SECURITY & AUTH PATTERNS · MODULE 5

Federation with OpenID Connect

Lesson Guide · Lab / Exam Guide

LearnWisej

Lesson Guide

OAuth versus OpenID Connect, the authorization code flow with PKCE through the ASP.NET Core cookie and OpenID Connect handlers registered in Startup.cs before app.UseWisej(), token validation, issuer and subject account mapping into the Wisej.NET session, local-login fallback and sign-out of both sessions. What this means for a developer securing the sign-in, identity and authorization paths of a production Wisej.NET app — and how to read this module.

Production focus

OAuth versus OpenID Connect, the authorization code flow with PKCE through the ASP.NET Core cookie and OpenID Connect handlers registered in Startup.cs before app.UseWisej(), token validation, issuer and subject account mapping into the Wisej.NET session, local-login fallback and sign-out of both sessions.

1. The production problem

AccessOps partners already have accounts at their own organisation, with its own multi-factor policy and leaver process, so the partner route delegates authentication to an OpenID Connect provider instead of keeping a second password. The identity now travels through the user's browser, where every parameter can be forged, replayed or swapped. Wisej.NET 4.1.7 has no federation component, so AccessOps wires ASP.NET Core's standard handlers into its host and hands their verified result to the Wisej.NET session.

2. Key concepts

Concept	Why it matters
OAuth versus OpenID Connect	OAuth authorizes a client to call a protected API; OpenID Connect adds an identity layer, an ID token that tells the client who signed in.
authorization code + PKCE	The browser only carries a short-lived code; the server redeems it in the back channel with a code verifier that never left the server, so a stolen code is useless.
state and nonce	State ties the callback to the browser attempt that started it; the nonce ties the ID token to that same request, so neither can be replayed into another sign-in.
ID token and access token	The ID token is identity evidence for AccessOps itself; the access token belongs to the resource server it was issued for and proves nothing about who is at the keyboard.
issuer + subject	The validated `iss` and `sub` pair is the only stable external account key; email addresses and display names can be reassigned.
exact callback, pinned authority	The provider sends codes only to a registered redirect URI, and the handler loads keys and endpoints only from one configured authority, so neither can be redirected.
hand-off handle	A one-time random value in an HttpOnly cookie that lets the Wisej.NET page load pick up a validated identity; the browser never holds a token.

3. Lesson narrative

One code, two proofs: the flow with PKCE, state and nonce

A partner clicks *Sign in with your work account*. The AccessOps page calls `IExternalSignIn.Begin()`, which navigates the tab to the ordinary ASP.NET Core endpoint `/account/signin`; that endpoint issues a challenge for `OpenIdConnectDefaults.AuthenticationScheme`. The handler now creates three secrets: a random state, protected and backed by a correlation cookie in this browser; a nonce, kept in a nonce cookie; and a PKCE code verifier whose SHA-256 hash travels to the provider as the S256 challenge. The partner signs in at the provider; AccessOps never sees the password.

The provider returns the browser to `/signin-oidc` with a code and the state. The handler first checks that the state unprotects and matches the correlation cookie of *this* browser. It then redeems the code at the token endpoint over the back channel with the client secret and the original verifier, proving the redeemer started the attempt. Finally it validates the ID token: signature against the provider's published keys, issuer, audience, lifetime and nonce. Only then is there an identity.

Keep the two tokens apart. The ID token answers *who signed in, at which provider, for this client*. An access token is addressed to a resource server, such as a partner directory API, and AccessOps only sends it there, from the server. Signing someone in from a decoded access token skips every check above.

Startup.cs: the handlers before app.UseWisej()

A Wisej.NET 4 application is an ASP.NET Core application, so federation is configured in `Startup.cs` with the `Microsoft.AspNetCore.Authentication.OpenIdConnect` 10.0.14 package:

