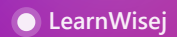


Memory, Allocations and Session Leaks

Lesson Guide · Lab / Exam Guide



Lesson Guide

Allocation pressure versus retained memory, the snapshot-compare workflow, the Wisej.NET retention patterns (static events, timers, background tasks, global caches) and the disposal checklist that proves a leak is gone. What this means for a Wisej.NET developer diagnosing bottlenecks in a production app — and how to read this module.



Production focus

Allocation pressure versus retained memory, the snapshot-compare workflow, the Wisej.NET retention patterns (static events, timers, background tasks, global caches) and the disposal checklist that proves a leak is gone.

1. The production problem

The WisejPerfLab dashboard now clears its CPU budget, and the app still falls over after four hours in production. Memory problems behave differently from CPU problems: they are invisible in a two-minute demo, they never show up as one slow click, and they are the reason a server that comfortably holds eighty sessions in the morning is refusing connections by mid-afternoon. There are two of them in the lab, and they are not the same problem. The ticket search allocates far more than it needs to, which costs GC time and CPU while it runs. The ticket detail form is never actually released after the user closes it, which costs memory forever. One is found with an allocation trace, the other only with two snapshots and a path to root.

2. Key concepts

Concept	Why it matters
allocation pressure	How much memory a scenario creates while running; high pressure raises GC frequency and steals CPU even when nothing leaks.
retained memory	What is still reachable after the scenario should be finished; this is what multiplies by session count and decides capacity.
.NET Object Allocation	The tool that records every managed allocation during a scenario and groups it by type and by allocating call path.
Memory Usage snapshots	Heap captures taken at chosen moments; comparing snapshot A and snapshot B shows which types gained instances that were never released.
path to root	The chain of references keeping an object alive; it is the actual evidence of a leak, and the only way to prove the leak is gone.
retention root	The thing at the top of that chain — a static field, a static event, a running timer, a background task or a long-lived cache.
per-session memory budget	A number, in megabytes retained per idle session, that turns "the app uses a lot of memory" into a pass or fail and later into a health-check threshold.

3. Lesson narrative

Why memory is a capacity question in Wisej.NET

Wisej.NET keeps server-side UI objects alive for the duration of the session. That is what makes the programming model pleasant, and it is what makes a leak expensive. A modal form that is never released, a `BindingSource` still holding fifty thousand rows, a static event that pins a page, a timer that keeps ticking after the panel is hidden: each is a few megabytes, invisible on a developer laptop and decisive at two hundred sessions.

The mental model to carry into the profiler is that memory has two separate questions attached to it. *How much did this scenario churn?* is about GC cost and CPU. *What is still alive now that the user has closed everything?* is about capacity. A scenario can be terrible at one and perfect at the other, so a report that answers only one of them has not cleared the app.

Finding the allocation pressure

For allocation, run the scenario under *.NET Object Allocation* in Release, exactly once, and open the allocation views. The type view answers "what is being created" and the allocation call tree answers "who is creating it". In the ticket search the answer is unflattering: row model objects and their formatted strings are rebuilt every time the grid redraws, so a single search allocates several times the data it displays.

The fix is the same discipline as Module 3, applied to objects instead of CPU: project once into a lightweight row type and hand that to the grid, instead of constructing new objects on each redraw. What

you are looking for in the after trace is a drop in both total allocated bytes and the instance count of the offending type — and, ideally, a visibly calmer GC line.

Finding the retention

Retention needs a different workflow, and the order of the steps is the whole technique. Start the app in Release and warm it up. Take snapshot A. Run the scenario that should be neutral — open the ticket detail form and close it, fifty times, so one leaked instance becomes fifty and cannot be mistaken for noise. Return the app to a genuine idle point. Take snapshot B, and compare.

In the comparison you are looking for types whose instance count grew and never came back:

`TicketDetailForm` at fifty instances is not ambiguous. Select the type, open the instances, and inspect the path to root. In WisejPerfLab two roots show up. One is a static event — the form subscribed to `GlobalTicketBus.TicketChanged` in its constructor and never unsubscribed, so the static delegate list holds every form ever opened. The other is a `System.Timers.Timer` the form starts to refresh its header; its callback captures the form, so as long as the timer is alive the form is too.

