

Real-Time Systems, Background Pipelines, Notifications, and Imports

Lesson Guide · Lab / Exam Guide

 LearnWisej

Lesson Guide

Background jobs, progress, notifications, cancellation, queues, server push. What this means for an advanced Wisej.NET team — and how to read this module.



Production focus

Background jobs, progress, notifications, cancellation, queues, server push.

1. The production problem

Advanced Wisej.NET systems often run imports, report generation, integrations, and notifications while users continue working. These pipelines must be observable, cancelable, and safe across sessions.

2. Key concepts

Concept	Why it matters
job model	A persisted record describing a background operation: its type, input, status, progress, owner, and result, independent of any open screen.
background queue	A durable queue that accepts jobs and processes them outside the request cycle, so long-running work never blocks the user interface.
Application.StartTask	The Wisej.NET API for running work on a background thread that stays attached to the session context, so it can update the UI safely when done.
Application.Update	Pushes pending UI changes made from a background thread to the browser immediately, which is how progress and notifications appear without user action.
notification service	A central service that delivers job results and alerts to the right user session, whether or not the originating screen is still open.
retry policy	Rules for how many times and how quickly a failed job is retried, with backoff, so transient errors recover without hammering downstream systems.
idempotency	Designing jobs so running them twice produces the same result as running them once, which makes retries and duplicate messages safe.
job audit	A record of who started each job, when it ran, and what it changed, so background work is as accountable as interactive work.

3. Lesson narrative

Jobs outlive screens

A long-running import should not depend on a single button click handler staying alive. The UI starts or observes a job; a service executes it; a job store records status; notification services publish updates to interested sessions.

Push meaningful state

Real-time updates should communicate meaningful milestones: queued, started, validated, imported, failed, completed. Calling update constantly wastes resources and makes debugging harder.

Separating execution from display

The Import Center in the lab has two halves that never share an object. A job service executes imports: it takes a job definition, reads the file, validates and writes rows in batches, and writes its progress and result to a job store. The screen observes: it starts a job by calling the service, then displays whatever the store says about it. The screen can be closed, reopened in a new session, or opened by a different user with the right permission, and the job neither knows nor cares.

The job model is the contract between them. Each job record holds an identifier, the type of work, who started it and in which tenant, its status, a progress value, a message, timestamps, and a result summary

with per-row errors. Because the record is persisted, the status survives a server restart, and a support engineer can see every job of the day without asking anyone.

The background queue sits between the two halves. Starting a job enqueues it; a worker picks it up. That is what makes the executing thread independent of the request that asked for it.

Real-time updates with intent

Wisej.NET can push changes to the browser without a request, which is how progress appears while the user waits. A background task started with `Application.StartTask` keeps the session context, changes the progress bar and status label, and calls `Application.Update` to flush those changes to the browser. Used carelessly, the same mechanism pushes an update for every imported row and turns a two-minute import into a flood of round trips that slows the server and the browser.

The lab's `ImportWorkOrdersJob` publishes milestones instead: queued, started, validated, every completed batch, failed, completed. A milestone sink receives them, and the sink decides whether to update the UI now or wait, so a job that produces a thousand events results in a handful of updates. Each update carries meaningful state, which is what the user actually wants to know.

The second lab question, how often the UI updates, has a numeric answer in a good design, and it is a small number per second at most.

Job status and notification models

A job's status is a small state machine: queued, running, completed, failed, cancelled. Progress is a separate value, because a running job at ninety percent and a running job at ten percent are the same status. The status store exposes the current record and a history, so the job detail screen can show when each transition happened and what message accompanied it.

Notifications are how the result reaches a person who is no longer watching. A notification service accepts a message for a user or a role, stores it, and delivers it to any live session of that user through the bell in the lab's Import Center. If the user is not logged in, the notification waits until they are. Notifications are records too, with read and unread state, so nothing is lost when a browser closes at the wrong moment.

Both models are tenant-aware. A job belongs to a tenant, its notifications belong to users of that tenant, and the store never returns a job across the tenant boundary.

Cancellation, retries, and closed sessions

A cancel button sets a flag; it does not stop anything by itself. The job honors cancellation by checking the flag between batches, finishing the current batch cleanly, recording what was done, and moving to the cancelled status. Cancellation that stops mid-row leaves half-written data, which is worse than finishing the batch. The retry policy distinguishes transient errors, such as a timeout writing one row, which are retried a bounded number of times with a delay, from terminal errors, such as a malformed file, which fail the job immediately.

Partial failure is reported per row. An import of ten thousand rows with twelve bad ones completes with a result that lists the twelve, and the user fixes and re-imports those, not the whole file. Idempotent row processing makes this safe: importing the same row twice yields one record.

Session closure is the lab's failure path. The user closes the browser mid-import, the job keeps running because it belongs to the service and not the session, and on reopening the screen finds the job in the store and shows its final state. Who owns the job after the session closes has one answer: the service.



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 background pipelines that separate job execution from UI progress display.
- ✓ Use real-time updates intentionally and avoid excessive client updates.
- ✓ Implement notification and job status models.
- ✓ Handle cancellation, retries, partial failures, and session closure.

Lab / Exam Guide

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



Hands-on lab

Build an Import Center with a job queue, progress observer, cancellation, retryable row processing, notifications, and a job detail screen. Demonstrate recovery after closing and reopening the session.

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

- ✓ Job model and status store
- ✓ Background queue abstraction
- ✓ Progress observer UI
- ✓ Notification panel
- ✓ Retry and cancellation policy

Student review questions

- ✓ Who owns the job after the session closes?
- ✓ How often does the UI update?
- ✓ Which errors are retryable and which are terminal?



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.