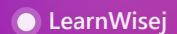


Visual Studio Profiling Workflow for Wisej.NET

Lesson Guide · Lab / Exam Guide



Lesson Guide

Running the Performance Profiler against a Wisej.NET app the right way: Release builds without the debugger, the tool selection matrix, attaching to dotnet or w3wp, and reading the summary timeline, call tree, hot path and caller/callee views. What this means for a Wisej.NET developer diagnosing bottlenecks in a production app — and how to read this module.



Production focus

Running the Performance Profiler against a Wisej.NET app the right way: Release builds without the debugger, the tool selection matrix, attaching to dotnet or w3wp, and reading the summary timeline, call tree, hot path and caller/callee views.

1. The production problem

Module 1 left WisejPerfLab with a number and no explanation: the dashboard refresh takes 1,840 ms. The obvious next step is to profile it, and the obvious next step is where most investigations go wrong. A trace collected under the debugger measures the debugger. A trace started before the app is warm measures JIT. A trace that covers two minutes of clicking measures everything and proves nothing. A trace collected with four tools enabled at once measures the profiler's own overhead. This module is the mechanical discipline that makes a trace admissible evidence: the right build, the right tool for the bucket you suspect, the right process when the app is not started by Visual Studio, one scenario per collection, and a report you know how to read.

2. Key concepts

Concept	Why it matters
Performance Profiler (Alt+F2)	The post-mortem collection path: it can start the project without the debugger, collect a whole scenario and produce a saved report, which is where real evidence comes from.
Diagnostic Tools window	The live, debugger-attached view; useful for a quick look while stepping through code, misleading as a source of timings because the debugger changes what the runtime does.
tool selection matrix	CPU Usage for computation, Instrumentation for call counts and wall-clock, Object Allocation for churn, Memory Usage for retention, Database for queries, File I/O for files, .NET Async for await chains, .NET Counters for live health.
.diagsession	The saved trace file; naming it with module, scenario, build, dataset and timestamp is what makes a before-and-after comparison possible weeks later.
summary timeline	The CPU or memory graph across the whole collection; narrowing the selection to the exact user action is the step that removes startup and warm-up from the call tree.
hot path	The single most expensive chain of calls in the selected range; Show Hot Path expands it so you can find your own code under the framework frames.
self vs total CPU	Total includes children, self is the function body alone: high self means the method itself is expensive, high total with low self means the cost is in what it calls.

3. Lesson narrative

Release, no debugger, one scenario

The default workflow has four fixed steps before anything interesting happens. Switch the configuration to Release. Start the app and run the scenario once so startup and JIT compilation are already paid for. Open *Debug > Performance Profiler* with Alt+F2 and tick exactly one tool. Then start collection, perform only the scenario under test, and stop promptly.

Each rule earns its place. Debug builds disable optimizations, inline nothing and can invent call stacks that make a trivial helper look like a hot path. The debugger adds its own overhead and changes timing. Warm-up matters because the first execution of any method includes JIT time, which is real but is not the cost the user feels on the hundredth refresh. And one tool at a time matters because collection is not free: running Object Allocation and Instrumentation together adds enough overhead to move the hot path.

The last rule is the one that gets broken most often: *one scenario per trace*. A collection that contains a login, three refreshes, a search and some scrolling produces a call tree in which every interesting number is an average of unrelated work. Close the extra browser tabs, click once, stop.

Choosing the tool from the bucket

Module 1's five buckets map onto the Performance Profiler's tool list almost one to one, and the mapping is the whole point of this module. Use *CPU Usage* first whenever the symptom is high CPU or slow computation — it samples stacks cheaply and shows where the processor went. Reach for *Instrumentation*

when you need exact call counts or wall-clock time, or when you suspect time is being spent waiting rather than computing: it measures entry and exit of every instrumented function, so it sees blocked time that sampling misses.

Use *.NET Object Allocation* when a scenario seems to churn memory, and *Memory Usage* snapshots when you suspect something is retained after the user closed it. Use *Database* when EF Core or ADO.NET queries may dominate, *File I/O* for imports, exports and report generation, and *.NET Async* when an async chain is slower than the sum of its awaits suggests it should be. *.NET Counters* is the odd one out: it is a live view of CPU, GC heap, allocation rate, exception rate and thread-pool health, which makes it the right tool for watching an app under load rather than dissecting one click.

In WisejPerfLab the lab runs the first two. CPU Usage answers "where did the processor go during the refresh"; Instrumentation answers "how many times was that called, and how much of the 1,840 ms was actually spent computing". Two traces, one scenario, and a conclusion that survives review.

Profiling a process Visual Studio did not start

When the startup project is the Wisej.NET app, the profiler launches it for you and there is nothing to choose. Production-like hosting is different. Under Kestrel the app is a `dotnet` process; under IIS or IIS Express it is `w3wp` or `iisexpress`, one per application pool. The Performance Profiler can attach to a running process, and the only real trap is attaching to the wrong one: on a machine with several sites, confirm the process id and the app pool, then send one request and check that the collection actually shows activity before running the scenario for real.

