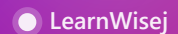


Auditing, Secrets and Configuration

Lesson Guide · Lab / Exam Guide



Lesson Guide

Audit trails for sensitive commands, correlation IDs, secrets management, environment configuration, logging without leaking data. What this means for a developer securing a production Wisej.NET app — and how to read this module.



Production focus

Audit trails for sensitive commands, correlation IDs, secrets management, environment configuration, logging without leaking data.

1. The production problem

AccessOps now authenticates people, resolves their permissions and guards its documents — and none of that helps on the morning somebody asks who approved request 4471, from which address, and whether anyone was refused first. The previous five modules decided what is allowed; this one decides what is recoverable afterwards. It also closes the gap that turns a small incident into a large one: a database password or an OIDC client secret in a file committed two years ago.

2. Key concepts

Concept	Why it matters
audit trail	A business record of every sensitive command — who acted, on what, when and from where — kept for a stated retention period because someone will have to answer for it.
append-only storage	An audit table the application account may only <code>INSERT</code> into. If the code that writes history can also rewrite it, the history proves nothing.
tamper evidence	A gapless sequence plus a hash chain, where each row's hash covers its own fields and the previous hash, so an edited row breaks the chain.
correlation ID	One identifier minted when a command starts and repeated on every log line and audit row it causes, so a single click can be reassembled later.
logging scope	An ambient property block — <code>ILogger.BeginScope</code> or a Serilog enricher — that attaches the correlation ID and actor to everything logged inside it.
secret store	Where a credential lives outside the build: user secrets locally, environment variables in a container, a managed vault in production — or a managed identity, with nothing stored at all.
redaction	A rule about what never reaches a log sink — passwords, tokens, cookies, connection strings and personal data — enforced by structured logging, not by memory.

3. Lesson narrative

The record that survives the argument

Two different things get called "logging". A log line is diagnostic and may be sampled or rotated away after a week. An audit row is evidence: it has a retention period, an owner, and a reader who is not a developer. AccessOps needs an `AuditService` as the single entry point for the second kind, called by every sensitive command — approving a request, granting or revoking a permission, downloading a confidential document, and every login attempt from Module 2. Nothing else writes to the audit table.

One row answers six questions: the actor's stable identifier, the action name (`Approval.Approve` , `Document.Download`), the target type and ID, the UTC timestamp from the server clock, the client address and user agent, and the correlation ID. Record the outcome too, because a denial is often more interesting than a success. In Wisej.NET the client's network address and browser user agent come from the per-session application context rather than from anything the page sends up, so capture them once when the `SessionContext` is created after login. Take the actor from that verified context as well — never from a control or a caller-supplied parameter.

Then make the row hard to unwrite. The application's database login gets `INSERT` and nothing else, so a bug or an injected statement cannot quietly erase a Tuesday. Give each row a gapless sequence and a hash over its own fields plus the previous row's hash; a scheduled pass recomputes the chain and names the first row that no longer matches. That is *tamper evidence*, not prevention — it makes a change visible rather than impossible. Finally, write the audit row in the same transaction as the business change, so an approval without its record is unreachable.

```

using (_log.BeginScope(new Dictionary<string, object>
    { ["CorrelationId"] = ctx.CorrelationId, ["UserId"] = ctx.UserId }))
using (var tx = _db.BeginTransaction())
{
    _approvals.Approve(requestId, ctx);           // the business change
    _audit.Record(AuditEntry.For(ctx,           // actor, address, correlation
        "Approval.Approve", "ApprovalRequest", requestId));
    tx.Commit();                                // both, or neither
}

```

One ID, one story

The correlation ID is created where a user command begins — at the top of a thin event handler, before any service is called. A `Guid` is enough; if you already run with `System.Diagnostics.Activity`, use the current activity's trace identifier so the ID matches anything else speaking the W3C trace format. Put it on the per-command context that also carries the actor and the client address, and pass that context into every service. Do not reach back into the session from inside a service: background work started with `Application.StartTask` can outlive the session that started it.

Attaching the ID to log lines is not the caller's job. Open a logging scope around the command — `ILogger.BeginScope` with a property dictionary, or Serilog's `LogContext` with an enricher — and every line written inside it carries the correlation ID and the actor automatically, including lines from libraries below you. Use structured logging so the ID is a named property in the sink, which makes "show me everything for correlation 7f3c" a query rather than a grep. Then surface it: when a command fails, end the friendly sentence with the ID as a reference.

Secrets belong to the environment, not the repository

AccessOps holds three credentials by now: the database connection string, the OIDC client secret from Module 4, and the signing key for the expiring share links from Module 5. None belongs in a tracked file. In development use .NET user secrets — the project gets a `UserSecretsId` and the values live outside the solution folder, so a clone of the repository has no credentials in it. In deployment, supply them as environment variables or read them from a managed vault through a configuration provider. Better still, use a managed identity: the app authenticates as itself and receives a short-lived token, so there is no stored secret to leak.

