SECURITY & AUTH PATTERNS · MODULE 4

Custom Sign In and Session Security

Lesson Guide · Lab / Exam Guide

LearnWisej

Lesson Guide

Owning the sign-in transaction, establishing the application principal and rotating the session ID, issuing host cookies only over HTTP, logout as revocation of a server-side grant, and closing request forgery, WebSocket origin and redirect paths. 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

Owning the sign-in transaction, establishing the application principal and rotating the session ID, issuing host cookies only over HTTP, logout as revocation of a server-side grant, and closing request forgery, WebSocket origin and redirect paths.

1. The production problem

Partners reach AccessOps over the public HTTPS route, where there is no Windows challenge to lean on: the application owns the whole sign-in, from the password box to the moment the console replaces the login view. Every shortcut becomes an attack path: a lockout that reveals which names exist, a session ID that survives sign-in, or a sign-out that hides a button while a second tab keeps exporting. This module builds the partner sign-in and proves that a revoked grant is denied everywhere.

2. Key concepts

Concept	Why it matters
Sign-in transaction	The server-side sequence throttle → verify → resolve account and tenant → apply policy → create grant → establish principal, where any failed step ends the attempt with no state left behind.
Login throttle	Counters of recent failures per account and per client address that refuse further attempts for a lockout period, kept in the login service because a Wisej click cannot be rate-limited by ASP.NET Core middleware.
Generic failure	One identical sentence for a wrong password, an unknown name, a disabled account and a lockout, with comparable timing, so the screen never confirms that an account exists.
Application principal	The `ClaimsPrincipal` assigned to `Application.User` for this Wisej session; it is session state, not a host cookie and not a Windows token.
Session ID rotation	`Application.RefreshSessionId()` gives the session a new ID without losing its state, so an ID that existed before sign-in is worthless after it.
Host authentication cookie	An ASP.NET Core cookie ticket issued with `Secure` and `HttpOnly` by an HTTPS endpoint, needed only when ordinary HTTP endpoints must recognise the user.
Login grant	A server-side record with an opaque random ID, account key, tenant, expiry and revocation flag, checked by every protected operation; revoking it is what logout means.

3. Lesson narrative

Own the sign-in transaction

The AccessOps login view on the anonymous public route does exactly one thing in `btnSignIn_Click`: it reads the two boxes, clears the password box and calls `LoginService.SignIn`. Credential verification belongs to a supported identity system — ASP.NET Core Identity, a directory, or, for local accounts, a framework derivation such as PBKDF2 through `Rfc2898DeriveBytes.Pbkdf2` with a per-account salt and a constant-time comparison. When the user name is unknown, the service still verifies against a dummy credential so that "no such user" costs the same time as "wrong password".

Throttling lives in that same service. The AccessOps `LoginThrottle` locks an account for 15 minutes after five failures inside a 15-minute window and an address after twenty, because credential stuffing tries many names from one place. ASP.NET Core rate limiting works on endpoints beside `app.UseWisej()` but cannot see a Wisej click, which arrives over the session's WebSocket. Read the address from `Application.UserHostAddress` and remember its limits: one browser can appear as both `::1` and `127.0.0.1`, and behind a proxy the real client comes only from forwarded headers you trust. Process-memory counters are correct for one server only.

Every refusal — bad password, unknown name, disabled account, lockout — shows one sentence: "That user name and password combination is not valid." The real reason goes only to the login audit table. The

password itself goes nowhere: not into `Application.Session`, not into a log message, not into an exception text.

Establish the principal only after policy passes

A verified password proves knowledge of a secret, not the right to use AccessOps. The service resolves the local account and its tenant, refuses a disabled account or an unapproved tenant with the same generic sentence, creates a login grant and only then builds the principal. The pack's `ApplicationPrincipal.Establish` helper turns account, tenant, grant and roles into claims and assigns `Application.User = new ClaimsPrincipal(new ClaimsIdentity(claims, "Portal"))`. That principal is session state: it issues no cookie, ASP.NET Core middleware on other requests never sees it, and it creates no Windows token.

The order matters, and on Wisej.NET 4.1.7 so does the request type. Calling `Application.RefreshSessionId()` from a click — a WebSocket event — switches the ID on the server, but the browser never adopts it; its next request starts a brand-new session and the sign-in is lost. Called during a page load the browser adopts the new ID, which it keeps in the tab's sessionStorage. So AccessOps verifies on the click, parks the verified result under a pending key, calls `Application.Reload()`, and finishes on the load:

```
// Program.Main: Application.ApplicationRefresh += (s, e) => OnPageLoad();
private static void OnPageLoad()
{
    if (Application.Session["PendingSignIn"] is not VerifiedSignIn p)
        return;

    Application.Session["PendingSignIn"] = null;
    Application.RefreshSessionId();          // new ID first (HTTP load)
    ApplicationPrincipal.Establish(p.AccountId, p.TenantId, p.GrantId, p.Roles);

    var login = Application.MainPage;
    Application.MainPage = new ConsolePage(); // replace the login view
    login?.Dispose();
}
```

The course's copy of `ApplicationPrincipal` therefore assigns `Application.User` without rotating; the load rotates once, before the identity exists.

