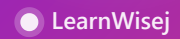


Data Controls, Browser Payloads and Large UI Surfaces

Lesson Guide · Lab / Exam Guide



Lesson Guide

Grids, lists, trees and repeaters at scale: virtual mode, paging, projection and lazy node loading, and using browser network evidence next to Visual Studio evidence to size the update payload. What this means for a Wisej.NET developer diagnosing bottlenecks in a production app — and how to read this module.



Production focus

Grids, lists, trees and repeaters at scale: virtual mode, paging, projection and lazy node loading, and using browser network evidence next to Visual Studio evidence to size the update payload.

1. The production problem

Large UI surfaces are the most common source of Wisej.NET performance problems, and they are the bucket the profiler cannot fully see. The WisejPerfLab ticket grid binds fifty thousand entities with their navigation properties; the customer tree builds every node at startup. On a desktop that pattern was acceptable, because the control and the data lived in the same process as the screen. Here the same code pays four times: server memory to hold the objects, server CPU to build and serialize the UI state, network to carry the update, and browser time to render it. This module is where the best optimization is usually not a faster algorithm but *less UI existing at once*.

2. Key concepts

Concept	Why it matters
projection row	A small type holding only the columns a screen displays, bound instead of the entity graph, so neither the server nor the browser carries data nobody looks at.
virtual mode	The grid holds a row count rather than row objects and asks for cell values as they are needed, which removes the per-row objects and the rendering of rows nobody scrolled to.
CellValueNeeded	The event that supplies a value on demand in virtual mode; it runs very often, so it is a hot path by construction and must return quickly.
server paging and filtering	Asking the database for the page the user is looking at, instead of loading everything and filtering in UI code.
lazy node loading	Building tree children when a branch is expanded, with a count placeholder while it is collapsed, instead of materializing the whole hierarchy at startup.
update payload	The serialized difference the server sends after a handler; large or frequent payloads cost network and browser time that no Visual Studio tool reports.
compression is not free work	Wisej.NET compresses HTTP and WebSocket traffic, which shrinks bytes but does not make an unnecessary update cheap to produce, transmit or render.

3. Lesson narrative

Grids: bind less, then hold less

The grid fix has two independent halves, and doing only one is the usual mistake. The first half is the data shape. Binding entities drags navigation properties, unused columns and large text fields into server memory and into every row the UI builds. Replacing that with a `TicketGridRow` projection — number, customer name, status, age text, and nothing else — shrinks the object graph, the allocations you measured in Module 4, and the update the browser receives.

The second half is how many rows exist at all. In virtual mode the grid keeps a `RowCount` and raises `CellValueNeeded` for the cells it is actually showing, so fifty thousand rows stop being fifty thousand row objects. The trade-off is explicit: that event now runs constantly, so its handler has to be fast and thread-safe, and it must be backed by a paged query service rather than a lookup that hits the database once per cell. Precompute display strings in the projection so the handler only selects a field, and profile the handler itself — virtual mode moves the cost, it does not remove it.

Trees, lists and repeaters

The same principle reshapes the other large controls. A `TreeView` built in full at startup pays for every branch nobody opens; loading children on expansion, with a count placeholder on collapsed branches, makes the cost proportional to what the user actually looks at. Lists and repeaters take the virtual treatment too, and repeaters have an extra trap: a card template with a dozen nested controls multiplies

by the item count, so prefer a summary with a drill-down over a rich control tree repeated a thousand times.

Prefetching is the counterweight to laziness. Virtual scrolling that fetches exactly what is on screen can feel sluggish on a fast scroll, so fetching a little ahead of the viewport is usually worth it. That is a tuning decision to make from measurements, not a default to apply everywhere.

Evidence the profiler cannot give you

Visual Studio tells you what the server did: CPU to build the rows, allocations to hold them, queries to fetch them. It says nothing about what crossed the wire. For that, the browser's network panel is the instrument — the payload size of the update after a grid load, the WebSocket frames after a tree expand, and the time the browser spent rendering them.

