SECURITY & AUTH PATTERNS · MODULE 1

Identity Boundaries in Wisej.NET

Lesson Guide · Lab / Exam Guide

LearnWisej

Lesson Guide

The three identities of a Wisej.NET request — host principal, application principal and effective Windows token — Application.User and the session lifecycle, and a resource-level deny at the service boundary that survives role revocation. 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

The three identities of a Wisej.NET request — host principal, application principal and effective Windows token — Application.User and the session lifecycle, and a resource-level deny at the service boundary that survives role revocation.

1. The production problem

AccessOps serves staff through an intranet Windows entry point and partners through a public HTTPS entry point, and both reach the same confidential documents. Every request carries several identities at once, and a bug that confuses them shows up as one tenant reading another tenant's record, or a revoked reader who can still export. This module pulls the identities apart and puts the decision where it can be proven: at the service boundary, before the repository runs.

2. Key concepts

Concept	Why it matters
host request principal	What the host authenticated for the request or WebSocket upgrade — `HttpContext.User` on ASP.NET Core; anonymous on a plain Kestrel run.
application principal	The subject your sign-in assigns to the Wisej.NET session after a trusted verifier returns an account: account ID, tenant, grant ID.
effective Windows identity	The token the operating system checks when the process opens a file — the application pool or process account unless code impersonates.
session override	A value assigned to `Application.User` in the current session; it wins over the request principal until it is set back to `null`.
grant	A revocable, expiring server-side record that an account may read in one tenant; every protected operation reads its current state.
service boundary	The one method every path — click, download, queued job — crosses before the repository, so one deny covers them all.
read-counting repository	A test repository that counts its reads, so a deny is proven at zero repository calls rather than inferred from an empty screen.

3. Lesson narrative

Three identities, and none of them is a permission

Follow `CONTOSO\reader1` on the intranet route. IIS authenticates the request, so the host principal is a Windows identity, yet when `DocumentService` opens a file the operating system checks the application pool's token, not reader1's, unless code impersonates (Module 3). A federated partner (Module 5) has a valid application principal and no Windows token at all. The nightly export has no signed-in user, yet the process account can read the whole document folder.

Record them as three separate observations and never let one stand in for another. None is an authorization decision: the host principal says who connected, a role claim says what was true at sign-in, the process token says what the operating system allows. AccessOps must answer a narrower question on every protected read: does the current grant allow this account to read this record in this tenant, right now?

`Visible` and `Enabled` describe the screen. Wisej.NET does verify on the server that a click comes from an enabled, visible control, but data bound to a hidden grid column still reaches the browser, and the same document is reachable from a handler, a download path and a job. The rule belongs in the service they all call.

How Application.User and IsAuthenticated resolve

`Application.User` is a static `IPrincipal` with a two-step getter. If a value was assigned in this session, it returns that session override; otherwise it returns the request principal — on ASP.NET Core the

`HttpContext.User` captured when the request or WebSocket upgrade arrived: a `WindowsIdentity` under IIS Windows Authentication, an unauthenticated principal on a developer Kestrel run. `Application.IsAuthenticated` is simply `User?.Identity?.IsAuthenticated == true`: it knows nothing about tenant, role or ownership. Setting `Application.User = null` removes the override and falls back to the host principal, so "signing out" a Windows user that way leaves them authenticated.

Assign the override only after a trusted verifier returns an account; a `ClaimsIdentity` built from a typed user name lets the browser choose its identity. The lab's `ApplicationPrincipal` helper builds claims from the verified account — immutable account ID, tenant, grant ID, roles — under the authentication type `"Portal"`. Key on that ID, never on a display name or email.

```
// Runs on a page load, after the test verifier returned an account
internal static void Establish(string accountId, string tenantId,
    string grantId, IEnumerable<string> roles)
{
    var claims = new List<Claim> {
        new Claim(ClaimTypes.NameIdentifier, accountId),
        new Claim("tenant", tenantId),
        new Claim("grant", grantId) };
    claims.AddRange(roles.Select(r => new Claim(ClaimTypes.Role, r)));

    Application.RefreshSessionId();   // new ID first, on the load
    Application.User = new ClaimsPrincipal(new ClaimsIdentity(claims, "Portal"));
}
```

`Application.UserIdentity` exists only in the .NET Framework (net48) build, where it is the System.Web logon identity; the net8.0 build a .NET 10 AccessOps references does not declare it. On .NET 10, read the Windows principal through `Application.User` and check that its `Identity` is a `WindowsIdentity` before relying on it.

The session lifecycle and work outside it

The first HTTP load builds the UI, later events arrive over the WebSocket, and each session transition is its own event. `ApplicationStart` fires for a new session after `Program.Main`; `ApplicationRefresh` fires on a browser reload or URL change in the same session; `SessionTimeout` announces idle expiry (set `Handled` to suppress the default prolong dialog); `ApplicationExit` fires when the session is terminated or disposed, the only event raised on expiration. Use it for cleanup, never as revocation: that lives in the grant store.

`Application.RefreshSessionId` issues a new session ID without losing the session, closing fixation after sign-in — but only on a page load. From a WebSocket click the server switches IDs, the browser never adopts the new one, and its next request starts a fresh session. So AccessOps verifies on the click, parks the account under a pending key, calls `Application.Reload()`, and in `ApplicationRefresh` rotates the ID first, then establishes the principal. Redact `Application.SessionId` in traces.