WebSocket scenarios need one extra thought. If the browser connects while the trace is still starting, the connection handshake and the initial render land in your measurement. Either start the browser after collection is under way, or record a separate warm-up trace and compare like with like.

Reading the report without being fooled

Every report opens on the summary timeline. Before looking at a single function name, drag a selection around the exact interval of the user action and let the views recompute; this is what removes startup, page load and idle time from the numbers. Then work top down: the call tree with *Show Hot Path* for the dominant chain, the flame graph for the same information as a shape, the functions view sorted by self CPU for expensive bodies, and caller/callee to learn who is calling an expensive function and how often.

In a Wisej.NET trace your own code sits some way down the stack, below Wisej.NET's event dispatch and the ASP.NET Core hosting frames. That is normal and is not a framework bug. The rule is to scroll past the plumbing to the first frame in your own namespace and start reasoning there. If the framework frame above your code is expensive, ask what your code is asking it to do — usually it turns out to be a mutation repeated thousands of times rather than a slow framework.

```
// Trace note saved beside the .diagsession
// Module:    02
// Scenario:  DashboardRefresh_5000Tickets
// Build:     Release, x64, no debugger, warm
// Hosting:   Kestrel, local SQL Server  Browser: Edge
// Tool:      CPU Usage, then Instrumentation follow-up
// Budget:    < 800 ms server handler time
```

Finally, respect the noise floor. A change of thirty milliseconds between two runs of a two-second scenario is not an improvement, it is variance. Repeat the scenario, take the median, and only claim a win you can reproduce.



Common pitfalls

Profiling startup when the complaint is a slow button click. Enabling every tool at once and drowning the scenario in collection overhead. Comparing a Debug trace with a Release trace. Leaving the profiler running for a long session until nobody can find the interaction. Attaching to the wrong `w3wp` and profiling an idle app pool.



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:

- ✓ Profile Release builds with Debug > Performance Profiler / Alt+F2.
- ✓ Collect CPU Usage, Instrumentation, .NET Object Allocation, Memory Usage, .NET Counters, Database, File I/O and .NET Async traces.
- ✓ Attach to a running Wisej.NET process when needed and identify the correct process.
- ✓ Interpret call tree, hot path, caller/callee, function list, flame graph, allocation tree and snapshot comparisons.

Lab / Exam Guide

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



Hands-on lab

Profile the slow WisejPerfLab dashboard refresh properly. Switch to Release, warm the app up once, then collect a CPU Usage trace of that one scenario with Alt+F2, narrow the summary timeline to the click itself and use Show Hot Path to find the top path below the Wisej.NET dispatch frames. Collect a second Instrumentation trace of the same scenario to get call counts and wall-clock time, decide from the two traces whether the cost is CPU inside your code or time spent waiting, and save both `.diagsession` files with a name that carries the module, scenario, build, dataset and timestamp.

Starting point

You open WisejPerfLab from Module 1 with its three instrumented scenarios and its `Baseline.md`. The dashboard refresh logs around 1,840 ms for 5,000 rows and nobody yet knows why. Nothing in the application changes in this lab: it is entirely about collecting evidence that the next module can act on.

Suggested approach

- ✓ Switch the solution to Release, close every browser tab except the app, reset the seeded data to the baseline dataset, and run the refresh once without recording so JIT and startup are paid for.
- ✓ Open *Debug > Performance Profiler* with Alt+F2, tick *CPU Usage* and nothing else, start collection, click Refresh exactly once, and stop as soon as the screen settles.
- ✓ In the report, drag a selection on the summary timeline around the click itself, then open the call tree, turn on *Show Hot Path*, and scroll past the ASP.NET Core and Wisej.NET dispatch frames to the first function in your own namespace. Record its total and self CPU, and use caller/callee to see who calls it and how often.
- ✓ Collect a second trace of the identical scenario with *Instrumentation*. Compare call counts and wall-clock time against the sampled CPU picture: a function with a large call count and a tiny per-call cost is a different problem from a function with a long body.
- ✓ Decide which bucket the evidence supports, write the trace note beside each `.diagsession`, and name both files so module, scenario, build, dataset and timestamp are readable without opening them.

Required deliverables

- ✓ Release CPU Usage trace of the dashboard refresh scenario, timeline narrowed to the click
- ✓ Hot path screenshot naming the top application function and its self versus total CPU
- ✓ Instrumentation trace of the same scenario with call counts and wall-clock time
- ✓ Trace note stating course, module, scenario, build, hosting, tool, browser and expected budget
- ✓ One-paragraph suspected root cause that says which of the five buckets the evidence points at

Student review questions

- ✓ Your CPU Usage trace accounts for only 300 ms of the 1,840 ms the probe measured. What does that gap mean, and which tool would you collect next?
- ✓ A framework method sits at the top of the hot path with a very high total CPU and almost no self CPU. Explain what that means and what you would look at instead of filing a framework bug.
- ✓ Why does a second person need the trace note to reproduce your result, when the `.diagsession` already contains the whole trace?



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.