`AddAccessOpsOpenIdConnect()` registers the handlers, and `MapAccessOpsAccountEndpoints()` calls `app.UseAuthentication()` and maps the account endpoints, all before `app.UseWisej()`, so `/signin-oidc` and `/signout-callback-oidc` are answered by the authentication middleware.

```
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie()
    .AddOpenIdConnect(o =>
    {
        o.Authority = oidc["Authority"];        // pinned, HTTPS discovery
        o.ClientId = oidc["ClientId"];
        o.ClientSecret = oidc["ClientSecret"];  // Oidc__ClientSecret at runtime
        o.ResponseType = "code";
        o.UsePkce = true;
        o.SaveTokens = false;                   // tokens stay on the server
        o.Scope.Clear(); o.Scope.Add("openid"); o.Scope.Add("profile");
        o.Events = new OpenIdConnectEvents { OnTokenValidated = HandOffToWisej };
    });
// app.UseAuthentication(); endpoints; then app.UseWisej();
```

`Authority` comes from configuration and is the only place discovery and signing keys are fetched from; `RequireHttpsMetadata` stays on outside the lab. The provider registration lists one exact HTTPS callback ending in the handler's `CallbackPath`, so behind a TLS-terminating proxy the canonical origin must survive (`app.UseForwardedHeaders()`), or the handler builds an `http://` redirect URI the provider refuses. The client secret arrives as `Oidc__ClientSecret` from the environment or user secrets, which only load when `GenerateAssemblyInfo` is true (the Wisej template sets it false). Keep `MapInboundClaims` off so `iss` and `sub` keep their protocol names, and match `ResponseMode` to the provider; the default is `form_post`.

From a validated token to a Wisej.NET principal

The callback request has no Wisej.NET session, and `Application.RefreshSessionId()` only reaches the browser on a page load. So `OnTokenValidated` does not sign anyone in: it builds an `ExternalIdentity` from the validated issuer and subject, parks it in `ExternalHandoffStore` under a random handle, writes that handle to the HttpOnly `AccessOps.Handoff` cookie, calls `HandleResponse()` so no host cookie session is created, and redirects to the start page.

On that page load, in `Program.Main` or `Application.ApplicationRefresh`, `WisejExternalSignIn.TryCompleteOnPageLoad()` reads the handle with `Application.Cookies[ … ]`, expires the cookie and redeems the handle once. `ExternalIdentityMapper` looks the issuer and subject up in `external_logins`, never by email, refuses a disabled account or a tenant AccessOps has not approved, and returns the local account with its tenant and roles from AccessOps data. Then the Module 4 `GrantStore` issues exactly one login grant, and `ApplicationPrincipal.Establish( … )` sets `Application.User` to a `ClaimsPrincipal` carrying account, tenant and grant and rotates the session ID. Nothing in the token became a permission.

The Module 4 password form stays under *Service accounts only*: `LoginService` accepts named service accounts such as `svc.reports` and gives every interactive account the generic failure, so the fallback is never a second way in.

Rejections before the grant, and two sign-outs

Every failure must stop before `GrantStore` runs. A callback carrying another attempt's state (fixture F06) fails to unprotect, and the handler raises `OnRemoteFailure`; AccessOps logs an allowlisted reason, never the callback query, and redirects to the sign-in view. A replayed callback (F07) finds its correlation cookie already consumed and its code already redeemed, and a replayed handle finds nothing in the store, so the attempt still has one grant in total. A valid identity from an unapproved tenant (F08) passes the protocol checks and is refused by the mapper with no principal. A provider denial arrives through `OnAccessDenied` or `OnRemoteFailure` and lands on the sign-in view with a retry path.

Sign-out ends two sessions, and each is verified separately. Locally, AccessOps revokes the grant, so an export retried from a second open tab is denied, then ends the Wisej.NET session. The browser then visits `/account/signout`, which returns `Results.SignOut` for the cookie and OpenID Connect schemes; the handler sends it to the provider's end-session endpoint and back through `/signout-callback-oidc`. Proof that this worked is the next sign-in asking for credentials again. Neither proves refresh tokens were revoked; check revocation, where offered, as a third result.

Security habit

Keep the identities apart and decide on the server. The host principal, the Wisej.NET application principal and the effective Windows token are three different facts, and none of them is a permission. Every protected read checks the current grant, the tenant and the record at the service boundary before the repository runs, and a hidden button, a role claim or a signed-out tab proves nothing until that check denies.

