature) and how it ensures data integrity.
2. **Role of Refresh Tokens**: Used to extend session duration without frequent re-authentication.
3. **Secure JWT and Refresh Tokens**: Use best practices to store, validate, and revoke tokens.
4. **Code Flow**: Be familiar with code for generating JWTs, securing endpoints, and implementing refresh logic.
5. **ASP.NET Core Integration**: Master setup and configuration in ASP.NET Core, including authentication, authorization, and token validation.