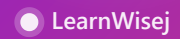


Scale, Health Checks, Load Balancing and Final Tuning

Lesson Guide · Lab / Exam Guide



Lesson Guide

Turning measurements into a capacity model, HealthCheck.json thresholds, session affinity and WebSocket-aware load balancing, production counters, and the final performance report that defends every change. What this means for a Wisej.NET developer diagnosing bottlenecks in a production app — and how to read this module.



Production focus

Turning measurements into a capacity model, HealthCheck.json thresholds, session affinity and WebSocket-aware load balancing, production counters, and the final performance report that defends every change.

1. The production problem

Six modules of profiling have produced a faster WisejPerfLab and a folder of traces. None of that answers the question the operations team will ask on the day of the rollout: *how many users fit on a server, and what happens when one is full?* Profiling on a developer machine tells you the cost of a scenario; production needs that cost converted into a capacity model, thresholds that refuse new sessions before the process is in trouble, a deployment shape that respects server-side session state, and monitoring that will notice if any of the six fixes regresses. That translation is this module, and the capstone report is what makes the work reviewable by someone who never opened Visual Studio.

2. Key concepts

Concept	Why it matters
capacity model	Memory retained per idle session and CPU per common scenario, multiplied by expected peak concurrency and divided into the instance's resources, with headroom kept deliberately.
HealthCheck.json	Configuration through which a Wisej.NET server can report itself unavailable based on session count, memory and CPU, so a load balancer stops sending it new users.
503 with retry-after	The response an unhealthy instance returns to a <i>first</i> request; existing sessions keep running on their node while new users are routed elsewhere.
session affinity	Consistent routing of a client to the instance that holds its session state, usually implemented with sticky cookies; without it a Wisej.NET session cannot survive a second request.
WebSocket-aware balancing	Load-balancer configuration that allows the upgraded connection through; when WebSocket is unavailable the app falls back to polling and real-time push is lost.
production counters	Lightweight ongoing metrics — CPU, GC heap, allocation rate, exception rate, thread-pool health — that show a regression without a profiler attached.
final performance report	Scenarios, environment, baseline evidence, root cause, change, after evidence, remaining risks, thresholds and the monitoring plan, in one reviewable document.

3. Lesson narrative

From a measurement to a number of sessions

The capacity model is arithmetic over numbers you already have. Module 4 produced megabytes retained per idle session; Modules 3 and 6 produced CPU and wait cost per common scenario. Multiply the memory figure by the concurrency you expect at peak and compare it with the memory the instance can actually use, keeping headroom for spikes, garbage collection and the operating system rather than planning to the last megabyte. Do the same for CPU: cost per scenario times the rate at which scenarios are executed at peak.

Two honesty checks belong here. First, an idle session and an active one are not the same, so say which you measured and how you drove the sessions. Second, a number derived from a developer laptop needs a note about the hardware it will actually run on. A capacity model that states its assumptions can be corrected; one that presents a single number as fact cannot.

Health checks with justified thresholds

Wisej.NET health checks let a server declare itself unavailable on session count, memory or CPU. The mechanism is simple; the value of this module is that your thresholds come from measurement rather than from a blog post. If a session retains 22 MB and the instance can give the app 4 GB with headroom, then a `maxSessions` of 150 is a number you can defend — and you can show the calculation.

```
{
  "enabled": true,
  "maxMemory": 70,      // % of available memory
  "maxSessions": 150,    // from 22 MB retained/session + headroom
  "maxCPU": 85,          // % sustained
  "returnCode": 503,
  "retryAfter": 10
}
```

What the check does in practice is worth stating precisely, because it is often misunderstood: an instance over its limits returns 503 to a request that would *start* a session, so the load balancer sends that user to another node, while the sessions already on the instance continue normally. It is a mechanism for graceful degradation, not a kill switch — and it is not a substitute for fixing the leak that is filling the process in the first place.

Deploying something that owns state

A Wisej.NET session owns server-side UI state, which decides the deployment shape. Requests from a client must return to the instance holding its session, so the load balancer needs session affinity — sticky cookies or another consistent routing mechanism. Scaling out without affinity does not degrade a Wisej.NET app gracefully; it breaks it.

WebSocket support is the second requirement. The balancer and any reverse proxy in front of it must allow the connection upgrade; where WebSocket is unavailable the app can still work through polling, but real-time push is lost and the update pattern changes. Write that in the deployment notes rather than discovering it during an incident. And benchmark with production settings — debug options, minification and compression all differ, and a measurement taken with the wrong ones is not comparable to the traces the report is built on.

