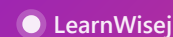


Background Tasks, Real-Time Updates & Synchronization

Use background tasks and real-time server push to perform long-running work, update UI safely, support cancellation, and avoid blocking the user interface.



Lesson goal

Use background tasks and real-time server push to perform long-running work, update UI safely, support cancellation, and avoid blocking the user interface.

In Module 1 you traced what happens on a single click: the browser sends a message, Wisej.NET raises your event on the server, your handler runs, and only the properties you changed travel back down as the round-trip closes. That model is fast *because* the round-trip is short. This module is about the work that **isn't** short — importing a file, calling a slow API, generating a report — and how to run it without freezing the very screen the user is looking at.

1. Why long work must not block the round-trip

Recall the click lifecycle. While your handler runs, that user's round-trip is *open*: the browser is waiting for the server to finish and send back its diff. If the handler returns in a few milliseconds, nobody notices. But if it spends thirty seconds inside a loop importing rows, the round-trip stays open for thirty seconds — and to the user the app looks **frozen**. No progress bar moves, no Cancel button responds, because the server has not yet finished the turn that would push those updates.

The fix is to **get off the round-trip quickly**. Start the long work, let the click handler return so the browser is released, and then feed updates back to the screen *as the work progresses* rather than all at once at the end. That is exactly what background tasks plus server push give you.



Why this matters

A handler holds the user's turn open until it returns. Short handlers feel instant; a long synchronous handler makes the whole session feel dead. The goal isn't just "do work in the background" — it's "return the turn quickly, then push updates."

2. Key concepts

The production focus of this module, concept by concept:

#	Concept	What it does for you
1	Application.StartTask	Runs work on a background thread that still knows which user session it belongs to.
2	Application.Update	Flushes queued control changes out to the browser (server push) outside a normal round-trip.
3	RunInContext	Marshals a piece of code back onto the correct application/session context before it touches controls.
4	WebSocket push	The persistent connection that lets the server send updates without the browser asking first.
5	Polling fallback	Periodic client check-ins used when a live push channel isn't available.
6	CancellationToken	The cooperative signal that lets the user stop long work cleanly.
7	Synchronization and locking	Protects shared state when background work runs alongside the UI thread.

3. Updating a control from a background thread is different

Inside a normal event handler you are already *on* the session's application context. When you write `label1.Text = "Saved"`, Wisej.NET is watching that session, records the change, and ships it in the round-trip's closing diff. You get thread affinity and the flush for free.

A background thread has neither. It runs outside any round-trip and, by default, is **not attached to any user's application context**. Two problems follow:

- **Affinity.** Touching a control from an unattached thread means Wisej.NET doesn't know *which* session's controls you mean. You must marshal your control updates back onto the owning application/session context first (this is what `Application.StartTask` and `RunInContext` are for).
- **Flushing.** Even once you're on the right context, there is no open round-trip whose diff will carry your change to the browser. You must explicitly push it — call `Application.Update` to flush the queued changes over the WebSocket connection. Without that call, the label's text changes on the server object but the user sees nothing until the next natural round-trip.

So the mental rule for background UI updates is two steps: **get onto the context, then flush the push**. Miss the first and you update the wrong (or no) session; miss the second and the change appears only after some later click.



Plain-English definition

Application context is the "which user am I?" ambient state a thread needs before it can safely read or write that user's controls. A background thread starts without it; you attach it (via `StartTask` / `RunInContext`) and then flush changes to the browser with `Application.Update` .

4. A worked background task

The example below imports a batch of records off the UI thread. It reports progress, honors cancellation, updates controls on the correct context, and pushes each update to the browser. Read it as: *start the task, loop with a cancellation check, marshal UI updates back to the session and flush them, then report the final state.*

Starting the work without blocking the click handler

```
private CancellationTokenSource _cts;

private void btnImport_Click(object sender, EventArgs e)
{
    // Return the round-trip fast: kick off the task and let the handler exit.
    _cts = new CancellationTokenSource();
    btnImport.Enabled = false;
    btnCancel.Enabled = true;

    // StartTask runs on a background thread that stays bound to THIS session.
    Application.StartTask(() => RunImport(_cts.Token));
}

private void btnCancel_Click(object sender, EventArgs e)
{
    // Cooperative cancellation – signals the token the task is watching.
    _cts?.Cancel();
}
```