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        GlobalTicketBus.TicketChanged -= OnTicketChanged; // the static root
        _refreshTimer?.Stop(); // the timer root
        _refreshTimer?.Dispose();
        bindingSource1.DataSource = null; // release the rows
        _largeImage?.Dispose();
    }
    base.Dispose(disposing);
}
```

Both fixes belong in the same place: an overridden `Dispose` that unsubscribes what it subscribed, stops and disposes what it started, and clears the bindings holding large data. Anything that touches the UI from a timer or a background task should also check `IsDisposed` before updating controls, because the callback can outlive the form by a few milliseconds even when disposal is correct.

Proving it, and writing the budget

A memory fix is not finished when the code looks right. Repeat the identical open-and-close scenario, take snapshot B again, and show two things: the instance count of `TicketDetailForm` returns to its baseline, and the old path to root no longer exists. The second is the important one — a count that drops because a garbage collection happened to run is not proof, and forcing collections until the numbers look acceptable is how a leak survives a review.

Finish with the two documents the rest of the course needs. The per-session memory budget states how many megabytes an idle session may retain, measured on your own app rather than guessed; Module 7 turns that number into the `maxSessions` threshold of the health check. The disposal checklist is the reviewable version of the same knowledge: unsubscribe events, stop and dispose timers, cancel background work, clear bindings, dispose images and streams, release large lists, and never store mutable user state in a static field.



Common pitfalls

Measuring allocations and calling the app leak-free without ever checking retention. Forcing garbage collections until the numbers look good instead of reading paths to root. Keeping a `DbContext`, a `DataTable` or an image cache in the session forever. Subscribing to a static event from a page and never unsubscribing. Updating controls from a timer callback without checking `IsDisposed`.



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 .NET Object Allocation to find allocation-heavy code paths during a scenario.
- ✓ Use Memory Usage snapshots to compare heap state before and after closing Wisej.NET UI objects.
- ✓ Identify retention roots caused by static fields, events, timers, caches and long-lived services.
- ✓ Create a per-session memory budget and a disposal checklist.

Lab / Exam Guide

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



Hands-on lab

Attack both halves of the memory problem in WisejPerfLab. Run the ticket search scenario under *.NET Object Allocation* and find the row models and strings allocated per search, then reduce them by projecting once instead of rebuilding objects per redraw. Next take a *Memory Usage* snapshot after warm-up, open and close the ticket detail form fifty times, take a second snapshot and compare: the retained forms are rooted by a static `GlobalTicketBus.TicketChanged` subscription and a refresh timer that was never stopped. Unsubscribe and stop both in an overridden `Dispose`, clear the `BindingSource` data source, then rerun the open/close scenario and show the retained heap returning close to snapshot A.

Starting point

WisejPerfLab after Module 3: the dashboard refresh is inside its CPU budget, the ticket grid still binds full entities, and the ticket detail form subscribes to `GlobalTicketBus.TicketChanged` in its constructor and starts a `System.Timers.Timer` to refresh its header. Neither is undone anywhere. The Module 1 budget already carries a retained-memory target that the app does not meet.

Suggested approach

- ✓ Run the ticket search once under *.NET Object Allocation* in Release. Use the type view to find what dominates and the allocation call tree to find who creates it, and record total allocated bytes plus the instance count of the top type.
- ✓ Project once into a lightweight row type and bind that, so the row objects and their display strings are created a single time per search rather than rebuilt on each redraw. Rerun the allocation trace on the identical scenario.
- ✓ Switch to *Memory Usage*. Warm the app up, take snapshot A, then open and close the ticket detail form fifty times so a single leaked instance becomes fifty, return the app to a genuine idle point and take snapshot B.
- ✓ Compare the snapshots, select the form type whose instance count grew, and read the path to root for two instances. Capture both roots: the static event subscription and the live timer holding the callback's captured form.
- ✓ Override `Dispose(bool)` on the form to unsubscribe the static event, stop and dispose the timer, clear the `BindingSource` data source and dispose any images; add an `IsDisposed` check before the timer callback touches controls. Rerun the identical open/close scenario and prove both the count and the path to root are gone.

Required deliverables

- ✓ Allocation trace naming the top allocating type and call path for the search scenario, before and after
- ✓ Snapshot A/B comparison showing the retained detail form instance count before the fix

- ✓ Path-to-root screenshot identifying the static event subscription and the live timer
- ✓ Dispose override that unsubscribes the static event, stops and disposes the timer and clears the binding
- ✓ Per-session memory budget and a disposal checklist, with the after-snapshot proving the retention path is gone

Student review questions

- ✓ Your allocation trace improved dramatically but the snapshot comparison is unchanged. What kind of problem did you fix, what kind did you not, and how would a user notice the difference?
- ✓ Why does the lab open and close the form fifty times rather than once, and what would you conclude from a single instance appearing in snapshot B?
- ✓ The timer callback is one line long and only sets a label. Explain precisely how it keeps an entire form alive, and what an `IsDisposed` check does and does not solve.



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.