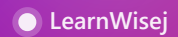


Authentication and Session Identity

Lesson Guide · Lab / Exam Guide



Lesson Guide

Login flows, password handling, a SessionContext with a verified identity, timeouts, logout, remember-me and session fixation. What this means for a developer securing a production Wisej.NET app — and how to read this module.



Production focus

Login flows, password handling, a SessionContext with a verified identity, timeouts, logout, remember-me and session fixation.

1. The production problem

AccessOps holds employee records, confidential documents and approval requests, so the first screen anyone reaches is also the first control that matters. Everything the rest of this course enforces — permissions, record-level rules, audit trails — rests on a single assumption: that the server knows who is on the other end of this session and can prove it. Module 1 drew the trust boundaries; this module puts a verified identity inside the session boundary and keeps it from being guessed, borrowed or outlived.

2. Key concepts

Concept	Why it matters
salted slow hash	A one-way derivation of the password with a random per-user salt and a deliberately expensive iteration count, so a stolen table cannot be reversed and every guess costs real work.
constant-time comparison	Comparing two hashes in a fixed number of operations with <code>CryptographicOperations.FixedTimeEquals</code> , so the time the server takes to answer never reveals how many bytes of a guess were right.
SessionContext	The object created only after a password is verified, holding the employee id, department, roles and sign-in details, and stored in <code>Application.Session</code> as the one place any code asks who the user is.
session timeout	The idle period after which Wisej.NET disposes the server-side session and everything it holds, configured for the application in <code>Default.json</code> rather than guessed per screen.
session fixation	An attack in which the identifier that carried the anonymous login page also carries the authenticated session afterwards, so an identifier the attacker planted becomes a logged-in one.
login throttling	Counting failures per account and per client address and refusing further attempts for a lockout period, which is what turns an unlimited guessing machine into a few tries an hour.
login audit record	One row per attempt with the user name tried, the time, the client address and the real outcome — the place the specific failure reason is written, since the screen only ever shows a generic one.

3. Lesson narrative

Verifying a password you never keep

AccessOps stores no passwords. For each employee it stores three values: the algorithm and iteration count used, a random salt from a cryptographic generator, and the derived bytes. Signing in runs the same derivation over the password just typed, using that employee's salt and iteration count, and compares the result with the stored bytes. Nothing is decrypted, because nothing was encrypted — a hash goes one way, and the salt makes two employees who chose the same password produce completely different stored values.

Two implementations are appropriate here. PBKDF2 through `Rfc2898DeriveBytes` gives explicit control: SHA-256, a 16-byte salt, a 32-byte output and an iteration count high enough that one verification takes a noticeable fraction of a second. ASP.NET Core Identity's `PasswordHasher` instead packs the algorithm marker, iteration count, salt and hash into one string and returns a `PasswordVerificationResult` whose `SuccessRehashNeeded` value tells you to re-derive with today's parameters. Either way the cost is the point: every attacker guess must pay for a full derivation.

The password itself lives for one method call. It is read from the login `TextBox`, passed to `PasswordService.Verify`, and forgotten: never a field on the window, never a value in the audit row, never part of an exception message, and never a log line — not even truncated.

```
public bool Verify(string password, StoredCredential stored)
{
    var candidate = Rfc2898DeriveBytes.Pbkdf2(
        Encoding.UTF8.GetBytes(password),
        stored.Salt,
        stored.Iterations,
        HashAlgorithmName.SHA256,
        stored.Hash.Length);

    return CryptographicOperations.FixedTimeEquals(candidate, stored.Hash);
}
```

One verified identity, and only one

When `Verify` returns true, the login service builds a `SessionContext`: the employee id, the display name, the department id, whether this person manages anyone, the roles resolved from the database at that moment, the sign-in time and the client address. It is stored once in `Application.Session`, which is per browser session, so nothing crosses between the operations manager and the payroll clerk. A static field would be shared by every session in the process.

From then on it is the only source of truth. Screens keep no copy of the user id, and services take no caller id as a parameter, because a parameter is something the calling code chooses — `DocumentService` reads the context itself. When the documents grid shows one department, that department comes from the context and not from a combo box the browser could have been asked to change. This is the course habit in code: a rule that lives only in what the UI offered is not a rule.

The context is created after verification and never before. There is no anonymous context with an empty id waiting to be filled in, because an object that exists in both states will eventually be trusted in the wrong one. Every `AccessOps` screen except login requires the context and sends the browser back when it is missing, and a typed accessor that throws when the session holds nothing makes that a habit rather than a review note.