Two cautions. Compression makes byte counts look reassuring; an update that should never have been sent is still wasted work on both ends, so compare the *number and size of updates* before and after, not only the compressed total. And while disabling WebSocket compression can help you read frames during debugging, do not benchmark with debug settings that production will not use.

```
dataGridView1.VirtualMode = true;
dataGridView1.RowCount = _tickets.Count(CurrentFilter); // rows, not objects
dataGridView1.CellValueNeeded += (s, e) =>
{
    var row = _tickets.GetProjectedRow(CurrentFilter, e.RowIndex); // paged, cached
    e.Value = e.ColumnIndex switch
    {
        0 => row.Number, 1 => row.Customer, 2 => row.Status, 3 => row.AgeText, _ => null
    };
};
```

Tuning rules worth remembering

Render fewer rows, fewer nodes and fewer controls. Send fewer updates. Bind smaller objects. Defer detail until the user asks for it. Cache immutable lookups; never cache mutable per-user data globally. Prefer server paging and filtering over loading everything and filtering in UI code. And do not refresh an entire grid when one row or one label changed — that single habit accounts for a surprising share of payload in dashboards driven by timers.

Underneath all of them is a design question rather than a technical one: what question does this screen answer for the user? A screen designed around every possible record will always be expensive to make fast. A screen designed around the decision the user has to make is usually cheap by construction, and the lab's before-and-after numbers are the argument you bring to that conversation.



Common pitfalls

Binding full ORM entities with large navigation graphs to every row. Building the entire tree at startup instead of loading nodes on demand. Refreshing whole dashboards on every timer tick. Assuming compression fixes oversized payloads. Writing a `CellValueNeeded` handler that queries the database once per cell.



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 DataGridView, ListView, TreeView and DataRepeater scenarios for CPU, allocations and rendering cost.
- ✓ Apply virtual mode, paging, filtering, projection and prefetch strategies appropriately.
- ✓ Reduce WebSocket/HTTP payload churn by limiting UI mutation and data surface size.
- ✓ Use browser developer tools together with Visual Studio to diagnose network/update size issues.

Lab / Exam Guide

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



Hands-on lab

Shrink the two largest surfaces in WisejPerfLab. The ticket grid binds 50,000 full entities with navigation properties: replace the binding with a `TicketGridRow` projection of just the visible columns, set `VirtualMode` and `RowCount` and serve cells from a `CellValueNeeded` handler backed by a paged query service. The customer tree builds every node at startup: load children on expand instead, using a count placeholder for collapsed branches. Measure each screen before and after with CPU Usage and .NET Object Allocation on the server and the browser network panel for update payload size.

Starting point

WisejPerfLab after Module 4: allocations during the search are down and the detail form no longer leaks, but the grid still binds the full entity list and the customer tree is still built in one pass at startup. The Module 1 budget has a tree-expand target the app misses by a wide margin.

Suggested approach

- ✓ Measure both screens before you touch them: CPU Usage and .NET Object Allocation for the grid load and the tree expand, plus the browser network panel for the update payload each one produces. Record the number and size of updates, not just the compressed total.
- ✓ Define `TicketGridRow` with only the displayed columns and their display strings precomputed, and have the query service return it. Nothing with navigation properties or large text columns should reach the UI.
- ✓ Switch the grid to virtual mode: set `VirtualMode`, set `RowCount` from a count query, and answer `CellValueNeeded` from a paged, cached page of projection rows. Treat that handler as a hot path — no per-cell database call, no formatting, no locking that blocks parallel requests.
- ✓ Rework the customer tree to load children on expand, showing a count placeholder on collapsed branches, and decide from measurement whether a small prefetch ahead of the viewport is worth the extra fetches.
- ✓ Rerun both scenarios on the same dataset and build, and assemble the comparison: server CPU, allocations and retained rows from Visual Studio; update count and payload size from the browser.

Required deliverables

- ✓ `TicketGridRow` projection bound instead of full entities, with only the displayed columns
- ✓ Virtual-mode grid with `RowCount` and a `CellValueNeeded` handler served from a paged query service
- ✓ Customer tree loading child nodes on expand with a count placeholder for collapsed branches
- ✓ Before/after CPU and allocation numbers for the grid load and the tree expand scenarios

- ✓ Browser network evidence comparing update payload size before and after, with the caveat about compression

Student review questions

- ✓ Virtual mode removed the row objects but your CPU trace barely improved. What does that tell you about where the cost actually was, and which measurement would you look at next?
- ✓ Your `CellValueNeeded` handler is correct but slow under fast scrolling. List three things it could be doing that a per-cell handler must never do.
- ✓ The compressed payload for the tree expand barely changed after lazy loading, but users say the screen feels faster. Give a plausible explanation that the byte count alone would hide.



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.