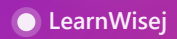


CPU Hot Paths, Event Handlers and Server-Side UI Work

Lesson Guide · Lab / Exam Guide



Lesson Guide

Expensive server-side UI code in event handlers, timers, layout refreshes and formatting logic, and the caching, batching and view-model patterns that make a click cheap again. What this means for a Wisej.NET developer diagnosing bottlenecks in a production app — and how to read this module.



Production focus

Expensive server-side UI code in event handlers, timers, layout refreshes and formatting logic, and the caching, batching and view-model patterns that make a click cheap again.

1. The production problem

Module 2 ended with a name: `TicketFormatter.FormatRow`, 68.9% self CPU, called fifty thousand times in one click. That is the shape of almost every CPU problem in a server-side UI. Nobody wrote a slow method; somebody wrote a reasonable method and then arranged for it to run once per cell, per refresh, forever. The second offender in the WisejPerfLab dashboard is the mirror image: `RebuildKpiPanel` is called once, but it disposes and recreates a panel full of controls that did not need to change. One is a call-count problem and one is a work-per-call problem, and this module is about recognising which you have and fixing it without guessing.

2. Key concepts

Concept	Why it matters
event-handler cost	A click, selection change or timer tick runs on the server and blocks that session's next event, so the amount of logic the click triggers is the amount of time the user is frozen.
flame graph	The same call-tree data drawn as width-proportional bars, which makes a single wide frame — one function eating the scenario — obvious at a glance.
self CPU vs call count	High self CPU with few calls means the body is expensive; low per-call cost with an enormous call count means the fix is to call it less, not to micro-optimize it.
precomputed view model	A small object carrying display-ready strings and values, built once by a service, so the UI layer assigns values instead of formatting them.
batched UI mutation	Setting the properties the user must see in one pass, with prepared values, instead of mutating controls inside a loop that also does work.
control rebuild	Disposing and recreating controls to refresh them; it re-runs construction, layout and theming, and it is almost always more expensive than updating the controls that exist.
immutable lookup cache	Reference data shared application-wide and never mutated; safe to cache globally, unlike per-user state which must stay in the session.

3. Lesson narrative

Finding the code that matters in the flame graph

Start where Module 2 left you: a CPU Usage trace with the timeline narrowed to the click. Switch to the flame graph and look for width. A single wide frame is a function that dominates the scenario; a wide frame made of hundreds of thin identical slices is the same function called over and over. Then use *Show Hot Path* in the call tree to confirm, and caller/callee to find out who is driving the calls.

The interpretive rules are short. High self CPU means the function body itself is expensive — look inside it for string building, reflection, serialization, image conversion or a sort. High total with low self means the cost is in children — keep descending. And if Instrumentation shows an enormous call count with a microscopic per-call time, stop optimizing the body: the profitable change is upstream, where the loop is.

In WisejPerfLab the trace shows both patterns in the same handler, which is why the module fixes both before re-measuring. Fixing only one and declaring victory is how an optimization ends up unprovable.

Why UI events get expensive

The catalogue of CPU hot paths in a Wisej.NET app is short and repetitive: formatting every grid cell on every refresh; rebuilding whole control trees instead of updating values; converting or resizing images repeatedly; parsing JSON or XML in the UI layer; reflection inside loops; sorting and filtering large in-memory collections on every event; recomputing theme or layout values; and preparing an export or a report synchronously on the click path.

They share a cause. Each is code that would be perfectly acceptable once, placed somewhere that runs per row, per cell or per tick. A formatter that costs 21 microseconds is fine; the same formatter invoked for fifty thousand rows is a second of frozen session. And because a Wisej.NET session processes its events one at a time, that second is not only the dashboard's problem — it delays whatever the user does next and consumes a core that the other sessions on the instance are competing for.

Cached view models and batched updates

The fix in the lab has two halves. First, the formatting moves out of the UI.

`DashboardService.GetSnapshot()` returns a `DashboardSnapshot` whose properties are already strings: `OpenTicketsText`, `OverdueTicketsText`, `AverageAgeText`, plus a small list of chart rows. Everything expensive — the aggregation, the culture-aware formatting, the age arithmetic — happens once, in a service that can be unit-tested and profiled on its own.

