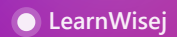


The Wisej.NET Performance Model

Lesson Guide · Lab / Exam Guide



Lesson Guide

The request and event cycle, WebSocket push, session state as the unit of capacity, the five cost buckets, and the scenario-plus-baseline discipline that every later trace is compared against. What this means for a Wisej.NET developer diagnosing bottlenecks in a production app — and how to read this module.



Production focus

The request and event cycle, WebSocket push, session state as the unit of capacity, the five cost buckets, and the scenario-plus-baseline discipline that every later trace is compared against.

1. The production problem

A support team reports that WisejPerfLab "got slow". That sentence contains no measurement, no scenario and no layer. The dashboard might be waiting on SQL, the refresh handler might be formatting fifty thousand cells, the browser might be repainting a grid that should never have been rendered in full, or the server might simply be holding two hundred sessions worth of forms that were closed an hour ago. Every one of those has a different fix and a different Visual Studio tool, and a developer who opens the profiler before deciding which one it is will collect a trace that cannot answer the question. This module builds the mental model first: what actually happens between a click in the browser and a repainted screen, which of those steps costs what, and how to write down a scenario and a budget so the next six modules have something to measure against.

2. Key concepts

Concept	Why it matters
server-side UI	Controls, their properties and their state live in the server process and the browser renders what the server describes, so every UI object you create is server memory and every UI mutation is server CPU plus a payload.
event round-trip	The full path of one user action: browser event, transport to the server, your handler, data access, control mutations, the serialized update back, browser repaint — the unit a user actually feels.
session as the unit of capacity	A session owns pages, forms, controls, timers, selections, bindings and cached data; capacity questions are answered per session, not per request, because that memory stays resident between clicks.
WebSocket push	The channel through which the server sends updates the user did not ask for; it makes live data cheap to deliver and makes an unthrottled background task expensive, because every intermediate state becomes a frame.
the five buckets	CPU in your code, blocking and waiting, allocation and retention, data access and I/O, and update payload — the classification that decides which profiler tool can even see your problem.
latency vs throughput	Latency is how long one user waits; throughput is how many sessions the server sustains. A fix that removes a wait helps latency; a fix that removes work or memory helps both, and some latency fixes cost throughput.
scenario and baseline	A named, repeatable sequence of user actions plus the environment, build configuration, dataset size and the measured numbers it produced — without one, a trace is noise and a later comparison is meaningless.

3. Lesson narrative

What one click actually costs

In a stateless web application the natural unit of measurement is a request: it arrives, it is served, the process forgets it. Wisej.NET gives you the WinForms programming model instead — a **Page** or **Form** with controls you set properties on — and pays for it by keeping that object graph alive on the server for as long as the user is connected. The browser holds a rendering of the UI; the server holds the UI.

Follow one click through. The browser raises an event and sends it over the open connection. The server locates the session, dispatches into your handler, and from there everything is ordinary C#: you query a database, build strings, set **Text** on a label, assign a **DataSource**, add rows. Wisej.NET records those property changes and, when the handler returns, serializes the resulting difference and sends it to the browser, which repaints. Five distinct costs sit on that path — browser rendering, network and payload, your handler's CPU, database and I/O wait, and any background work the click started — and a symptom that looks identical to the user ("the screen is slow") can come from any of them.

Two consequences matter immediately. First, client events for a session are processed one at a time, so a handler that takes four seconds freezes that user for four seconds, whatever else the server is doing. Second, a handler that burns CPU is not just slowing its own session; it is competing for the same cores as every other session on the instance. That is why the course measures per-scenario cost and then reasons about per-session capacity, instead of declaring a fast single-user click "fast enough".

Five buckets, five different tools

The classification is not academic — it is how you choose a tool. CPU work in your own code is what CPU Usage and the flame graph show. Blocking and waiting time is largely invisible to CPU Usage and is what Instrumentation and .NET Async are for. Allocation pressure and retained memory need .NET Object Allocation and Memory Usage snapshots respectively, and they are different problems: a scenario can allocate heavily and retain nothing, or allocate almost nothing and leak a whole form. Data access and I/O latency belong to the Database and File I/O tools. Update payload cost is the one bucket Visual Studio cannot see at all, and browser developer tools have to supply the evidence.

The habit that follows is a question you ask before starting a collection: *which bucket do I believe this is, and what would prove me wrong?* A slow screen with low server CPU is not a CPU problem no matter how many CPU traces you collect. Module 2 turns this into a tool-selection matrix and a repeatable collection workflow; this module is where you learn to make the guess explicitly so the trace can refute it.