Cookies over HTTP, logout as revocation

A WebSocket frame has no response headers, so a click cannot deliver a `Set-Cookie` with `HttpOnly`. `Application.Cookies` can write HttpOnly cookies only during `Program.Main`, `ApplicationStart`, `ApplicationRefresh` or when `Application.IsWebSocket` is false, and `Application.Browser.CookieStorage` is written by JavaScript, which can never set HttpOnly. When ordinary HTTP endpoints must recognise the partner, AccessOps maps an HTTPS endpoint before `app.UseWisej()` that redeems a single-use, short-lived handle created by the session, calls `HttpContext.SignInAsync` with the `"Cookies"` scheme, and lets the cookie handler emit `Secure`, `HttpOnly` and `SameSite=Strict`. That ticket is not Wisej's: the session ID lives in sessionStorage, and Wisej 4.1.7 writes one HttpOnly request-validation cookie of its own that you leave alone.

Logout is not a hidden button or a deleted cookie. The `GrantStore` holds opaque, expiring grants; `ExportService` checks the grant from the principal's `grant` claim, the tenant and the record before the repository runs. Sign-out revokes the grant first, audits, expires the host cookie through the sign-out endpoint, clears the session identity and calls `Application.Exit()`. Setting `Application.User` to `null` is not enough — the getter falls back to the host principal, and under IIS Windows authentication a reload signs straight back in. A second tab still shows Export; its next click is denied with zero repository reads (fixture F05). `CookieAuthenticationEvents.OnValidatePrincipal` applies the same grant check to host requests.

Close the cross-site paths

The browser attaches the host cookie to any request, including one started by another site, so every state-changing POST that trusts it — sign-in, sign-out, the cookie hand-off — validates an antiforgery token. The Wisej session ID is in sessionStorage, which a foreign page cannot make the browser send, but the WebSocket handshake still carries cookies and is not covered by CORS: a small middleware ahead of `app.UseWisej()` refuses an upgrade whose `Origin` is not an AccessOps origin, and the ingress repeats the rule.

Continuation URLs are input. A `returnUrl` from `Application.QueryString` or an endpoint's query is accepted only when it is a local path that does not start with `//` or `/\`, or an entry in an explicit server allowlist; anything else becomes `/`. Derive scheme and host only after `app.UseForwardedHeaders()` is configured for known proxies. Fixture F11 — a cross-origin POST or upgrade — must end with no state change.

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:

- ✓ Implement a sign-in that verifies credentials with a supported identity system, throttles attempts, returns generic failures and keeps passwords out of logs and session state.
- ✓ Establish the Wisej.NET application principal only after account and tenant policy pass, rotate the session ID with Application.RefreshSessionId and replace the login view.
- ✓ Issue and expire Secure, HttpOnly host cookies from a dedicated HTTPS endpoint rather than a WebSocket event, and treat logout as revocation of a server-side grant checked by every protected operation.
- ✓ Close cross-site paths with antiforgery on state-changing posts, WebSocket origin checks and an allowlist for continuation URLs.

Lab / Exam Guide

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

Hands-on lab

Give the AccessOps partner route its own sign-in: a login view on the anonymous public route that verifies a fixture identity through a supported verifier or a training mock that never ships, rejects a bad credential and a disabled account with the same generic message, resolves the local account and tenant, establishes the principal with ApplicationPrincipal and rotates the session ID. Back it with a GrantStore of opaque, expiring, revocable login grants that the export service checks before every read, add an HTTPS sign-in and sign-out endpoint for the host cookie, then open two browser profiles, revoke the grant with the second tab open and retry the export in both tabs and after a refresh, and test a forged post, an off-origin WebSocket upgrade and an external continuation URL.

Starting point

Open the AccessOps solution from Module 1 with its tenant A and B fixtures, the read-counting repository and the `DocumentService` gate, and add a `LoginPage` on the anonymous public route. Run it on an HTTPS hostname (or `https://localhost`) and keep two browser profiles ready for the revocation test.

Suggested approach

- ✓ Write `LoginService.SignIn` with `LoginThrottle`, a dummy-credential check for unknown names and one generic failure sentence; record outcomes in the audit table and confirm no password value reaches a log or `Application.Session`.
- ✓ Add `GrantStore` (opaque random ID, account, tenant, expiry, revoked flag) and make `ExportService` check the grant, tenant and record before the repository runs.
- ✓ Park the verified sign-in on the click, call `Application.Reload()`, and on `ApplicationRefresh` call `Application.RefreshSessionId()`, then `ApplicationPrincipal.Establish`, then swap the login view for the console; log the old and new IDs as a boolean "rotated".
- ✓ Map the HTTPS cookie hand-off and sign-out endpoints before `app.UseWisej()` with antiforgery, an `Origin` check on WebSocket upgrades and a local-path check for `returnUrl`; capture the `Set-Cookie` attributes with the value redacted.
- ✓ Revoke the grant from tab 1, retry Export in tab 2 and after a refresh, then run the forged post, wrong-origin upgrade and external continuation URL tests and draw the auth-state transition chart.

Required deliverables

- ✓ Login view with generic failures, throttling and disabled-account handling
- ✓ Principal establishment with session ID rotation
- ✓ HTTPS cookie endpoint with redacted Set-Cookie attributes
- ✓ GrantStore with revocation enforced across open tabs
- ✓ Auth-state transition chart and forged post, wrong origin and continuation URL tests

Student review questions

- ✓ Your first draft called `Application.RefreshSessionId()` inside `btnSignIn_Click` and the console appeared, then vanished on the next click. What did the browser do with the new ID, and why does a page load behave differently?
- ✓ After sign-out, tab 2 still shows the Export button and still holds a principal with a role claim. Which single check makes its next export fail, and why would deleting the host cookie alone not achieve it?
- ✓ The Wisej session ID is not in a cookie, yet AccessOps still needs antiforgery and an origin check. Which requests carry ambient credentials that another site could trigger?

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.