Dokumentasjon: Sikkerhet i ASP.NET Core – AuthenticationHandler, Authorize og Identity

1. Introduksjon

ASP.NET Core har et kraftig **autentiserings- og autorisasjonssystem** bygget inn i rammeverket. Dette systemet er modulært og består av tre sentrale deler:

1. **Authentication (autentisering)**
   – Hvem er du? Bevis identitet via passord, JWT, API-key, OIDC, osv.

2. **Authorization (autorisasjon)**
   – Hva har du lov til å gjøre? Styr tilgang til ressurser via roller, policies og claims.

3. **User Management**
   – Hvordan håndteres brukere og deres data? ASP.NET Identity tilbyr et komplett brukerhåndteringssystem, men du kan også bygge dine egne løsninger.

---

2. Helhetlig arkitektur

```
[Klient: nettleser/app/postman]
            |
            v
HTTP Request (med token, cookie, API-key)
            |
            v
+--------------------------+
| Authentication middleware |
+--------------------------+
            |
   Validerer legitimasjon via
   registrerte autentiserings-skjemaer:
   - JWT
   - Cookies
   - API Key
   - OAuth2/OpenID Connect
            |
            v
HttpContext.User = ClaimsPrincipal
            |
            v
[Authorize] filter sjekker claims/policies
            |
            v
Controller-action utføres eller 401/403
```

> **Viktig:** `HttpContext.User` er alltid **kilden** for hvem som er logget inn og hvilke claims/roller som finnes.

---

3. AuthenticationHandler – kjernen i autentisering

3.1 Hva er en AuthenticationHandler?

En `AuthenticationHandler<TOptions>` er en klasse som **implementerer logikken** for å validere legitimasjon. Eksempler:

- `JwtBearerHandler` (JWT)
- `CookieAuthenticationHandler` (cookies)
- Din egen `ApiKeyAuthenticationHandler`

Oppgave:

1. Les header eller cookie fra request.
2. Verifiser legitimasjon (signatur, passord, nøkkel).
3. Bygg en `ClaimsPrincipal` for brukeren.
4. Returner `AuthenticateResult.Success(ticket)`.

3.2 Hvorfor bruke AuthenticationHandler?

- Integreres **sømløst** med ASP.NET Core auth-systemet.
- Gir **standard håndtering** av 401 Unauthorized og 403 Forbidden.
- Gjør det mulig å bruke `[Authorize]`, policies og roller.

3.3 Eksempel: Minimal API Key Handler

```
public class ApiKeyAuthenticationHandler :
AuthenticationHandler<AuthenticationSchemeOptions>
{
    private const string HeaderName = "x-api-key";
    private readonly IConfiguration _config;

    public ApiKeyAuthenticationHandler(
        IOptionsMonitor<AuthenticationSchemeOptions> options,
        ILoggerFactory logger,
        UrlEncoder encoder,
        ISystemClock clock,
        IConfiguration config)
        : base(options, logger, encoder, clock)
    {
        _config = config;
    }

    protected override Task<AuthenticateResult> HandleAuthenticateAsync()
    {
        if (!Request.Headers.TryGetValue(HeaderName, out var providedKey))
            return Task.FromResult(AuthenticateResult.Fail("Missing API Key"));

        var expectedKey = _config["ApiSettings:Key"];
        if (providedKey != expectedKey)
            return Task.FromResult(AuthenticateResult.Fail("Invalid API Key"));

        var claims = new[] { new Claim(ClaimTypes.Name, "ApiClient") };
        var identity = new ClaimsIdentity(claims, Scheme.Name);
        var principal = new ClaimsPrincipal(identity);
        var ticket = new AuthenticationTicket(principal, Scheme.Name);

        return Task.FromResult(AuthenticateResult.Success(ticket));
    }
}
```

Registrering i `Program.cs`:

```
builder.Services.AddAuthentication("ApiKey")
    .AddScheme<AuthenticationSchemeOptions, ApiKeyAuthenticationHandler>("ApiKey",
null);
```

---

4. Hvordan `[Authorize]` fungerer

4.1 `[Authorize]` kobles til autentisering

Når du markerer en controller eller action med `[Authorize]`, skjer dette i pipeline:

1. ASP.NET Core ser at `[Authorize]` er aktivt.
2. Frameworket ser etter **aktive autentiserings-skjemaer** via `AddAuthentication()`.
3. Den første (eller spesifikt navngitte) handler validerer requesten.
4. Hvis validert → `HttpContext.User` fylles med claims fra handler.
5. `[Authorize]` sjekker om `HttpContext.User.Identity.IsAuthenticated == true`.

