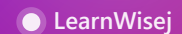


State, Sessions, Background Work & Errors

State, Sessions & Background Work · Safe Error Handling & Logging



State, Sessions & Background Work

Per-user session state vs shared state (and the static-field trap), plus running long work in a responsive background job.



Lesson goal

Understand **per-user session state** — and why it differs from shared state — then run longer work in a **background task** that keeps the page responsive and updates progress on purpose.

1. Per-user session state

A Wisej.NET application runs on the server, but every connected user should feel like they have their own copy of the app. **Session state** is data that belongs to one connected user session — the selected ticket, the current filter, the current background job, and that user's progress messages.

State type	What it means	Beginner example
UI state	Information used only to display the current screen.	Selected row, visible panel, status label text.
Session state	Information that belongs to one user session.	Current import job, completed jobs for this logged-in user.
Business state	Actual application data that should be saved or shared intentionally.	Tickets stored by a service / repository.
Static / shared state	State shared across users unless carefully designed.	A static list of jobs that every user can accidentally change.



Warning: static / shared state

Don't store normal per-user information in **static** fields unless you intentionally want it shared. A static list can let one user see another user's data or job progress — one of the easiest beginner mistakes in server-side apps.

2. Background work and responsive pages

Some work takes too long to run directly inside a button click — exporting records, importing files, sending notifications, processing reports. A good app starts the job, disables unsafe buttons, shows progress, and lets the page stay usable while the job continues.

```
User opens page → a new session sees its own state
Button click    → the UI event starts server logic
Background job  → long work runs without freezing the UI
Progress update → status, bar and log update safely
Complete / error → the user sees a clear final message
```

3. Bad pattern vs better pattern

Bad pattern	Better pattern
Put all the work directly in the click event and freeze the page.	Start a background task and update progress / status intentionally.
Let users click Start many times and create duplicate jobs.	Disable Start while a job is running; enable Cancel when appropriate.
Show only a generic error or crash.	Log details for developers, but show a short, friendly message to the user.



You're ready for the next reading

You can tell session state from shared state, spot the static-field trap, and describe a responsive background-job workflow. Next: handling errors safely and logging useful detail.

Safe Error Handling & Logging

Catch exceptions and show safe messages, log useful detail, and keep the job handler readable with try/catch/finally.



Lesson goal

Catch exceptions and show useful messages **without** exposing internal details, add simple logging, and keep the background-job event handler readable with a clear try / catch / finally pattern.

1. Safe error handling

Error handling should protect both the user and the system. The user needs a clear message about what happened and what to do next; the developer needs enough detail in the log to troubleshoot. Never expose connection strings, file paths, stack traces, secrets or internal server details in the user message.

Error detail	Show to user?	Why
"Import failed. Please try again or contact support."	Yes	Useful and safe.
<code>NullReferenceException</code> at TicketService.cs line 52	No	Internal implementation detail — put it in a log instead.
Database password or connection string	Never	Sensitive information.
Job ID and timestamp	Sometimes	Can help support find the log, if it doesn't expose secrets.

2. A simple logging checklist

- Log when the job starts, each major step, completion, cancellation and failure.
- Include timestamps and job names so the log is useful later.
- Keep user-facing messages short; keep technical details in the log.
- Don't log passwords, API keys, full card numbers or private data without a strong reason and protection.
- When troubleshooting, check: button state, current job state, progress value, exception message, and the most recent log entry.

3. The beginner pattern to remember

The button click starts the workflow. The background job does the long work. The UI updates progress and status. The service or helper owns reusable logic. The logger records details. The user sees safe messages.

```
private async void btnStartImport_Click(object sender, EventArgs e)
{
    try
    {
        SetJobRunning(true);
        await RunImportJobAsync();
        lblStatus.Text = "Import completed successfully.";
    }
    catch (Exception ex)
    {
        LogError(ex);
        lblStatus.Text = "Import failed. Please try again or contact support.";
    }
    finally
    {
        SetJobRunning(false);
    }
}
```



Why finally matters

finally runs whether the job succeeds or throws, so it's the right place to return buttons and UI state to a safe condition — the page never gets stuck in “running” mode.



You're ready for the lab

Next: watch the background-work walkthrough, pass the knowledge check, then build a job runner — Start/Cancel buttons, a progress bar and log, a try/catch/finally that logs detail but shows a safe message, and UI state that always resets.