

Asp.net core day 8 - JWT Authentication Flow

JWT Authentication Flow & Middleware Setup

(Minimal APIs Only)

1. JWT Authentication Flow (Step-by-Step)

JWT authentication in Minimal APIs follows a **request–token–validation** cycle.

Flow:

Client



Login Request (credentials)



API validates credentials



API generates JWT



Client stores JWT



Client sends JWT in Authorization header



API validates JWT on every request



Access granted / denied

JWT is validated **on every request**, making the system stateless.

2. Authentication vs Authorization (Clear Separation)

Term	Meaning
------	---------

Authentication Verifying *who* the user is

Authorization Verifying *what* the user can access

JWT primarily handles **authentication**, while **claims and roles** enable authorization.

3. Required Package for JWT (Minimal APIs)

JWT authentication requires the **JWT Bearer middleware**.

Install package:

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

This package adds:

- JWT validation middleware
 - Bearer token support
 - Token parsing & claims creation
-

4. JWT Configuration Section (appsettings.json)

JWT settings are usually externalized.

```
{  
  "Jwt": {  
    "Key": "THIS_IS_A_SUPER_SECRET_KEY_12345",  
    "Issuer": "MyMinimalApi",  
    "Audience": "MyMinimalApiUsers",  
    "ExpireMinutes": 60  
  }  
}
```

Meaning:

- **Key** → Secret used to sign tokens
 - **Issuer** → Token creator
 - **Audience** → Intended token consumer
 - **ExpireMinutes** → Token validity
-

5. Add JWT Authentication Services

JWT authentication is configured during the **builder stage**.

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

```

var builder = WebApplication.CreateBuilder(args);

var jwtConfig = builder.Configuration.GetSection("Jwt");

builder.Services

    .AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

    .AddJwtBearer(options =>

    {

        options.TokenValidationParameters = new TokenValidationParameters

        {

            ValidateIssuer = true,

            ValidateAudience = true,

            ValidateLifetime = true,

            ValidateIssuerSigningKey = true,


            ValidIssuer = jwtConfig["Issuer"],

            ValidAudience = jwtConfig["Audience"],

            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(jwtConfig["Key"]))
        }
    });

});

builder.Services.AddAuthorization();

```

6. Middleware Order in Minimal APIs (Critical)

Authentication and Authorization must be added **in the correct order**.

```
var app = builder.Build();
```

```
app.UseHttpsRedirection();
```

```
app.UseCors("FrontendPolicy");

app.UseAuthentication(); // validates JWT
app.UseAuthorization(); // enforces access rules

app.MapGet(...);

app.Run();
```

Rules:

- UseAuthentication() **must come before** UseAuthorization()
 - Both must execute **before protected endpoints**
-

7. What Authentication Middleware Does Internally

For each incoming request:

1. Reads Authorization header
2. Extracts Bearer <token>
3. Validates:
 - Signature
 - Expiry
 - Issuer
 - Audience
4. Builds a ClaimsPrincipal
5. Attaches it to HttpContext.User

If validation fails → request is rejected.

8. Authorization Header Format

JWT must be sent in this format:

Authorization: Bearer eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9...

Any deviation (missing Bearer, invalid token) results in **401 Unauthorized**.

9. Protecting Minimal API Endpoints

9.1 Require Authentication

```
app.MapGet("/profile", (ClaimsPrincipal user) =>
{
    return $"Welcome {user.Identity?.Name}";
})
.RequireAuthorization();
```

This endpoint:

- Requires a valid JWT
 - Rejects unauthenticated requests
-

9.2 Public Endpoint (No JWT Required)

```
app.MapGet("/public", () => "Public endpoint");
```

Endpoints without .RequireAuthorization() remain public.

10. Reading User Information from JWT

Once authenticated, user information is available via ClaimsPrincipal.

```
app.MapGet("/claims", (ClaimsPrincipal user) =>
{
    return user.Claims.Select(c => new { c.Type, c.Value });
})
.RequireAuthorization();
```

Claims originate from the JWT payload.

11. Role-Based Authorization (Minimal APIs)

JWT often includes a role claim.

Example protected endpoint:

```
app.MapGet("/admin", () => "Admin access")
    .RequireAuthorization(policy => policy.RequireRole("Admin"));
```

Behavior:

- Token must be valid

- Token must contain role = Admin
 - Otherwise → **403 Forbidden**
-

12. Common JWT Response Codes

Status Code	Meaning
401 Unauthorized	Missing / invalid / expired token
403 Forbidden	Token valid but insufficient role
200 OK	Authorized request
500	Misconfiguration or server error

13. JWT + CORS Dependency (Reminder)

JWT requests usually include:

Authorization: Bearer <token>

This triggers **CORS preflight** in browsers.

CORS policy must allow:

- Authorization header
- Required HTTP methods

Example:

```
policy.AllowAnyHeader()  
    .AllowAnyMethod();
```

14. JWT Token Expiration Handling

When ValidateLifetime = true:

- Expired token → automatically rejected
- API returns **401 Unauthorized**

Expiration time is checked against exp claim.