Timeout, idle logout and a Logout that means it

A `Wisej.NET` session is server state, and its lifetime is configuration, not code. The session timeout is set in the application's `Default.json`, and when it elapses the server disposes the session together with the windows, the controls and the `SessionContext` inside it. Pick a value the business can defend — twenty minutes suits a portal showing salary bands and signed approvals — and keep it shorter than any proxy timeout in front of the app, so the behaviour is the one you configured.

Idle logout is the visible half of the same rule. A `Wisej.Web.Timer` restarted on user activity warns a minute before the deadline, then runs the same clean-up as the Logout button: clear the context, drop anything cached about the person, write the audit record, and call `Application.Exit()`. Both paths call one method, because two clean-up routines drift apart and the one nobody clicks is the one that forgets something.

- The session ends on the server — closing the browser tab is not a logout.
- Every screen state that named a person is gone before the session is disposed.
- The browser lands on the login page, and going back in history shows no data.

Brute force, credential stuffing and fixation

An unthrottled login page is an unlimited guessing machine. `LoginThrottle` keeps counters per account and per client address; after five failures inside a short window the account is refused for a lockout period, and the counters reset on success. Give every failed answer the same shape as well as the same wording: when the user name does not exist, still verify against a fixed dummy credential, so a missing account and a wrong password cost the same time.

The message on screen is one generic sentence for every failure — unknown user, wrong password, locked, disabled. The real reason belongs in the audit row, where an administrator can see thirty unknown-user attempts from one address in ten minutes. That pattern is credential stuffing: pairs harvested from someone else's breach and tried in bulk, stopped by the per-address counter rather than the per-account one, since each account may be tried only once.

Session fixation is the subtler one. The identifier that carried the anonymous login page must not carry the verified identity, so the pre-login session ends after verification and the browser continues on a new one, with nothing copied across. Remember-me follows the same discipline: never a stored password and never a guessable value, but a long random per-device token, hashed at rest, tied to one employee, given an expiry, revocable from the admin console, and re-checked against the live account before the `SessionContext` is rebuilt.



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:

- ✓ Implement a login page that verifies credentials against a salted hash and never stores the password.
- ✓ Create a `SessionContext` that holds a verified identity in `Application.Session` and is the only source of truth for who the user is.
- ✓ Configure session timeout, idle logout and explicit logout so state is disposed and cannot be reused.
- ✓ Protect the login flow from brute force, credential stuffing and session fixation.

Lab / Exam Guide

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



Hands-on lab

Implement AccessOps authentication: a login page with lockout after five failures, password verification with a modern salted hash, a SessionContext created only after verification and stored in the session, an idle timeout that returns the user to login and clears state, an explicit Logout that ends the session, and an audit record for every login attempt.

Starting point

You open the AccessOps solution you created in Module 1, which is still a shell: a start-up window, a placeholder login page whose button simply opens the main console, and the threat model document that named authentication as the first control to build. The employee table exists with names and departments but no credentials, and nothing in the solution yet knows who is signed in.

Suggested approach

- ✓ Write `PasswordService` first, with a `Hash` method that generates a fresh random salt and a `Verify` method that re-derives with the stored salt and iteration count and compares using a constant-time equality check; add the salt, iteration count and hash columns to the employee table and seed two test accounts through the service.
- ✓ Define `SessionContext` with the employee id, display name, department id, manager flag, roles and sign-in time, plus a typed accessor over `Application.Session` that returns the current context and throws when nothing is signed in.
- ✓ Build `LoginService.SignIn` as the only path that verifies a credential: ask `LoginThrottle` whether the account is locked, verify the password, write the audit record for every outcome, and on success reset the counters, end the pre-login session and create the `SessionContext` for the new one.
- ✓ Rebuild the login page so its button handler only calls `LoginService` and reacts to the result — one generic sentence for every failure, the console for success — and make every other screen require the context and redirect to login when it is missing.
- ✓ Set the session timeout in `Default.json`, add the idle timer with its one-minute warning, and write a single Logout routine that clears the context, records the sign-out and calls `Application.Exit()`, called by both the button and the idle path.

Required deliverables

- ✓ Login page with lockout and generic error messages
- ✓ Password hashing and verification service
- ✓ SessionContext with verified identity
- ✓ Idle timeout and explicit logout
- ✓ Login audit records

Student review questions

- ✓ Your login page answers a wrong password in 480 ms and an unknown user name in 6 ms. Explain what an attacker learns from that difference alone, and describe the two changes that remove the signal without changing the message.
- ✓ A colleague suggests keeping the signed-in employee in a static property because "it is simpler than reaching into the session every time". Describe exactly what the second concurrent user sees, and at which moment the two sessions first interfere.
- ✓ Your Logout button clears the SessionContext but does not end the session. Name two things that are still recoverable afterwards, and say which one you consider the more serious in a portal that shows confidential documents.



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.