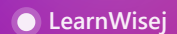


# OAuth and OpenID Connect Integration

Lesson Guide · Lab / Exam Guide



## Lesson Guide

Delegating login to an identity provider, the authorization code flow, redirect handling in a Wisej.NET app, token validation and mapping claims to the SessionContext. What this means for a developer securing a production Wisej.NET app — and how to read this module.



### Production focus

Delegating login to an identity provider, the authorization code flow, redirect handling in a Wisej.NET app, token validation and mapping claims to the SessionContext.

## 1. The production problem

AccessOps can already verify a password of its own, but the organisation running it does not want a second directory: staff sign in to everything else with the corporate identity provider, and joiners, leavers and multi-factor policy are administered in one place. Delegating login removes the password from AccessOps entirely, and replaces it with a different obligation — the portal is now handed an identity by a third party, through the user's own browser, and has to prove that identity is genuine, unmodified and meant for it before a single department record is shown.

## 2. Key concepts

| Concept                        | Why it matters                                                                                                                                                                                               |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| authorization code flow        | The OpenID Connect flow in which the browser only ever carries a short-lived, single-use code, while the tokens are fetched by your server over a direct HTTPS call the browser never sees.                  |
| PKCE                           | A random code verifier generated per sign-in, sent to the authorize endpoint only as its SHA-256 hash and revealed at the token endpoint, so an intercepted code cannot be redeemed by anyone else.          |
| ID token                       | A signed JWT asserting that this provider authenticated this subject, for this client, at this time. It is the only token AccessOps consults to decide who the user is.                                      |
| access and refresh tokens      | The access token is a bearer credential for calling somebody else's API; the refresh token buys replacements in the back channel. Neither is evidence of identity for your own app.                          |
| state and nonce                | Two random values you generate per sign-in: state ties the callback to the request you actually started, nonce ties the returned ID token to that same request so an older token cannot be replayed into it. |
| redirect URI                   | The exact, pre-registered URL the provider returns the browser to. In AccessOps it is a small server-side endpoint rather than a Wisej.NET window, because it has to run before any UI exists.               |
| claim mapping and provisioning | Turning provider claims into an AccessOps employee: the issuer and subject pair identifies the account, and a local row is created on first sign-in carrying no permissions until someone grants them.       |

## 3. Lesson narrative

### Three tokens and one round trip

Everything starts at the provider's discovery document, the well-known configuration URL that lists the authorize, token and end-session endpoints and the JWKS address of its signing keys. AccessOps reads it once at start-up and caches it, because every later step depends on the exact values it publishes — the issuer string above all, which has to match character for character.

The sign-in itself is two half-conversations. In the front channel the browser is sent to the authorize endpoint with the client id, the registered redirect URI, `response_type=code`, the scopes AccessOps needs (`openid profile email`), a fresh state, a fresh nonce and a code challenge with `code_challenge_method=S256`. The user authenticates at the provider — AccessOps never sees that page and never sees the password — and the browser returns to the redirect URI carrying only a code and the state. In the back channel your server posts that code to the token endpoint together with the code verifier and the client credential, and receives the tokens in the response body.

Each token has exactly one job. The ID token is the authentication receipt and the only thing that answers who signed in. The access token is a credential for a resource API such as the directory graph, and must

never be inspected to decide identity in your own app; it may be opaque, and it is addressed to a different audience. The refresh token buys new tokens without another trip through the browser. PKCE is what makes the split safe: without the verifier, a code stolen from a browser history, a referrer header or a compromised proxy is worthless to whoever holds it.

## Leaving the Wisej.NET session and finding your way back

In AccessOps the redirect starts from server code with `Application.Navigate`, pointing the browser at the authorize URL you assembled. The browser has now left your application, and it will come back to the redirect URI — which cannot be a Wisej.NET window, because it has to answer an HTTP request that arrives before any window exists. The callback here is a small server-side endpoint hosted in the same application: a custom Wisej.NET HTTP handler, or a companion ASP.NET Core endpoint on the same origin. Its whole job is the token exchange, the validation and one redirect.

The thread back to the user's context is the state value. Before navigating, AccessOps creates a random handle and stores a pending-sign-in record against it on the server: the code verifier, the nonce, the moment it was created, and where the user was trying to go. Only the handle travels in the query string. The callback looks it up, rejects anything unknown, expired or already used, and deletes the record so it cannot be redeemed twice. Nothing else in that query string is trusted, because the entire front channel is under the browser's control.

Once the tokens are validated the handler redirects back to the AccessOps start URL with a one-time ticket; the application reads that ticket at start-up from the request's query string, redeems it exactly once against the same store, and builds the SessionContext from the record rather than from anything the browser carried. That indirection keeps the session honest whether the original one is still alive or the return trip started a fresh one. Opening the provider in a second browser window and signalling the waiting session with `Application.Update` is an equally valid shape; the rule does not change.