The background method: progress, cancellation, safe UI updates, and a flush

```

private async Task RunImport(Cancellation_token token)
{
    try
    {
        var rows = await _importService.LoadRowsAsync(token); // slow I/O, off the UI
        thread
        int total = rows.Count;

        for (int i = 0; i < total; i++)
        {
            token.ThrowIfCancellationRequested(); // stop promptly if the user
            cancelled

            await _importService.SaveRowAsync(rows[i], token); // the real work

            int done = i + 1;
            // Marshal UI changes back onto this session's context, then push them.
            RunInContext(() =>
            {
                progressBar.Value = done * 100 / total;
                lblStatus.Text = $"Imported {done} of {total}";
                Application.Update(this); // flush the diff to the
                browser now
            });
        }

        RunInContext(() =>
        {
            lblStatus.Text = "Import complete";
            btnImport.Enabled = true;
            btnCancel.Enabled = false;
            Application.Update(this);
        });
    }
    catch (OperationCanceledException)
    {
        RunInContext(() =>
        {
            lblStatus.Text = "Import cancelled";
            btnImport.Enabled = true;
            btnCancel.Enabled = false;
            Application.Update(this);
        });
    }
    catch (Exception ex)
    {
        _log.Error(ex);
        RunInContext(() =>
        {
            lblStatus.Text = "Import failed - see the log.";
            btnImport.Enabled = true;
            btnCancel.Enabled = false;
            Application.Update(this);
        });
    }
}

```

```

    });
  }
}

```

Notice the shape. The click handler is tiny — it starts the task and returns, so the browser is released immediately. Every control touch inside the background method is wrapped so it runs on the session context and is followed by a flush, so the user sees the progress bar and status text move in real time. And a single `CancellationToken`, checked once per iteration, is enough to stop the loop the moment the user hits Cancel.



Production habit

Define progress, cancellation, partial failure, logging, and post-completion UI state *before* coding the task. Notice each of those has a home in the example above — and each `catch` restores the buttons so the screen never gets stuck mid-import.

5. Choosing an approach

Not every "slow" thing needs a background thread. Match the tool to the shape of the work:

Approach	Best for	Watch out for
async/await in the handler	Work that is slow because it <i>waits</i> (a database call, an HTTP request). <code>await</code> frees the thread during the wait and the update rides the normal round-trip.	It still holds the user's turn open until it completes — fine for a couple of seconds, not for a two-minute import.
Background worker (StartTask)	Long or CPU-bound work for <i>one</i> user that must not block their screen. Reports progress and pushes updates as it goes.	You must marshal UI updates back to the context and call <code>Application.Update</code> to flush — none of that is automatic.
Server push to many sessions	Broadcasting an event to <i>every</i> connected user — a price tick, a chat message, a "system going down" banner.	Each target session must be updated on its own context; a shared timer touching many sessions needs care with locking and per-session flushes.

Under the hood, all three rely on the same persistent **WebSocket** connection to deliver updates the browser didn't explicitly ask for. When a live socket isn't available, Wisej.NET can fall back to **polling** — the client periodically checks in — which still delivers pushed changes but with more latency. Your code is the same either way; only the timeliness differs.



Learning objectives

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

- ✓ Start session-bound background work and update the correct user interface safely.
- ✓ Use `Application.Update` appropriately and understand polling-fallback behavior.
- ✓ Design progress, cancellation, and error reporting for long-running tasks.

- ✓ Avoid race conditions when background work touches shared application state.

6. Application.Update: flush deliberately, not in a tight loop

Every call to `Application.Update` sends a message over the connection. That makes it powerful — it's how progress reaches the screen mid-task — but also something to **spend deliberately**. Flushing on every row of a 50,000-row import floods the connection with tiny messages and can make the UI stutter more than it helps. Flush after a *meaningful* change: every N rows, on a percentage tick, or when a phase completes. Batch the small changes; push the visible ones.

7. Synchronization and locking: protect shared data, not the UI

Once you start a background task, two threads can touch the same object at the same time: the task itself, and the session's request thread that is still handling clicks, timers and other events. If both read or write the same field — a counter, a list of row errors, an "is running" flag — you can get wrong values, half-updated state and bugs that show up once a week and never under the debugger. That is a **race condition**, and synchronization is how you prevent it.

The problem: two threads, one counter

```
private int _processed;    // written by the import task, read by a Refresh button and a
                           // timer

// _processed++ is really three steps: read, add one, write back.
// If two threads interleave those steps, one increment is lost.
```

A `lock` lets only one thread at a time run a critical section. The rules for using it are short: lock on a private object nobody else can see, keep the locked block small, and lock *every* access to the shared state — reads included.

The fix: one private lock object guards every access

```

private readonly object _sync = new object();
private int _processed;
private readonly List<string> _rowErrors = new();

private void AddProcessed(string error = null)
{
    lock (_sync)
    {
        _processed++;
        if (error != null) _rowErrors.Add(error);
    }
}

private (int processed, int errors) Snapshot()
{
    lock (_sync)
    {
        return (_processed, _rowErrors.Count); // read under the same lock
    }
}

```

For a single number, `Interlocked.Increment(ref _processed)` does the same job without a lock. Reach for `lock` as soon as more than one field must change together — the counter and the error list above are a pair, and the lock keeps them consistent as a pair.