Learning objectives

By the end of this module you'll be able to:

- ✓ Explain the OpenID Connect authorization code flow with PKCE, state and nonce, and what the ID token and the access token are each for.
- ✓ Configure the ASP.NET Core cookie and OpenID Connect handlers in Startup.cs before app.UseWisej() with an exact registered callback, minimal scopes, a pinned authority and secrets loaded at runtime.
- ✓ Map the validated issuer and subject to a local AccessOps account and tenant, establish the Wisej.NET session principal from trusted data, and keep a local-login fallback for service accounts.
- ✓ Reject a forged state, a replayed callback and an unapproved tenant before any grant is created, and sign out of the local and the provider session as separate, verified outcomes.

Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.

> **Hands-on lab**
>
> Add partner federation to AccessOps: register the app with a test OpenID Connect provider and tenant, configure the cookie and OpenID Connect handlers in Startup.cs before app.UseWisej() for the authorization code flow with PKCE, an exact HTTPS callback, the canonical origin preserved through the proxy and the client secret loaded at runtime, and hand the validated identity to the Wisej.NET session where an account mapper looks up the issuer and subject, assigns the tenant and roles from AccessOps data and creates exactly one login grant. Keep the Module 4 local login as the fallback for service accounts, run one successful sign-in, then use the disposable fixtures to change the state, replay a consumed callback, sign in from an unapproved tenant and simulate a provider denial, showing no app grant is created in any of the four, and finish with a sign-out that revokes the local grant and ends the provider session.

Starting point

Open AccessOps as you left it after Module 4: the partner route's sign-in page with its password form, the `GrantStore` that every export checks, and `ApplicationPrincipal` establishing the session principal on a page load. Use a non-production provider registration and a disposable test tenant (a local test identity provider is fine), and the reserved fixtures F06, F07 and F08 for the negative cases.

Suggested approach

- ✓ Register AccessOps at the test provider with one exact HTTPS callback ending in `/signin-oidc`, the scopes `openid profile email` only, and a client secret you set as `Oidc__ClientSecret` in the environment or user secrets, never in `appsettings.json`.
- ✓ Write `AddAccessOpsOpenIdConnect()` in `Startup.cs`: cookie scheme plus OpenID Connect with `ResponseType` code, `UsePkce`, `SaveTokens` false and a pinned `Authority`; map `/account/signin` and `/account/signout` and call `app.UseAuthentication()` before `app.UseWisej()`.
- ✓ In `OnTokenValidated`, park the issuer and subject in `ExternalHandoffStore`, set the HttpOnly `AccessOps.Handoff` cookie and call `HandleResponse()`; on the page load, let `TryCompleteOnPageLoad()` redeem it once, run `ExternalIdentityMapper`, create one grant and call `ApplicationPrincipal.Establish`.
- ✓ Run the success path, then F06 (state from another attempt), F07 (the same callback twice), F08 (an unapproved tenant) and a provider denial, counting grants after each one and logging only a redacted correlation and the allowlisted reason.
- ✓ Sign out, retry an export from a second open tab, then sign in again and record whether the provider asks for credentials; draw the sanitized sequence diagram from what you observed.

Required deliverables

- ✓ Cookie and OpenID Connect handler configuration in Startup.cs with secrets out of source

- ✓ Issuer and subject account mapper creating one grant per sign-in
- ✓ Local-login fallback for service accounts
- ✓ Five transaction outcomes: one success and four rejections with no grant
- ✓ Sanitized sequence diagram and sign-out covering both sessions

Student review questions

- ✓ An attacker copies a partner's callback URL, code and state included, and opens it in their own browser a few seconds later. Which two independent checks refuse it, and what would still protect AccessOps if the code had not yet been redeemed?
- ✓ A partner's email changes at their provider and a new hire there receives the old address. What does your mapper do on each person's next sign-in, and why would an email-keyed mapper get it wrong?
- ✓ After sign-out the next user on a shared machine is signed straight back in without a password. Which of your two sign-out outcomes failed, and what evidence would have caught it before release?

Security habit

Keep the identities apart and decide on the server. The host principal, the Wisej.NET application principal and the effective Windows token are three different facts, and none of them is a permission. Every protected read checks the current grant, the tenant and the record at the service boundary before the repository runs, and a hidden button, a role claim or a signed-out tab proves nothing until that check denies.