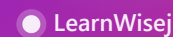


Advanced Session, State, Tenant, and Concurrency Design

Lesson Guide · Lab / Exam Guide



Lesson Guide

Session ownership, tenant isolation, optimistic concurrency, static-state audits. What this means for an advanced Wisej.NET team — and how to read this module.



Production focus

Session ownership, tenant isolation, optimistic concurrency, static-state audits.

1. The production problem

Advanced applications serve many users, tenants, browser tabs, and background operations. State mistakes become security bugs, data leaks, or inconsistent workflows.

2. Key concepts

Concept	Why it matters
session context	The per-user server state Wisej.NET keeps alive for each browser session. It is where user identity and working state belong, never in static fields.
tenant context	The current tenant identity resolved once per session and enforced in services and data access, so one customer can never see another customer's data.
correlation ID	A unique identifier attached to a request or job and carried through logs, so one user action can be traced across services and servers.
optimistic concurrency	Detecting that two users edited the same record by comparing a version or timestamp, and handling the conflict instead of silently overwriting.
tab-scoped state	State that belongs to one browser tab or window rather than the whole session, so two open tabs do not corrupt each other's work.
static state audit	A review of every static field and singleton to confirm it holds no per-user or per-tenant data, because statics are shared by every session on the server.
race condition review	Checking code paths where background tasks, timers, or multiple requests touch the same session state so updates are synchronized correctly.

3. Lesson narrative

State is the architecture

Wisej.NET keeps rich server-side state, which is a powerful advantage. In an advanced system that advantage must be explicit. Every piece of state needs an owner: application, tenant, user, session, browser tab, workflow, background job, or request.

Tenant isolation is not a filter label

Multi-tenant applications must enforce tenant boundaries in services and data access. A grid filter is only a visual condition. A command handler, repository, and authorization check must all understand tenant identity and reject cross-tenant access.

Five scopes of state

Deliberate state design starts by naming the scope of every value. Application state is shared by everyone on the server and must be immutable or synchronized: configuration, reference data, feature flags. Tenant state belongs to one customer and is reached only through the tenant context. User state follows a person across sessions, such as preferences. Session state lives in `Application.Session` for one browser session: identity, current tenant, working context. Tab state belongs to one browser window, such as an open work order, and request state exists for one command, such as its correlation ID.

Each scope has a home. Application and tenant state live in services and caches keyed by tenant. Session state lives in the session context object. Tab state lives in the form or page that owns the tab. Request state lives in the command context that travels through the call. A value stored one scope too wide becomes visible to the wrong people, and a value stored one scope too narrow is lost when the user opens a second tab.

In review, ask of every field: who else can see this, and what happens when the user opens the same screen twice.

A tenant-aware SessionContext and CommandContext

The lab's `SessionContext` is a single object created at login and stored in the session. It holds the user ID, display name, roles, permissions, culture, and the current tenant, resolved from verified claims and the tenant the user chose from the ones they are entitled to. Screens read it to adapt what they show. Services read it to decide what they allow. Nothing reads tenant identity from a UI control, because a dropdown value can be changed from the browser.

The `CommandContext` is the per-request companion. When a handler starts a command, it builds a context from the session context and adds a new correlation ID, a timestamp, and the command name. That context is passed explicitly into services, repositories, and background jobs, so every layer knows the tenant and the correlation ID without reaching back into the session. Background work in particular cannot rely on the session, because the session may end before the job does.

The tenant guard in every service compares the tenant on the context with the tenant of the record being touched and rejects a mismatch before any business logic runs.

Optimistic concurrency and conflict resolution

Two dispatchers can open the same work order in two sessions. Locking the record for the first one blocks the second for as long as a lunch break, so enterprise editing screens use optimistic concurrency instead. Each record carries a version, a row version column or a timestamp, that is read when the record is loaded and checked when it is saved. If the version changed in between, another session saved first, and the save is rejected.

The rejection is where the design work is. A bare error message leaves the user with lost work and no idea why. The lab's conflict dialog explains that the record was changed by someone else since it was opened, and offers three paths: Reload discards the local edits and shows the current record, Compare shows both versions field by field so the user can merge, and Cancel leaves the screen as it is for the user to decide later. The choice is recorded in the audit trail with the correlation ID.

Concurrency checks belong in the command handler, not the form, so the same rule applies when the save comes from a batch operation or an import.

Auditing static and shared state

A Wisej.NET server process hosts every session in the same memory. A static field, a singleton service, or a cache without a tenant key is therefore shared by every user of every tenant at once. A static that holds the current user is the classic leak: the last session to log in overwrites it, and the next request from another user runs as that person. The static state audit in the lab lists every static field, singleton, and cache in the solution and answers one question for each: what does it hold, and could that ever be per-user or per-tenant.

Thread safety is the second half. Background tasks started with `Application.StartTask`, timers, and concurrent requests from several tabs can all touch the same objects at the same time. A shared dictionary that is read and written without synchronization corrupts silently. The race condition review walks each shared object and checks whether its access is synchronized or whether it should not be shared at all.

- Every static holds only configuration or immutable reference data.
- Every cache is keyed by tenant, and its entries cannot be read across tenants.
- Every shared mutable object has a documented synchronization strategy.



Advanced production habit

Do not stop when the screen looks correct. Explain state ownership, security implications, failure paths, and production behavior before considering the module complete.



Learning objectives

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

- ✓ Design session, user, tenant, tab, and request state deliberately.
- ✓ Implement a tenant-aware SessionContext and command context.
- ✓ Use optimistic concurrency and conflict resolution in editing workflows.
- ✓ Audit static/shared state for thread safety and tenant leakage.

Lab / Exam Guide

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



Hands-on lab

Extend EnterpriseOps with tenant selection, tenant-aware services, correlation IDs, and optimistic concurrency on work orders. Create a conflict dialog that explains stale edits and offers Reload, Compare, and Cancel paths.

Suggested approach

- ✓ Read the production problem and identify the architecture risk before touching the UI.
- ✓ Create or update the relevant service, command, model, or component boundary first.
- ✓ Build the Wisej.NET screen or component using the established project standards.
- ✓ Add a failure-path demonstration, not only the happy path.
- ✓ Record what changed in the learning log and update the relevant checklist.

Required deliverables

- ✓ Tenant-aware SessionContext
- ✓ CommandContext with correlation ID
- ✓ Static-state audit report
- ✓ Optimistic concurrency implementation
- ✓ Conflict-resolution dialog

Student review questions

- ✓ Can tenant ID be spoofed from a UI control?
- ✓ What happens when the same work order is edited in two sessions?
- ✓ Which services can safely be shared?



Advanced production habit

Do not stop when the screen looks correct. Explain state ownership, security implications, failure paths, and production behavior before considering the module complete.