A common shared flag: stop a double-click from starting the same job twice

```

private bool _isRunning;

private void btnImport_Click(object sender, EventArgs e)
{
    lock (_sync)
    {
        if (_isRunning) return; // second click while the first is still starting
        _isRunning = true;
        _cts = new CancellationTokenSource();
    }

    Application.StartTask(async () =>
    {
        try { await RunImport(_cts.Token); }
        finally { lock (_sync) { _isRunning = false; } }
    });
}

```

Lock the data, then marshal to the UI — never the other way round

A lock is for shared *data*. It does not replace `RunInContext`, and it must not wrap it. Take the lock only long enough to copy the values you need, release it, and then marshal those copies to the controls.

Holding a lock while you update controls and call `Application.Update` makes every other thread that

needs the lock wait on a network flush — and if the UI code ever needs the same lock, you have a deadlock.

Copy under the lock, update the UI outside it

```
private void ReportProgress()
{
    var (processed, errors) = Snapshot();           // lock taken and released inside

    RunInContext(() =>                               // no lock held here
    {
        lblStatus.Text = $"Processed {processed}, {errors} errors";
        Application.Update(this);
    });
}
```

Tool	Protects	Use it when
<code>lock</code>	Shared fields and collections	Two or more threads read or write the same state, or several fields must change together.
<code>Interlocked</code>	A single counter or flag	You only need an atomic increment, exchange or compare-and-swap.
<code>RunInContext</code>	Controls and session state	Code needs to touch a control from outside the session's own request thread.
<code>CancellationToken</code>	The task's lifetime	Another thread needs to ask the task to stop — never a shared <code>bool</code> read without a lock.

The best lock is the one you never need. Keep task data *local* to the background method — the row list, the running totals, the error list — and marshal only the finished or periodic values out to the UI. Shared mutable state should be small, named, and guarded in one place. And remember from Module 2 that anything `static` is shared across *every* session, not just this user's task, so it needs the same discipline multiplied by the number of users.



Rule of thumb

If two threads can touch the same variable, ask: "Do I need to protect this with a lock?" If the answer is yes, lock the shared data — briefly, and on every access. If you are touching controls, you do not need a lock; you need `RunInContext` and a flush.

8. Common mistakes

Mistake	Why it hurts	Do instead
Touching controls from the background thread without marshaling onto the context.	The thread has no session affinity — you update the wrong session or none at all, and may crash.	Wrap UI changes in <code>RunInContext</code> (or use <code>StartTask</code> 's session-bound thread).
Changing a control but never flushing.	The server object updates, but the change won't appear until the next natural round-trip — so it looks broken.	Call <code>Application.Update</code> after meaningful UI changes to push them now.
Doing the long work synchronously in the click handler.	Holds the round-trip open; the whole session feels frozen and Cancel can't respond.	Return the handler fast and run the work with <code>StartTask</code> (or <code>await</code> for short waits).
No cancellation.	A user who started a huge import by mistake has no way out and just waits.	Thread a <code>CancellationToken</code> through and check it each iteration.
Flushing on every iteration of a tight loop.	Floods the connection with tiny messages and can stutter the UI.	Batch changes and flush every N items or per phase.
Sharing mutable state between the task and handlers with no lock.	Race conditions and torn reads when both threads touch the same field.	Guard shared state with a lock, or keep the task's data local and marshal only the result.

9. Key vocabulary

- **Background task** — work run off the UI round-trip so the user's screen stays responsive.
- **Application context** — the ambient "which session am I?" state a thread needs before touching that user's controls.
- **Marshaling / RunInContext** — moving a piece of code back onto the correct session context before it updates controls.
- **Server push** — the server sending updates to the browser without the browser asking first, over the persistent connection.
- **Application.Update / flush** — explicitly sending queued control changes to the browser outside a normal round-trip.
- **WebSocket** — the live connection that carries pushed updates; **polling** is the fallback when it isn't available.
- **CancellationToken** — a cooperative signal a long task checks so the user can stop it cleanly.
- **Synchronization / locking** — protecting shared state when background work and handlers run concurrently.

10. Common pitfall to avoid



Common pitfall

Updating controls from the wrong thread or session context — or updating the right control but forgetting to flush, so the change never reaches the browser. Both look like "my code ran but nothing happened."

Designer & code review checklist

- ✓ The screen remains usable in the Visual Studio / Wisej.NET Designer.
- ✓ Controls are named clearly enough for a teammate to follow the event code.
- ✓ Business logic is not trapped in visual event handlers.
- ✓ Every background UI update runs on the session context and is followed by a flush.
- ✓ Long tasks accept a `CancellationToken` and restore UI state on cancel and failure.
- ✓ Failure paths are visible, logged, and explained without leaking internals.
- ✓ The deliverable can be reviewed without running the whole course.



Self-check before the lab

What UI controls are touched from the background task, and is each touch marshaled onto the session context? How often — and after what kind of change — do you call `Application.Update`? Does a `CancellationToken` reach every long loop? What happens if the user closes the session during the import? If you can answer all four, you're ready.

Companion code

`BackgroundTaskPatterns.cs` collects the patterns from this module in one compilable file. Download it and read it alongside the lab.

↓ [Download BackgroundTaskPatterns.cs](#)