Outside the request, ambient identity fails. `Application.StartTask` restores the session on its pool thread; a raw `Task.Run` or timer callback has none, and `Application.Update` only restores one from a stored `Application.Current`. So the background export receives an explicit subject — account ID, tenant,

grant ID, record ID — and `DocumentService` rechecks the grant when the job starts. If two browser profiles show the same account, look for a static field holding the principal.

The deny at the service boundary

`DocumentService.Read(subject, tenant, recordId)` loads the current grant and refuses when it is missing, revoked, expired or lacks read access, or when its tenant differs from the requested one; only then does it call the repository. A role check is not enough: tenant A's Reader really is a Reader, record 17 exists in both tenants, and `IsInRole("Reader")` lets the horizontal read through.

The proof is a number. In fixture F02, tenant A's Reader requests tenant B's record 17: deny, and the read count does not move. Then revoke the Reader grant from the admin console and click Export on the screen that still shows the button: the service reads the current grant and denies before any file opens. Each decision writes an audit event with an opaque subject, action, classification, decision and correlation ID — never cookies, session IDs or tokens.

The self-contained `PolicyHarness` console (net8.0 in the pack; net10.0 works the same) applies the rule without Wisej.NET, IIS or an identity provider: one allowed read, then tenant-boundary, missing-role, unknown, expired and revoked denies at zero repository calls. The harness proves the policy anywhere; the app proves every path uses it.

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:

- ✓ Distinguish the application subject, the host request principal and the effective Windows identity of an AccessOps request, and explain why none of them is an authorization decision by itself.
- ✓ Explain how Application.User and IsAuthenticated resolve, including the session override and the fallback to the host principal, and why Application.UserIdentity exists only on .NET Framework.
- ✓ Trace the session lifecycle — ApplicationStart, ApplicationRefresh, SessionTimeout, ApplicationExit and Application.RefreshSessionId — and pass an explicit subject to work that runs outside the session.
- ✓ Enforce a tenant and record check at the service boundary and prove the deny happens before the repository reads, even after the role is revoked on an open screen.

Lab / Exam Guide

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

Hands-on lab

Create the AccessOps solution with tenants A and B each holding record 17 with distinct contents, a Reader account in tenant A and an unassigned account. Build a DocumentService that checks the current grant, tenant and record before calling a repository that counts its reads, establish the session principal from a trusted test verifier with ApplicationPrincipal, and add a diagnostics view (kept off public routes and removed before packaging) that shows the authentication type, the authenticated flag, a redacted subject and the effective OS identity. Trace startup, a click, a browser refresh and a background job in two browser profiles, request tenant B's record 17 as tenant A's Reader, revoke the Reader grant between page load and Export on an already open screen, and run the self-contained PolicyHarness to show the same allow and deny decisions without Wisej, IIS or an identity provider.

Starting point

A new Wisej.NET web application named AccessOps on `net10.0` with the `Wisej-4` 4.1.7 package, run on Kestrel on your own machine, plus the pack's `PolicyHarness` console project added to the same solution. This module needs no IIS, Windows domain or identity provider: every step runs locally, and the "effective OS identity" you record is simply the account the `dotnet` process runs as.

Suggested approach

- ✓ Seed the fixtures first: record 17 in tenant A and in tenant B with different contents, a Reader grant for tenant A and an account with no grant, and a repository whose `ReadCount` you can show on screen.
- ✓ Write `DocumentService.Read(subject, tenant, recordId)` so that it loads the current grant and refuses a missing, revoked or expired grant, a missing read right and a tenant mismatch before it touches the repository.
- ✓ Wire sign-in to a training verifier that never ships: verify on the click, park the account, call `Application.Reload()`, and on the load call `ApplicationPrincipal.Establish`, which rotates the session ID before setting `Application.User`.
- ✓ Add the diagnostics view behind a debug-only compile switch, then fill the four-column matrix for startup, a click, a refresh and a background job in two browser profiles, including a job started with `Application.StartTask` and one started with `Task.Run` that receives an explicit subject.
- ✓ Run the denies — tenant B's record 17 as tenant A's Reader, then Export after revoking the grant on an open screen — check that `ReadCount` did not move, run `PolicyHarness`, and confirm the published build contains no diagnostics view.

Required deliverables

- ✓ AccessOps solution with tenant A and B fixtures and a read-counting repository
- ✓ DocumentService resource gate with grant, tenant and record checks

- ✓ Identity diagnostics view, removed from the packaged build
- ✓ Four-column identity matrix for request, session, operation and outcome
- ✓ PolicyHarness run with one allowed read and two denies at zero repository calls

Student review questions

- ✓ Your diagnostics view shows the same effective OS identity for both browser profiles while the application subjects differ. What does that tell you about which identity the file system will check, and what would change on IIS with Windows Authentication?
- ✓ After the Reader grant is revoked, the Export button is still on the open screen and `Application.User` still carries the Reader role claim. Which single read in your code makes the deny happen, and why is the role claim not that read?
- ✓ Your background export runs on `Task.Run` an hour after the click. What exactly does it receive, what does it recheck when it starts, and what would go wrong if it read `Application.User` instead?

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.