15. Summary

- JWT authentication is middleware-based
- Tokens are validated on every request

- Authentication builds ClaimsPrincipal
 - Authorization checks roles and policies
 - Minimal APIs use .RequireAuthorization()
 - Middleware order is critical
 - JWT integrates tightly with CORS
-

Exercises for Students

1. Configure JWT authentication with issuer and audience validation.
 2. Create a protected endpoint /profile requiring JWT.
 3. Add an /admin endpoint restricted to role Admin.
 4. Display all claims from a valid JWT.
 5. Test token expiration by reducing expiry time.
-

JWT + Dapper User Validation (Database-backed Login)

Minimal APIs Only

1. Objective

Implement **database-backed login** using:

- **Dapper** for user lookup/validation
- **JWT** for token issuance
- **Minimal APIs** for endpoints

Result:

- POST /login validates username/password from DB
 - API issues JWT on success
 - Protected endpoints require JWT
-

2. Database Table Design (Recommended)

2.1 Users Table (SQL Server)

```
CREATE TABLE Users (
```

```
    UserId     INT IDENTITY(1,1) PRIMARY KEY,  
    Username   NVARCHAR(50) NOT NULL UNIQUE,
```

```
        PasswordHash NVARCHAR(200) NOT NULL,  
        PasswordSalt NVARCHAR(200) NOT NULL,  
        Role      NVARCHAR(20) NOT NULL,  
        IsActive   BIT NOT NULL DEFAULT 1,  
        CreatedAt  DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME()  
    );
```

2.2 Why Salt + Hash

- Passwords are never stored as plain text.
 - PasswordHash + PasswordSalt supports secure verification.
-

3. Appsettings.json (JWT + Connection String)

```
{  
    "ConnectionStrings": {  
        "Default": "Server=.;Database=MyDb;Trusted_Connection=True;TrustServerCertificate=True"  
    },  
    "Jwt": {  
        "Key": "THIS_IS_A_SUPER_SECRET_KEY_12345_THIS_SHOULD_BE_LONG",  
        "Issuer": "MyMinimalApi",  
        "Audience": "MyMinimalApiUsers",  
        "ExpireMinutes": 60  
    }  
}
```

4. Required Packages

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

5. Models Used

5.1 Database Entity

```
public class DbUser  
{
```

```
public int UserId { get; set; }

public string Username { get; set; } = "";

public string PasswordHash { get; set; } = "";

public string PasswordSalt { get; set; } = "";

public string Role { get; set; } = "User";

public bool IsActive { get; set; }

}
```

5.2 Login Request DTO

```
public record LoginRequest(string Username, string Password);
```

5.3 Login Response DTO

```
public record LoginResponse(string Token, DateTime ExpiresUtc, string Username, string Role);
```

6. Password Hashing (PBKDF2 Recommended)

Use PBKDF2 via Rfc2898DeriveBytes (no external library required).

6.1 Hash Helper

```
using System.Security.Cryptography;
```

```
static (string hashBase64, string saltBase64) HashPassword(string password)
```

```
{
```

```
    byte[] salt = RandomNumberGenerator.GetBytes(16); // 128-bit salt
```

```
    using var pbkdf2 = new Rfc2898DeriveBytes(
```

```
        password,
```

```
        salt,
```

```
        iterations: 100_000,
```

```
        hashAlgorithm: HashAlgorithmName.SHA256);
```

```
    byte[] hash = pbkdf2.GetBytes(32); // 256-bit hash
```

```
    return (Convert.ToBase64String(hash), Convert.ToBase64String(salt));
```

```
}
```

6.2 Verify Helper

```
static bool VerifyPassword(string password, string storedHashBase64, string storedSaltBase64)
{
    byte[] salt = Convert.FromBase64String(storedSaltBase64);
    byte[] storedHash = Convert.FromBase64String(storedHashBase64);

    using var pbkdf2 = new Rfc2898DeriveBytes(
        password,
        salt,
        iterations: 100_000,
        hashAlgorithm: HashAlgorithmName.SHA256);

    byte[] computedHash = pbkdf2.GetBytes(32);

    return CryptographicOperations.FixedTimeEquals(storedHash, computedHash);
}
```

7. Dapper User Lookup (Repository Pattern)

7.1 Interface

```
using System.Threading.Tasks;
```

```
public interface IUserRepository
{
    Task<DbUser?> FindByUsernameAsync(string username);
}
```

7.2 Implementation (Dapper)

```
using System.Data;
using System.Data.SqlClient;
using Dapper;
```

```
public class UserRepository : IUserRepository
```

```

{
    private readonly string _connStr;

    public UserRepository(IConfiguration cfg) =>
        _connStr = cfg.GetConnectionString("Default")!;

    private IDbConnection CreateConnection() => new SqlConnection(_connStr);

    public async Task<DbUser?> FindByUsernameAsync(string username)
    {
        const string sql = @"
SELECT UserId, Username, PasswordHash, PasswordSalt, Role, IsActive
FROM Users
WHERE Username = @Username";

        using var conn = CreateConnection();
        return await conn.QueryFirstOrDefaultAsync<DbUser>(sql, new { Username = username });
    }
}

