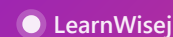


Background Tasks, Real-Time Updates & Synchronization

Go deeper on background work: running off the UI thread, marshaling updates back safely, cancellation, and per-row error reporting for the CSV import.



Applied goal

Build a background CSV import for the TicketOps Console that runs off the UI thread, drives a progress bar and log panel, marshals every UI update back through the session context, honors a `CancellationToken`, and records per-row errors without aborting the whole file.

1. Why the import can't run inline

A Work Order operator drops a *work-orders.csv* onto the TicketOps import screen. The file may hold thousands of rows, each parsed, validated, and written to the ledger. If the click handler processes the file inline, the request that carries that click stays open, the browser spins, and the whole session is unresponsive until the last row lands.

The fix from the Lesson Guide is to hand the work to `Application.StartTask`, which runs the delegate on a worker thread while the click handler returns immediately. The operator keeps a live screen — a progress bar, a running log, and a **Cancel** button that actually responds.

Approach	UI while importing	Cancel	Per-row errors
Inline in click handler	Frozen until done	Impossible	One throw aborts all
<code>Application.StartTask</code> + push	Live progress & log	<code>CancellationToken</code>	Logged, import continues

2. The thread boundary you must respect

Wisej.NET is server-side: your controls live inside a specific user's session, not on the worker thread. Reaching out from the background task to set `progressBar.Value` or append to a `TextBox` directly touches session state from a thread that isn't bound to that session. The safe boundary is `Application.RunInContext(session, () => { ... })` — it re-enters the correct session so control changes and the subsequent `Application.Update` push are valid.

Capture the session reference *before* the task starts, on the UI thread, and pass it in. Don't read `Application.Session` from inside the worker — there it is ambiguous.

Update from...	Touch controls directly?	Correct pattern
Click / event handler (UI thread)	Yes	Direct, already in context
Background task delegate	No	Wrap in <code>RunInContext</code> + <code>Application.Update</code>



Pitfall

Setting `progressBar.Value` or appending log text straight from the background thread may appear to work in a demo, then intermittently corrupt session state or throw under load. Never touch controls from the worker without re-entering the session context first.

3. The ImportService

Keep the parsing, validation, and ledger writes in a plain `ImportService` so the screen only wires events. The service reports progress and log lines through callbacks and checks the `CancellationToken` once per row — cancellation is cooperative, so a row that ignores the token can't be interrupted mid-write.

```
// ImportService.cs - no Wisej types here; pure import logic
public class ImportResult { public int Imported; public int Failed; public bool Cancelled; }

public class ImportService
{
    public ImportResult Run(
        Stream csv,
        int totalRows,
        Action<int, int> onProgress,          // (rowIndex, totalRows)
        Action<string> onLog,
        CancellationToken token)
    {
        var result = new ImportResult();
        int row = 0;
        foreach (var line in ReadLines(csv))
        {
            // Cooperative cancellation: check before each row
            if (token.IsCancellationRequested) { result.Cancelled = true; break; }
            row++;
            try
            {
                var workOrder = ParseWorkOrder(line);    // may throw on bad data
                _ledger.Insert(workOrder);
                result.Imported++;
            }
            catch (FormatException ex)
            {
                // Per-row failure: log it, keep going - one bad row must not abort the file
                result.Failed++;
                onLog($"Row {row}: skipped - {ex.Message}");
            }
            onProgress(row, totalRows);
        }
        return result;
    }
}
```

4. Wiring the task and the safe UI updates

The click handler captures the session, starts the task, and lets the service call back. Each callback re-enters the session and calls `Application.Update` so the change is pushed to the browser. Throttle the push — updating on every single row floods the socket; update on progress steps or every N rows.

```
// WorkOrderImportPage.cs
private CancellationTokenSource _cts;

private void btnImport_Click(object sender, EventArgs e)
{
    var session = Application.Session; // capture on the UI thread
    _cts = new CancellationTokenSource();
    btnImport.Enabled = false;
    btnCancel.Enabled = true;

    Application.StartTask(() =>
    {
        var service = new ImportService();
        var result = service.Run(
            _uploadedStream, _totalRows,
            onProgress: (row, total) =>
            {
                if (row % 25 != 0 && row != total) return; // throttle the push
                Application.RunInContext(session, () =>
                {
                    progressBar.Value = row * 100 / total;
                    Application.Update(this); // push to this browser
                });
            },
            onLog: (msg) => Application.RunInContext(session, () =>
            {
                logPanel.AppendText(msg + Environment.NewLine);
                Application.Update(this);
            }),
            token: _cts.Token);

        // Completion state - also inside the session context
        Application.RunInContext(session, () =>
        {
            lblStatus.Text = result.Cancelled
                ? $"Cancelled: {result.Imported} imported, {result.Failed} skipped."
                : $"Done: {result.Imported} imported, {result.Failed} skipped.";
            btnImport.Enabled = true;
            btnCancel.Enabled = false;
            Application.Update(this);
        });
    });
}

private void btnCancel_Click(object sender, EventArgs e) => _cts?.Cancel();
```



Production habit

Always re-enable **Import** and disable **Cancel** from a completion block that runs in the session context — including the cancelled and failed paths. A task that exits without restoring button state leaves the screen stuck.

5. Thread-safety and shared state

The worker thread and the UI thread can touch the same fields. The `_cts` token source is written by the click handler and read by the task — assign it before `StartTask` and never reuse a cancelled source. If the import accumulates into a shared collection (a summary list, a running counter outside `ImportResult`), guard it with a `lock`; a plain `List<T>.Add` from two threads can corrupt the list.

Prefer to keep mutable state local to the task and only publish finished, immutable results back through `RunInContext`. That keeps the locking surface small and the UI updates deterministic.



Production habit

Wrap each ledger write in its own try/catch keyed to the row, so a single malformed Work Order becomes a logged skip — never an exception that unwinds the whole import and loses the rows already processed.

Decision cheat-sheet

- ✓ **Application.StartTask** — moves the import off the request thread so the screen stays live.
- ✓ **RunInContext(session, ...)** — the only safe place to touch controls from the worker.
- ✓ **Application.Update(this)** — pushes the change to that browser; call it after meaningful updates, throttled.
- ✓ **CancellationToken** — checked once per row; cancellation is cooperative, not forced.
- ✓ **Per-row try/catch** — turns a bad row into a logged skip, not a whole-file abort.
- ✓ **lock** — guards any state written by both the task and the UI thread.