Second, the handler becomes a sequence of assignments of already-prepared values. It disables the button, mutates the labels and the chart data source once, and re-enables in a `finally`. Nothing is formatted, nothing is recreated, and the probe scope from Module 1 still wraps the whole thing so the change shows up in the same log line you have been comparing since the baseline.

```
private void btnRefresh_Click(object sender, EventArgs e)
{
    using (_probe.Measure("Dashboard", "Refresh"))
    {
        btnRefresh.Enabled = false;
        try
        {
            var snapshot = _dashboardService.GetSnapshot(); // formatted once
            lblOpen.Text = snapshot.OpenTicketsText; // assign, don't format
            lblOverdue.Text = snapshot.OverdueTicketsText;
            lblAvgAge.Text = snapshot.AverageAgeText;
            chartLoad.DataSource = snapshot.ChartRows;
        }
        finally { btnRefresh.Enabled = true; }
    }
}
```

Two cautions travel with the pattern. Cache immutable lookup data — status names, priority labels, SLA tables — at application scope, but never put mutable per-user state in a static field: in a multi-session server that is a correctness bug, not just a style preference. And do not "fix" CPU work by marking the method `async`: awaiting does not reduce computation, it just moves the same CPU onto a different server thread.

Proving the change with a second trace

The deliverable is not a faster app, it is a defensible optimization. Rerun the identical scenario — same dataset, same build, same warm-up — and collect a second CPU Usage trace. What you are looking for is a changed hot path, not merely a smaller number: `FormatRow` should have disappeared from the top of the tree, the call count should have collapsed, and whatever is now on top becomes the next candidate or the point at which you stop because the budget is met.

Record both traces in a before-and-after table with three columns that matter: total CPU for the scenario, the top function, and its call count. If the numbers moved but the hot path is unchanged, you probably measured variance. And write the remaining risk down honestly — a cached snapshot introduces a staleness window, and the reviewer should see that you know it.



Common pitfalls

Formatting controls row by row on every refresh when a cached value would do. Calling `Application.Update` inside a tight CPU loop. Treating high total CPU in framework code as a framework bug without inspecting the child calls. Hiding a CPU problem behind `async` keywords that still do the same work on server threads. Caching mutable user state in a static field.



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:

- ✓ Use CPU Usage, flame graph and call tree to locate slow event-handling paths.
- ✓ Recognize repeated formatting, reflection, serialization, image work and synchronous loops inside UI events.
- ✓ Reduce event-handler work with caching, batching, lazy loading and precomputed view models.
- ✓ Distinguish CPU-bound work from blocking time and database wait time.

Lab / Exam Guide

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



Hands-on lab

Fix the WisejPerfLab Refresh button. Profile the refresh scenario with CPU Usage, open the flame graph and Show Hot Path, and identify the two offenders: a formatter that runs per cell on every refresh and a control rebuild that recreates the KPI panel each time. Replace them with a DashboardSnapshot view model whose display strings are computed once in the service, mutate the labels and the chart data source once inside a probe scope instead of row by row, and keep the button disabled in a try/finally while the work runs. Rerun the identical scenario and capture the new hot path.

Starting point

You open WisejPerfLab with the Module 2 traces in hand: the dashboard refresh costs around 1,840 ms, `TicketFormatter.FormatRow` holds 68.9% self CPU across roughly 50,000 calls, and `RebuildKpiPanel` disposes and recreates the KPI panel on every click. The service layer currently returns raw entities and the page does all the formatting.

Suggested approach

- ✓ Reproduce the finding first: collect the CPU Usage trace yourself, open the flame graph, and confirm from caller/callee which loop drives the 50,000 formatter calls. Do not change code you have not seen in a trace.
- ✓ Introduce a `DashboardSnapshot` view model with display-ready properties — open count text, overdue count text, average age text and a small list of chart rows — and build it once inside `DashboardService.GetSnapshot()`.
- ✓ Move every formatting call out of the page and into the service, including the culture-aware number and duration formatting. Cache only immutable lookup data such as status and priority labels; keep anything per-user in the session.
- ✓ Rewrite the handler as assignments of prepared values inside the existing probe scope: disable the button, set the labels and the chart data source once, re-enable in a `finally`. Delete the KPI panel rebuild and update the existing controls instead.
- ✓ Rerun the identical scenario — same dataset, same build, same warm-up — collect a second CPU Usage trace, and build the before-and-after table from total CPU, top function and call count.

Required deliverables

- ✓ Before CPU Usage trace naming the repeated formatter and the control rebuild with their total and self CPU
- ✓ DashboardSnapshot view model with display strings precomputed in the service
- ✓ Batched refresh handler that mutates controls once inside a probe scope with try/finally re-enable
- ✓ After CPU Usage trace of the identical scenario with the new hot path

- ✓ Before/after table of total CPU, top function and call count, plus the remaining risk

Student review questions

- ✓ Both offenders were slow, but for opposite reasons. Explain which was a call-count problem and which was a work-per-call problem, and why the profiler evidence for each looks different.
- ✓ A colleague proposes marking the refresh handler `async` to make it faster. What would actually change, what would not, and how would the trace show it?
- ✓ Your snapshot is built once per refresh. What staleness or correctness risk does that introduce, and what would you have to change if the same snapshot were cached across sessions?



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.