Monitoring, and the report that survives you

Visual Studio is for investigations, not for watching production. Ongoing health comes from .NET counters and runtime metrics — CPU, GC heap size, allocation rate, exception rate, thread-pool health — plus application-level structured logs around the scenarios you instrumented in Module 1. That probe is still in the code, and its **PERF end** records are the cheapest possible regression detector: if the dashboard refresh starts logging 1,800 ms again, you will know before a user reports it.

The capstone report ties it together: scenario definitions, environment, baseline evidence, root-cause explanations, the changes, the after evidence, remaining risks, health-check thresholds and the deployment checklist. Its test is a question you should ask yourself before submitting it — could a colleague who never opened your profiler reproduce a measurement, understand why each change was made, and know which production metric would tell them the problem came back? If yes, the tuning work is finished. If not, the app may be faster, but the knowledge is still only in your head.



Common pitfalls

Scaling horizontally without session affinity. Choosing `maxSessions` without measuring per-session memory. Benchmarking with debug, minification or compression settings that differ from production. Treating load balancing as a substitute for fixing leaks and slow scenarios. Leaving profiler traces running as if they were monitoring.



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:

- ✓ Estimate per-session memory and per-scenario CPU cost from measurements.
- ✓ Configure Wisej.NET health checks with max sessions, max memory and max CPU thresholds.
- ✓ Explain why Wisej.NET deployments require session affinity and WebSocket-aware load balancing.
- ✓ Build a final performance report with baseline, fixes, retest results, risks and monitoring plan.

Lab / Exam Guide

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



Hands-on lab

Deliver the WisejPerfLab capstone. Derive a capacity model from your own measurements: memory retained per idle session, CPU per common scenario, expected peak concurrency and the headroom you are keeping. Write a `HealthCheck.json` whose `maxSessions`, `maxMemory` and `maxCPU` come from those numbers and explain each threshold, returning 503 with a `retry-after` so a load balancer routes new users elsewhere while existing sessions keep running. Write the deployment notes for two instances behind a load balancer with session affinity and WebSocket support, and assemble the final report: scenarios, environment, baseline evidence, root causes, changes, after evidence, remaining risks and the production counters that will show if a regression returns.

Starting point

WisejPerfLab after six modules: the dashboard refresh, the ticket search, the tree expand and the export are all measurably faster, the detail form no longer leaks, and you have before-and-after evidence for each. What does not exist yet is anything an operations team could act on — no capacity number, no health check, no deployment notes and no monitoring plan.

Suggested approach

- ✓ Collect the inputs from your own reports: retained megabytes per idle session from Module 4, CPU and wait cost per scenario from Modules 3 and 6, and the expected peak concurrency for the app. State which measurements are idle and which are active.
- ✓ Build the capacity model as visible arithmetic — memory per session times concurrency against usable instance memory, CPU per scenario times peak scenario rate — and keep explicit headroom for spikes, garbage collection and the operating system rather than planning to the last megabyte.
- ✓ Write `HealthCheck.json` with `maxSessions`, `maxMemory`, `maxCPU`, the 503 return code and a `retry-after`, and write one sentence per threshold explaining the measurement it came from. Verify that an instance over its limits refuses new sessions while existing ones continue.
- ✓ Write the deployment notes for two instances behind a load balancer: session affinity and how it is implemented, WebSocket support through every proxy in the path, what changes if WebSocket is unavailable, and the production settings that measurements must use.
- ✓ Assemble the final report from the six modules' evidence, then finish with the monitoring plan: which counters, which thresholds, which alerts, and which `PERF` log records would show each specific regression returning.

Required deliverables

- ✓ Capacity model deriving sessions per server from measured per-session memory and per-scenario CPU

- ✓ HealthCheck.json with maxSessions, maxMemory, maxCPU, return code and retry-after, each threshold justified
- ✓ Deployment notes covering session affinity, WebSocket support and the polling fallback
- ✓ Final performance report with baseline, root cause, change and after-evidence for every module fix
- ✓ Monitoring plan naming the counters, thresholds and alerts that would catch each regression

Student review questions

- ✓ Your health check refuses new sessions at 150. A colleague asks why not 300, since the process has not crashed at 300 in testing. Answer with your own numbers.
- ✓ A deployment goes out behind a load balancer without session affinity. Describe exactly what a user experiences, and why this is a correctness failure rather than a slowdown.
- ✓ Six months from now the dashboard refresh is slow again. Which single production signal would tell you first, and what would you collect next?



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.