```

8. JWT Token Generation (With Claims)

8.1 Token Generator

```

using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;

```

```

static (string token, DateTime expiresUtc) CreateJwt(DbUser user, IConfiguration config)
{
    var jwt = config.GetSection("Jwt");
    var key = jwt["Key"]!;
    var issuer = jwt["Issuer"]!;

```

```

var audience = jwt["Audience"]!;
var expireMinutes = int.Parse(jwt["ExpireMinutes"]!);

var expires = DateTime.UtcNow.AddMinutes(expireMinutes);

var claims = new List<Claim>
{
    new(JwtRegisteredClaimNames.Sub, user.UserId.ToString()),
    new(ClaimTypes.Name, user.Username),
    new(ClaimTypes.Role, user.Role),
    new(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
};

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

var token = new JwtSecurityToken(
    issuer: issuer,
    audience: audience,
    claims: claims,
    expires: expires,
    signingCredentials: creds);

var tokenStr = new JwtSecurityTokenHandler().WriteToken(token);
return (tokenStr, expires);
}

```

9. Minimal API Setup (Authentication + Authorization + DI)

9.1 Program.cs (Core Setup)

```

using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;

```

```
using System.Text;

var builder = WebApplication.CreateBuilder(args);

// DI for repository
builder.Services.AddScoped<IUserRepository, UserRepository>();

// JWT auth
var jwtConfig = builder.Configuration.GetSection("Jwt");
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(o =>
{
    o.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = jwtConfig["Issuer"],
        ValidAudience = jwtConfig["Audience"],
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(jwtConfig["Key"]!))
    }
});

builder.Services.AddAuthorization();

var app = builder.Build();

app.UseHttpsRedirection();
```

```
app.UseAuthentication();  
app.UseAuthorization();
```

10. Login Endpoint (Database-backed)

```
app.MapPost("/login", async (LoginRequest req, IUserRepository repo, IConfiguration config) =>  
{  
    // 1) lookup user  
  
    var user = await repo.FindByUsernameAsync(req.Username);  
  
    // 2) generic failure (avoid telling whether user exists)  
  
    if (user is null || !user.IsActive)  
        return Results.Unauthorized();  
  
    // 3) verify password  
  
    bool ok = VerifyPassword(req.Password, user.PasswordHash, user.PasswordSalt);  
    if (!ok)  
        return Results.Unauthorized();  
  
    // 4) create token  
  
    var (token, expiresUtc) = CreateJwt(user, config);  
  
    // 5) return response  
  
    return Results.Ok(new LoginResponse(token, expiresUtc, user.Username, user.Role));  
};
```

Key points:

- Uses Dapper repo for DB access
 - Uses secure password verification
 - Issues JWT containing user id, username, role
-

11. Protected Endpoint Examples

11.1 Any authenticated user

using System.Security.Claims;

```
app.MapGet("/profile", (ClaimsPrincipal user) =>
{
    var name = user.Identity?.Name;
    return Results.Ok(new { name });
})
```

.RequireAuthorization();

11.2 Role restricted

```
app.MapGet("/admin", () => Results.Ok("Admin area"))
    .RequireAuthorization(p => p.RequireRole("Admin"));
```

12. Seed User (Create Admin Row)

Example insert (hash first using HashPassword helper):

```
// one-time generation example (run in a small console or temp endpoint):
var (hash, salt) = HashPassword("1234");
```

Then insert:

```
INSERT INTO Users (Username, PasswordHash, PasswordSalt, Role, IsActive)
VALUES ('admin', '<HASH_BASE64>', '<SALT_BASE64>', 'Admin', 1);
```

13. Common Failure Codes

Result	Meaning
200 OK	login success, token issued
401 Unauthorized	invalid username/password or inactive
403 Forbidden	token valid but role not allowed
500	misconfiguration (JWT key/issuer/audience) or DB failure

14. Security Notes

- Never store plain passwords.

- Use PBKDF2/BCrypt/Argon2 (PBKDF2 shown here).
 - Use HTTPS only.
 - Avoid exposing “user not found” vs “wrong password”.
 - Keep JWT secret key long and private.
 - Short token expiry (15–60 minutes) is typical.
 - If using browser clients, ensure CORS allows Authorization header.
-

15. Exercises for Students

1. Add Email column and include it as a claim in JWT.
 2. Add endpoint /change-password requiring authorization; update hash+salt in DB.
 3. Add endpoint /users/me that returns userId and role from claims.
 4. Block login if IsActive = 0.
 5. Add a second role Manager and restrict an endpoint using role policy.
-