This works because ASP.NET Core configuration is layered and later sources win: `appsettings.json`, then the environment overlay, then user secrets in development, then environment variables, then a vault provider. The committed files hold structure and harmless defaults; the secret-shaped values arrive per environment. Wisej.NET's own `Default.json` configures the shell — theme, session timeout, start-up window — and is no more a secret store than `Web.config` is.

A secret you cannot rotate calmly is one you will rotate badly at 3 a.m., so the rotation procedure is a deliverable here. Write down, per secret: the owner, the store, the expiry, and the replacement steps. Design for two valid credentials at once — a second client secret, a second database login — so rotation is a rolling restart rather than an outage. And once a value has been committed, deleting the line does not help: rotate it and revoke the old one.

Log the incident, not the person

Logs are copied to search indexes, shipped to third-party services, opened on support laptops and kept far longer than anyone intends. They must hold enough to reconstruct an incident and nothing damaging on its own. Never log a password (not even a failed one), a token, an authorization header, a cookie, a connection string, or the contents of a confidential document.

Personal data needs the same discipline for a different reason: an audit row is a lawful record with a retention policy, while a debug line containing an employee's home address is an uncontrolled copy of it. Log the stable user ID rather than the name and email, and the document ID rather than a file name that may itself be sensitive. Structured logging helps, because you log named properties from an allowlist instead of serialising a whole request object and hoping.

- Never log: passwords, tokens, cookies, connection strings, document contents.
- Log by reference: user ID, document ID, request ID — not names or free-text personal data.
- Enforce it at the sink: a masking policy, plus a test asserting a known secret marker never reaches the output.

Close the loop with a review rather than a rule: walk the log output of one real approval and one real download end to end, and mark every field as necessary, redacted or removed.



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:

- ✓ Record who did what, when and from where for every sensitive command with a tamper-evident audit trail.
- ✓ Carry a correlation ID through logs so one user action can be traced across services.
- ✓ Keep secrets out of source control using environment configuration, secret stores and managed identities.
- ✓ Log enough to investigate an incident without logging passwords, tokens or personal data.

Lab / Exam Guide

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



Hands-on lab

Add auditing and secret management to AccessOps: an `AuditService` that records approvals, permission changes and document downloads with user, time, address and correlation ID, an append-only audit table, a correlation ID created per command and written to every log line, connection strings and the OIDC client secret moved to environment configuration with a documented rotation procedure, and a log review that redacts sensitive fields.

Starting point

You open the AccessOps solution as it stands at the end of Module 5: a verified `SessionContext` created at login, a `PermissionService` and an authorization guard inside `DocumentService` and `ApprovalService`, OIDC sign-in, and the documents feature with its upload pipeline and signed share links. The approval screen, the admin permission editor and the document list all work — they simply leave no trace of what they did, and the database connection string and the OIDC client secret are still sitting in a configuration file that source control tracks.

Suggested approach

- ✓ Create the audit table with a gapless sequence, the actor, action, target type and ID, UTC timestamp, client address, user agent, outcome, correlation ID and a hash column, then grant the application's database login `INSERT` only — no `UPDATE`, no `DELETE`.
- ✓ Write `AuditService` as the only writer of that table, taking the actor and client details from the verified `SessionContext`, computing each row's hash over its own fields plus the previous row's hash, and enrolling the insert in the same transaction as the business change.
- ✓ Add a per-command context that mints a correlation ID at the top of each event handler, pass it into `ApprovalService`, `DocumentService` and the permission editor, and open a logging scope so every line written during the command carries the ID and the actor.
- ✓ Move the connection string, the OIDC client secret and the share-link signing key out of the tracked configuration files into user secrets for development and environment variables for deployment, then write the rotation procedure with owners, expiry dates and the two-valid-credentials overlap.
- ✓ Call `AuditService` from approve, permission grant and revoke, and document download — recording denials as well as successes — then run one approval and one download and review every resulting log line for passwords, tokens, headers, connection strings and personal data.

Required deliverables

- ✓ `AuditService` and append-only audit table
- ✓ Correlation ID in every command and log line
- ✓ Secrets moved to environment configuration

- ✓ Secret rotation procedure
- ✓ Log redaction review

Student review questions

- ✓ Your audit table is insert-only for the application, and every row is hash-chained. Name something an attacker with the application's own database login can still do to the trail, and explain exactly how the verification pass would show it.
- ✓ An approval is started on the approval screen and finishes in a background task started with `Application.StartTask` after the user has signed out. Where does that task get its correlation ID and its actor from, and what breaks if it reads them from the session instead?
- ✓ A developer removes the OIDC client secret from the tracked configuration file and pushes the change. Why is the secret still compromised, and what has to happen before the incident is actually closed?



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.