## Validating the ID token, then mapping it

A JWT is signed, not secret: anyone can decode one, so reading claims before validating is the classic mistake. AccessOps checks five things with `System.IdentityModel.Tokens.Jwt` and the `Microsoft.IdentityModel` packages: the signature against the provider's published keys, the issuer against the discovery issuer, the audience against your own client id, the lifetime with a couple of minutes of clock skew, and the nonce against the value stored with the pending record. The keys come from the JWKS endpoint through a configuration manager that refreshes them, so a key rollover at the provider does not become an outage at three in the morning.

```

var parameters = new TokenValidationParameters
{
    ValidIssuer      = config.Issuer,          // exactly the discovery issuer
    ValidAudience    = _options.ClientId,     // this client, nobody else
    IssuerSigningKeys = config.SigningKeys,    // refreshed from the JWKS
    ValidateLifetime  = true,
    ClockSkew         = TimeSpan.FromMinutes(2)
};

var principal = new JwtSecurityTokenHandler()
    .ValidateToken(idToken, parameters, out var validated);

if ((string)((JwtSecurityToken)validated).Payload["nonce"] != pending.Nonce)
    throw new SecurityTokenException("Nonce mismatch - sign-in rejected.");

```

Mapping comes after that, and the identifier is the `sub` claim paired with the issuer — never the email address, which can be renamed or reassigned to a new joiner. `name`, `preferred_username` and `email` are display material only. Group claims from the directory are recorded as evidence, but the permissions the portal enforces are still resolved by its own `PermissionService` into the `SessionContext`: the provider says who the person is, `AccessOps` decides what they may do here.

When no employee matches, first-login provisioning creates one in the same transaction that writes the login audit record, with a placeholder department and no elevated rights. Store the issuer alongside the subject, because subject values are only unique within one issuer, and a second provider added later would otherwise collide with the first.

### Expiry, refresh and signing out of both sessions

An ID token proves a login happened at an instant; it is not a session and should not be treated as one. The `AccessOps` session lifetime and idle timeout from Module 2 continue to govern. Keep the token expiry alongside the tokens, and when a downstream call needs an access token that has expired, exchange the refresh token in the back channel and replace both before making the call. Refresh tokens are credentials in their own right: encrypt them at rest, scope them to one user and client, and where the provider rotates them, treat the reuse of a retired one as a compromise signal rather than a retry.

Sign-out is the half that is routinely forgotten. Disposing the `SessionContext` and abandoning the `Wisej.NET` session ends the local session only; the provider session in the same browser survives, so the next sign-in completes silently and the user who logged out on a shared machine has not. Single sign-out also sends the browser to the end-session endpoint with the ID token as the hint and a registered post-logout redirect URI. Handle the other direction too where your provider supports it: a logout notification arriving from the provider should end the matching `AccessOps` session.

- Local sign-out disposes the `SessionContext` and abandons the session.
- The end-session redirect closes the provider session in the same browser.
- The password path from Module 2 survives only for named service accounts, rate limited and audited.



### Security habit

The browser is never trusted. Every decision that matters — who the user is, what they may do, which record they may see — is made on the server in a service, and the UI only reflects it. If a rule exists only as a hidden button, it does not exist.



### Learning objectives

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

- ✓ Explain the authorization code flow with PKCE and what each token is for.
- ✓ Redirect a Wisej.NET session to an identity provider and handle the callback without losing the session.
- ✓ Validate the ID token, map provider claims to the SessionContext and provision the local user on first login.
- ✓ Handle token expiry, refresh and single sign-out.

## Lab / Exam Guide

---

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



### Hands-on lab

Add external sign-in to AccessOps: an OpenID Connect client using the authorization code flow with PKCE against a test identity provider, a callback handler that validates the ID token and its nonce, a claim mapping that fills the SessionContext and provisions a local employee record on first login, a fallback to local login for service accounts, and sign-out that ends both the local and the provider session.

## Starting point

Open the AccessOps solution as you left it after Module 3: the login page verifies a salted password hash, the SessionContext carries a verified identity, and the PermissionService resolves roles and claims into permissions that every DocumentService and ApprovalService method checks. You will need a test identity provider you control — a local Keycloak realm, a developer tenant, or any conformant server — with a confidential client registered for AccessOps and the callback URL of your development site added to its allowed redirect URIs.

## Suggested approach

- ✓ Register the AccessOps client at the provider, then read its discovery document at start-up and cache the authorize, token, end-session and JWKS addresses instead of hard-coding them.
- ✓ Add a `Sign in with your work account` button that builds the authorize URL with state, nonce and an S256 code challenge, parks a pending-sign-in record on the server keyed by state, and calls `Application.Navigate`.
- ✓ Write the callback endpoint as a Wisej.NET HTTP handler or a companion ASP.NET Core endpoint: match the state, exchange the code and verifier for tokens over a back-channel HTTPS call, and validate the ID token's signature, issuer, audience, lifetime and nonce.
- ✓ Map the issuer and subject pair to an employee record, creating one with no elevated permissions on first sign-in, then hand the result back to the application through a one-time ticket and build the SessionContext from it.
- ✓ Implement sign-out in both directions and keep the password path alive only for named service accounts, then demonstrate a rejected callback with a tampered state and a replayed ID token.

## Required deliverables

- ✓ OIDC client configuration with PKCE
- ✓ Callback handling with token and nonce validation
- ✓ Claim mapping and first-login provisioning
- ✓ Local login fallback for service accounts
- ✓ Sign-out covering both sessions

## Student review questions

- ✓ An attacker captures the authorization code out of a browser history on a shared machine. What stops them exchanging it for tokens, and what would have failed to stop them without PKCE?
- ✓ Your provider issues group claims that look exactly like the AccessOps permission names. Why does the PermissionService still resolve permissions itself instead of copying those claims straight into the SessionContext?
- ✓ A user clicks Sign out, closes nothing, and immediately clicks Sign in again — and lands back inside AccessOps without typing anything. Which session survived, and which redirect was missing?



### Security habit

The browser is never trusted. Every decision that matters — who the user is, what they may do, which record they may see — is made on the server in a service, and the UI only reflects it. If a rule exists only as a hidden button, it does not exist.