ASP.NET Core Web API – Authentication & Authorization

By
Narasimha Rao T
Microsoft.Net FSD Trainer
Professional Development Trainer
tnrao.trainer@gmail.com

1. Authentication Concepts

Authentication verifies **who** a user is. Two primary approaches in web apps:

1.1 Session-Based Authentication

- Server stores user session in memory or distributed cache.
- Client receives a **session ID cookie**.
- Server must keep session state, making the system **stateful**.
- Harder to scale → sticky sessions, shared distributed cache.
- Typical in traditional ASP.NET MVC, PHP apps.

1.2 Token-Based Authentication

- Server issues a **token** (usually JWT) after login.
- Client stores the token and sends it with each request (via Authorization header).
- Server **does not store session state → stateless**.
- Perfect for distributed, cloud-native apps (microservices, SPAs, mobile).

2. JWT Authentication – Introduction

JWT = **JSON Web Token**, a digitally signed token containing user identity & claims.

2.1 JWT Structure

A JWT has **three Base64Url-encoded segments**:

1. **Header**

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

2. **Payload (Claims)**

   Contains user info:

```
{
  "sub": "userId",
  "email": "example@test.com",
  "role": "Admin",
  "exp": 1700000000
}
```

3. **Signature**

```
HMACSHA256(base64Header + "." + base64Payload, secretKey)
```

Full token format:

```
header.payload.signature
```

2.2 JWT Authentication Flow

1. User sends login credentials (email & password).
2. Server validates credentials.
3. Server issues a **JWT** signed using a secret/private key.
4. Client stores the token (localStorage/sessionStorage/mobile secure storage).
5. For each API request, client sends:

```
Authorization: Bearer <token>
```

6. Server validates the signature and expiration.
7. Request is executed if token is valid.

3. Implementing JWT-Based Login (ASP.NET Core)

3.1 Add JWT configuration in `appsettings.json`

```
"Jwt": {
  "Key": "supersecretkey123456",
  "Issuer": "myapi",
  "Audience": "myapi_users",
  "ExpiresInMinutes": 30
}
```

3.2 Register JWT Services in `Program.cs`

```
builder.Services.AddAuthentication("Bearer")
    .AddJwtBearer(options =>
    {
        var config = builder.Configuration.GetSection("Jwt");
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidIssuer = config["Issuer"],
            ValidateAudience = true,
            ValidAudience = config["Audience"],
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(config["Key"])),
            ValidateLifetime = true
        };
    });
```

3.3 Token Generation Service

```
public string GenerateToken(ApplicationUser user, string role)
{
    var claims = new[]
    {
        new Claim(JwtRegisteredClaimNames.Sub, user.Id),
        new Claim(JwtRegisteredClaimNames.Email, user.Email),
        new Claim(ClaimTypes.Role, role)
    };

    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

    var token = new JwtSecurityToken(
        issuer: _config["Jwt:Issuer"],
        audience: _config["Jwt:Audience"],
        claims: claims,
        expires: DateTime.UtcNow.AddMinutes(30),
        signingCredentials: creds
    );

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

3.4 JWT-Protected Endpoint

```
[Authorize]
[HttpGet("secure-data")]
public IActionResult GetSecureData()
{
    return Ok("You are authenticated!");
}
```

4. ASP.NET Core Identity Overview

ASP.NET Core Identity is a **membership system** providing:

- User registration & login.
- Password hashing.
- Role management.
- Claims-based authentication.
- Token generation (email confirmation, password reset).

Identity uses `IdentityUser` class and EF Core database tables (`AspNetUsers` , `AspNetRoles` , etc).

5. User Registration & Login with Identity

5.1 Register Identity in `Program.cs`

```
builder.Services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

5.2 Registration Endpoint

```
[HttpPost("register")]
public async Task<IActionResult> Register(RegisterDto model)
{
    var user = new ApplicationUser
    {
        UserName = model.Email,
        Email = model.Email
    };

    var result = await _userManager.CreateAsync(user, model.Password);

    if (!result.Succeeded)
        return BadRequest(result.Errors);

    return Ok("User registered successfully");
}
```

5.3 Login Endpoint with JWT Issuing

```
[HttpPost("login")]
public async Task<IActionResult> Login(LoginDto model)
{
    var user = await _userManager.FindByEmailAsync(model.Email);
    if (user == null) return Unauthorized();

    if (!await _userManager.CheckPasswordAsync(user, model.Password))
        return Unauthorized();

    // Get role(s)
    var roles = await _userManager.GetRolesAsync(user);
    var token = _jwtService.GenerateToken(user, roles.FirstOrDefault());

    return Ok(new { token });
}
```

6. Password Hashing, Claims, and Identity Security

6.1 Password Hashing

- Identity uses **PBKDF2** hashing algorithm.
- Passwords are NEVER stored in plain text.
- Hash includes a salt and iteration count.

6.2 Claims

Claims represent attributes of the user:

- Name
- Email
- Roles
- Custom claims (e.g., department, permissions)

7. Role-Based Authorization

7.1 Decorating Endpoint with Role Requirement

```
[Authorize(Roles = "Admin")]
[HttpGet("admin-panel")]
public IActionResult AdminOnly()
{
    return Ok("Admin access granted");
}
```

Assigning a role to user

```
await _roleManager.CreateAsync(new IdentityRole("Admin"));
await _userManager.AddToRoleAsync(user, "Admin");
```

8. Custom Policy-Based Authorization

Policies allow custom rules beyond roles.

8.1 Configure Policy

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("RequireAge18", policy =>
        policy.RequireClaim("Age", "18"));
});
```

8.2 Apply Policy

```
[Authorize(Policy = "RequireAge18")]
[HttpGet("adult-section")]
public IActionResult AdultSection()
{
    return Ok("You meet the age requirement");
}
```

Custom requirement handler

You can implement custom logic by inheriting `IAuthorizationRequirement`.

9. Securing Endpoints Using JWT + Identity

Steps:

1. User registers via Identity.
2. User logs in with Identity, password checker ensures security.
3. Server generates a JWT with Identity claims + roles.
4. Client stores token securely.
5. Each API request includes `Authorization: Bearer <token>`.

6. The `[Authorize]` attribute ensures:
    - Token validity
    - Claims/roles availability
    - Policy matching
7. Server processes the request only when authenticated & authorized.

Summary Slide

- **Session vs Token**: Stateful vs Stateless.
- **JWT**: Three-part token → Header, Payload, Signature.
- **JWT Flow**: Login → Validate → Generate token → Secure endpoints.
- **Identity**: Full user management system with password hashing & roles.
- **Authorization**: Role-based and Policy-based.
- **JWT + Identity** = Best practice for modern Web API security.

Q & A

Narasimha Rao T
tnrao.trainer@gmail.com