Eksempel:

```
[Authorize] // krever at brukeren er autentisert
[HttpGet("secure-data")]
public IActionResult GetSecureData()
{
    return Ok(new { message = "Kun for autentiserte klienter!" });
}
```

4.2 Policies og roller

Du kan utvide `[Authorize]` med policies eller roller:

```
[Authorize(Roles = "Admin")]
public IActionResult DeleteUser(int id) => Ok();

[Authorize(Policy = "CanEditPosts")]
public IActionResult EditPost(int id) => Ok();
```

Policies defineres slik:

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("CanEditPosts", policy =>
        policy.RequireClaim("scope", "posts:edit"));
});
```

---

5. Identity – komplett brukerhåndtering

5.1 Hva er Identity?

ASP.NET Identity er et **ferdig system for brukere og roller**, inkludert:

- Registrering og innlogging
- Passordhåndtering (med hashing og salt)
- Roller og claims
- 2FA, e-postbekreftelse, passord-reset
- Integrasjon med Entity Framework Core

5.2 Identity og AuthenticationHandler

Identity bruker **CookieAuthenticationHandler** under panseret. Når en bruker logger inn via Identity:

- En cookie genereres og signeres.
- På neste request leser handleren cookien, validerer den og bygger en `ClaimsPrincipal`.
- `[Authorize]` fungerer helt likt som for andre handlers.

> **Viktig:** Du kan **kombinere Identity med andre autentiseringsmetoder**, som JWT eller API Keys.

5.3 Oppsett av Identity med EF Core

Installer pakken:

```
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
```

DbContext med Identity-integrasjon:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options) { }

    public DbSet<Post> Posts { get; set; } = null!;
}

public class ApplicationUser : IdentityUser
{
    // Egne felter kan legges til her
    public string FullName { get; set; } = string.Empty;
}
```

Registrering i `Program.cs`:

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"
)));

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

5.4 Migreringer og database

```
dotnet ef migrations add InitialIdentitySchema
dotnet ef database update
```

Dette oppretter tabeller som:

- **AspNetUsers**
- **AspNetRoles**
- **AspNetUserRoles**
- **AspNetUserClaims**
- **AspNetUserLogins**
- **AspNetUserTokens**

---

6. Egendefinert brukerhåndtering uten Identity

Du **må ikke** bruke Identity. Du kan lage egne modeller og tabeller for brukere.

- Du må selv implementere registrering, passordhashing, pålogging.
- Lag en `AuthenticationHandler` eller JWT-generator som bygger `ClaimsPrincipal` fra ditt system.

Eksempel på minimal User-modell:

```
public class User
{
    public Guid UserId { get; set; }
    public string UserName { get; set; } = string.Empty;
    public string PasswordHash { get; set; } = string.Empty;
    public ICollection<UserRole> Roles { get; set; } = new List<UserRole>();
}
```

DbContext:

```
public class CustomDbContext : DbContext
{
    public CustomDbContext(DbContextOptions<CustomDbContext> options) :
base(options) { }

    public DbSet<User> Users { get; set; } = null!;
    public DbSet<Role> Roles { get; set; } = null!;
    public DbSet<UserRole> UserRoles { get; set; } = null!;
}
```

> Fordel: Full kontroll.
> Ulempe: Du mister ferdige funksjoner som 2FA og passord-reset.

---

7. Integrasjon av egne brukere med `[Authorize]`

Selv om du ikke bruker Identity, kan du likevel fylle `HttpContext.User` med dine data via en handler eller JWT.
Dette gir samme sluttresultat som Identity: `[Authorize]` og policies fungerer uansett.

Eksempel: JWT med egen bruker:

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.NameIdentifier, user.UserId.ToString()),
    new Claim(ClaimTypes.Name, user.UserName)
};
foreach (var role in user.Roles)
{
    claims.Add(new Claim(ClaimTypes.Role, role.Name));
}
```

Disse claims legges i token → token valideres → `HttpContext.User` bygges.

---

8. Oversikt: Identity vs Egne løsninger

Funksjon	Identity	Egendefinert løsning
Registrering/logg inn	Innebygd	Må bygges selv
Passordhåndtering	Ferdig med hashing/salt	Må bygges selv
Roller/claims	Ferdig tabeller	Må designes selv
2FA, e-postbekreftelse	Ferdig	Må bygges selv
Integrasjon med `[Authorize]`	Ja	Ja (via handler/JWT)
Kontroll/fleksibilitet	Begrenset	Full kontroll

> Anbefaling: Start med Identity for rask utvikling. Bygg egen løsning når du trenger spesialfunksjoner.

---

9. Flyt med Identity + JWT (kombinasjon)

```
[Klient] ---- (brukernavn/passord) ----> [API: Identity login endpoint]
               Bekrefter bruker og roller
               Genererer JWT med claims
[Klient] <---- JWT tilbake

[Klient] ---- (Bearer token) ----> [API: Secure endpoint]
               JwtBearerHandler validerer token
               Fyller HttpContext.User
               [Authorize] tillater eller blokkerer
```

Denne kombinasjonen er vanlig i moderne API-design.

---

10. Oppsummering

- **AuthenticationHandler** er nøkkelen til å bygge dine egne autentiseringsmekanismer.
- `[Authorize]` bruker alltid `HttpContext.User` for å avgjøre tilgang.
- ASP.NET **Identity** er en ferdig løsning for brukere, men du kan lage egne modeller og handlers.
- JWT eller API Keys kan enkelt integreres med Identity eller egne løsninger.
- Policies gir deg fleksibel, claim-basert autorisasjon.

---

11. Referanser

- Microsoft Docs: Authentication in ASP.NET Core
- Microsoft Docs: Identity in ASP.NET Core
- Microsoft Docs: Authorization in ASP.NET Core