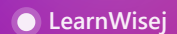


Database, File I/O, Async and External Waits

Lesson Guide · Lab / Exam Guide



Lesson Guide

The latency that CPU Usage cannot see: the Database tool for query timings and N+1 patterns, File I/O for import and export work, and .NET Async for chains that are blocked or accidentally sequential. What this means for a Wisej.NET developer diagnosing bottlenecks in a production app — and how to read this module.



Production focus

The latency that CPU Usage cannot see: the Database tool for query timings and N+1 patterns, File I/O for import and export work, and .NET Async for chains that are blocked or accidentally sequential.

1. The production problem

Module 2 measured 1,840 ms of elapsed time and found only 300 ms of CPU. Module 3 removed most of that CPU and the screen is still not fast, because the missing 600 ms was never computation at all — it was waiting. This is the most under-diagnosed category in a business application: the server is idle, the CPU graph is flat, and the user is looking at a spinner. Somewhere a query is running once per row, a details form is pulling a column nobody displays, or a click handler is blocking on `.Result` while a thread-pool thread does nothing. The tools that see this are Database, File I/O, .NET Async and Instrumentation, and this module is where WisejPerfLab's remaining latency finally has a name.

2. Key concepts

Concept	Why it matters
Database tool	Records the ADO.NET and EF Core queries a scenario issues, with their durations and order, so each query can be matched to the UI action that caused it.
N+1 query	One query to list, then one more per returned item; the list feels fine with ten rows and collapses at a thousand, and it is invisible to CPU profiling.
over-fetching	Selecting columns or rows the screen never shows — large text and binary columns are the usual offenders, and they cost query time, transfer and memory.
no-tracking query	A read-only query that does not set up change tracking, appropriate for grids and lists that display data rather than edit it.
.NET Async tool	Shows async activities and where they wait, which is how an accidentally sequential or blocked chain becomes visible.
blocking on async	Calling <code>.Result</code> or <code>.Wait()</code> from a handler; it holds a thread while waiting and, under load, produces symptoms that look like thread-pool starvation.
background export	Long file or report generation moved off the immediate click path with <code>Application.StartTask</code> , reporting progress at meaningful intervals rather than every step.

3. Lesson narrative

Matching queries to clicks

Run one scenario under the Database tool and read the query list as a story. A grid load should not issue one query per displayed row. A lookup editor should not reload unchanged reference data on every interaction. A details form should not select a large description or attachment column when the screen shows a summary. Each of those is obvious once the queries are listed next to the action that produced them, and nearly impossible to spot in code review.

WisejPerfLab has two of the three. The ticket grid issues a customer query per row, so the search that returns two hundred rows issues two hundred and one queries; and the details query selects every column of the ticket including the full conversation text. The combined fix is a single no-tracking projection that selects exactly the grid's columns, joins the customer name instead of lazy-loading it, orders and pages the result. Indexes matter for the filter and the join, and split queries are a tool for cartesian explosion — not a default to sprinkle on every query.

Async is about threads, not speed

The second scenario is the CSV export, and it fails in two different ways at once. The click handler calls an async service and blocks on `.Result`, which holds a thread for the whole generation; and the writer emits many small writes instead of streaming. Under a single user neither is fatal. Under fifty concurrent sessions, blocked threads accumulate and the symptoms look like something else entirely — the whole app slows down, including screens that have nothing to do with exports.

It is worth being precise about what `async` does. Awaiting an I/O operation frees the thread while the wait happens, which improves how many concurrent requests a server can carry. It does not make the operation finish sooner, and it does not reduce CPU work. So `async` is the right answer to "my threads are all blocked" and the wrong answer to "this method is slow". Two further cautions: never run parallel operations against objects that are not thread-safe, and keep UI updates in the session context rather than firing them from arbitrary threads.

```
public async Task<IReadOnlyList<TicketGridRow>> SearchTicketsAsync(TicketFilter f,
    CancellationToken ct)
    => await _db.Tickets
        .AsNoTracking() // read-only list: no change tracking
        .Where(t => t.Status == f.Status)
        .OrderByDescending(t => t.UpdatedAt)
        .Select(t => new TicketGridRow // projection, not the entity
        {
            Id = t.Id, Number = t.Number,
            Customer = t.Customer.Name, // joined, not lazy-loaded per row
            Status = t.Status, UpdatedAt = t.UpdatedAt
        })
        .Take(f.PageSize) // the page, not the table
        .ToListAsync(ct);
```

Moving the export off the click path

File I/O profiling shows the other half: repeated small reads and writes where a stream would do, and a template re-read on every request. The pattern that fixes it is the one the course has been building towards. Disable the button, start the work with `Application.StartTask`, stream the output rather than assembling it in memory, and push progress at meaningful steps — every ten percent, not every row. Re-enable the button and deliver the file in a `finally`, and check `IsDisposed` before touching controls, because the user may have navigated away.