Making WisejPerfLab measurable

The lab creates WisejPerfLab, the support-operations dashboard the whole course tunes, and then does the unglamorous work of making it measurable. The instrument is a small `ScenarioProbe` service registered in the host builder and injected into pages. `Measure()` returns an `IDisposable` scope that starts a `Stopwatch`, writes a structured "PERF start" log line, and on disposal writes "PERF end" with the elapsed milliseconds and the row count. Structured logging matters here: the scenario name, the user action and the row count are separate fields, so the same three scenarios can be filtered and compared across every later run.

```
using (_probe.Measure("Dashboard", "Refresh", rows.Count))
{
    // everything inside this scope is one measured scenario:
    // query, projection, control mutation and the update that follows
    RefreshKpis(rows);
    RefreshGrid(rows);
}
// PERF end Dashboard Refresh elapsedMs=1840 rows=5000
```

Three scenarios get a probe: dashboard refresh, ticket search and customer tree expand. They were chosen because they land in different buckets — the refresh will turn out to be CPU in formatting code, the search will turn out to be database time, the tree expand will turn out to be payload and retained control count — and by Module 7 you will have before-and-after numbers for all three.

Baselines and budgets, written down

A baseline is the scenario plus everything that could change its result: the machine, the build configuration, the dataset size, whether the database is local, which browser, how many sessions were connected, and

whether the app was warmed up first. Skip any of those and the comparison you make in Module 3 will be an argument, not evidence. Release matters more than developers expect — Debug changes JIT optimization, adds overhead and can produce call stacks that send you after the wrong function entirely.

A budget is the other half: an acceptance threshold per scenario, written before the tuning starts.

"Dashboard refresh under 800 ms of server handler time after warm-up, with fewer than 40 MB retained per idle session" is a budget. It gives every later module a pass or fail instead of a feeling, it tells you when to *stop* optimizing, and in Module 7 it becomes the source of the health-check thresholds and the production alerts. Write the budget in the same document as the baseline, and note next to each threshold which tool would prove it.



Common pitfalls

Profiling a Debug build and treating the result as production evidence. Optimizing a method before the end-to-end scenario is defined. Ignoring retained session memory because CPU looks acceptable. Mistaking network or WebSocket delay for server CPU. Measuring one session and assuming the number holds at two hundred.



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:

- ✓ Explain the Wisej.NET request/event cycle, WebSocket push, session state and server memory model.
- ✓ Separate browser rendering time, network/payload time, server event-handler time, database time and background-task time.
- ✓ Create a repeatable performance scenario with baseline metrics and acceptance thresholds.
- ✓ Identify which performance questions require Visual Studio CPU, memory, database, async, counters or instrumentation tools.

Lab / Exam Guide

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



Hands-on lab

Create the WisejPerfLab solution with a support dashboard, a ticket grid and a customer tree fed by seeded data, then make it measurable. Add a ScenarioProbe service that wraps a scenario in a Stopwatch scope and writes structured PERF start/end logs with the scenario name, the user action, the elapsed milliseconds and the row count, register it in the host builder, and wrap three scenarios in it: dashboard refresh, ticket search and customer tree expand. Run each scenario once to warm up, then record the baseline and write a performance budget that states the environment, build configuration, dataset size and an acceptance threshold per scenario.

Starting point

You start from an empty Wisej.NET application and Visual Studio 2022 with the profiling tools installed. Nothing exists yet: this lab builds the three screens the rest of the course tunes, and seeds them with enough data to be slow on purpose — roughly 50,000 tickets and a customer hierarchy several thousand nodes deep. Do not fix anything you notice. The point of Module 1 is to measure the app as built.

Suggested approach

- ✓ Create the WisejPerfLab solution and three screens: a dashboard `Page` with KPI labels, a refresh button and a chart; a ticket grid bound to the seeded ticket data; and a customer `TreeView`. Seed the data from a script or an `IHostedService` so every run starts from the same dataset.
- ✓ Write `ScenarioProbe` as a small service: `Measure(scenario, userAction, rowCount)` returns an `IDisposable` scope that starts a `Stopwatch` and logs "PERF start", and on `Dispose` logs "PERF end" with `elapsedMs` and `rows` as structured fields, not string concatenation.
- ✓ Register it as a singleton in the host builder alongside the console logging provider, and make the container reachable from your Wisej.NET pages so each screen can obtain the probe it needs.
- ✓ Wrap the three scenarios: dashboard refresh, ticket search and customer tree expand. Put the probe scope around the whole user action — query, projection and control mutation — not just the query, because the UI work is part of what the user waits for.
- ✓ Switch to Release, run each scenario once to absorb startup and JIT noise, then run each three times and record the median. Write the baseline note and the budget together, and against each threshold name the Visual Studio tool that would prove or disprove it.

Required deliverables

- ✓ WisejPerfLab solution with a dashboard, ticket grid and customer tree over seeded data
- ✓ ScenarioProbe service writing structured PERF start/end logs with elapsed milliseconds and row counts
- ✓ Three instrumented scenarios: dashboard refresh, ticket search and customer tree expand

- ✓ Baseline note recording environment, build configuration, dataset size and warm-up procedure
- ✓ Performance budget with an acceptance threshold per scenario and the tool that would prove each one

Student review questions

- ✓ Your dashboard refresh logs 1,900 ms but the server's CPU barely moves during the scenario. Which of the five buckets does that rule out, which does it point at, and what is the cheapest next measurement?
- ✓ The probe measures the handler. Name two costs the user experiences on the same click that your `elapsedMs` number does not include, and say how you would measure each.
- ✓ Why does the baseline note have to record the dataset size and the build configuration, when the scenario name and the elapsed time seem to be the interesting part?



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.