Caching deserves one line of judgement here. Export templates are shared and immutable, so caching them is free capacity. Generated documents are per-user and often confidential, so caching them globally is both a memory problem and a disclosure risk. The distinction is the same one Module 3 and Module 4 drew, and it does not change for files.

Proving a wait was removed

The evidence for this module is not a CPU number. For the search, report the query count before and after, the duration of the slowest query, and the scenario's elapsed time from the probe that has been in place since Module 1. For the export, report the wall-clock time, whether the handler still blocks a thread, and the I/O pattern from the File I/O trace. Then rerun with a realistic number of concurrent sessions if you can — waiting problems are the category that behaves worst under concurrency, and a fix that only holds for one user is not a fix.



Common pitfalls

Treating low CPU as proof the app is fast. Using `.Result` in a click handler and creating thread-pool starvation symptoms. Letting a grid load produce one query per displayed row. Reloading static lookup data on every action without measuring. Caching user-private generated documents globally.



Evidence habit

Define the scenario before you open a profiler, collect from a Release build without the debugger, change one thing, then rerun the same scenario and compare. Classify the delay into one of five buckets — CPU in your code, blocking and waiting, allocation and retention, data access and I/O, or update payload — because each bucket has its own Visual Studio tool, and an optimization you cannot defend with a before and after trace is a guess.



Learning objectives

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

- ✓ Collect a Visual Studio Database trace for ADO.NET or EF Core queries triggered by Wisej.NET screens.
- ✓ Find slow queries, repeated queries, N+1 patterns and over-fetching.
- ✓ Use .NET Async and Instrumentation to identify blocked or sequential async flows.
- ✓ Profile file import/export scenarios and move expensive I/O out of immediate UI events.

Lab / Exam Guide

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



Hands-on lab

Profile the two wait-heavy scenarios in WisejPerfLab. Run the ticket search under the Database tool and match each query to the UI action: the grid load issues one customer query per displayed row and the details query pulls large text columns nobody shows. Replace both with a single no-tracking projection query that selects only the grid columns and pages the result. Then run the CSV export under File I/O and .NET Async, find the blocking .Result call in the click handler and the repeated small writes, and move generation into an Application.StartTask background workflow that streams the file and reports throttled progress instead of pushing every step.

Starting point

WisejPerfLab after Module 5: the grid is virtual and projected for display, but the query behind it still lazy-loads the customer per row, and the details form still selects every column including the conversation text. The CSV export button calls `_exportService.BuildAsync( ... ).Result` and writes the file a line at a time. The probe from Module 1 still measures all three scenarios.

Suggested approach

- ✓ Collect a Database trace of the ticket search in Release and list the queries in order next to the UI actions that produced them. Count them, and note the duration of the slowest one and the columns the details query selects.
- ✓ Replace the per-row customer lookup and the over-fetching details query with one `AsNoTracking` projection that selects exactly the grid's columns, joins the customer name, orders and pages. Check the filter and join columns are indexed before concluding the query itself is slow.
- ✓ Collect a `.NET Async` trace of the export, and an Instrumentation trace if the wait is still unclear, to prove where the click handler blocks. Confirm the `.Result` call is holding a thread rather than merely being untidy.
- ✓ Move export generation into `Application.StartTask`: disable the button, stream the output instead of assembling it in memory, push progress at meaningful intervals rather than per row, and re-enable the button and deliver the file in a `finally` after checking `IsDisposed`.
- ✓ Rerun both scenarios, and if you can, rerun the export with several concurrent sessions — waiting problems behave worst under concurrency, and a single-user improvement may not survive it.

Required deliverables

- ✓ Database trace matching queries to UI actions, with the N+1 query count before the fix
- ✓ Single no-tracking projection query with paging replacing the per-row and over-fetching queries
- ✓ File I/O and .NET Async evidence for the export, naming the blocking call and the write pattern

- ✓ Export moved to a background task that streams output and pushes progress only at meaningful steps
- ✓ Before/after query count, query duration and export wall-clock time, with remaining risks

Student review questions

- ✓ The search now issues one query instead of two hundred and one, but the elapsed time only halved. Where is the rest of the time, and which tool would you use to find out?
- ✓ Your colleague says the export is fine because `.Result` "only blocks one thread". Explain what happens at fifty concurrent exports and which screens the user would notice slowing down.
- ✓ Which of the things this lab caches is safe to share between sessions and which is not, and what is the rule you used to decide?



Evidence habit

Define the scenario before you open a profiler, collect from a Release build without the debugger, change one thing, then rerun the same scenario and compare. Classify the delay into one of five buckets — CPU in your code, blocking and waiting, allocation and retention, data access and I/O, or update payload — because each bucket has its own Visual Studio tool, and an optimization you cannot defend with a before and after trace is a guess.