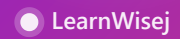


Async Loads, Related Data, Filtering, and Performance

Lesson Guide · Lab / Exam Guide



Lesson Guide

The related-data decision table (projection, Include, filtered Include, explicit loading, no lazy loading in grids), tracking versus no-tracking, paging and indexed filters, async coordination with loading states and background tasks, and measuring before optimising. What this means for a Wisej.NET developer working with real databases — and how to read this module.



Production focus

The related-data decision table (projection, Include, filtered Include, explicit loading, no lazy loading in grids), tracking versus no-tracking, paging and indexed filters, async coordination with loading states and background tasks, and measuring before optimising.

1. The production problem

The Support Desk ticket browser worked in Module 3 with fifty seed tickets. With fifty thousand it will not, unless the query does less work in the right place. The failure is quiet: a grid that takes three seconds because formatting code touched a navigation property per row, a session holding thousands of tracked entities nobody will edit, a search that scans a table. This module is about doing less work, in SQL, on purpose, and proving it with a log rather than a hunch.

2. Key concepts

Concept	Why it matters
projection	A <code>Select</code> that returns exactly the fields the screen needs, such as <code>TicketListItem</code> , so the database joins the related names in one statement.
Include and filtered Include	Eager loading of related entities with the root; a filtered <code>Include</code> narrows a collection (the last five comments) so an editor loads a controlled aggregate.
explicit loading	Loading a navigation later, on demand, through the entry's <code>Collection</code> or <code>Reference</code> and <code>LoadAsync</code> , when the user expands a detail section.
lazy loading and N+1	Navigation properties loaded transparently on first access; in a grid that is one hidden query per row or per navigation access.
AsNoTracking and identity resolution	Results that bypass the change tracker for read-only screens; <code>AsNoTrackingWithIdentityResolution</code> still returns one instance per entity when the same row appears repeatedly.
query-level paging	<code>Where</code> , <code>OrderBy</code> , <code>Skip</code> , <code>Take</code> and <code>Select</code> composed before <code>ToListAsync</code> , with a separate <code>CountAsync</code> for the total.
EF Core logging	The generated SQL of every command with its duration, through <code>LogTo</code> or <code>Microsoft.Extensions.Logging</code> .

3. Lesson narrative

Where the slow grid comes from

Start with the deliberately slow ticket browser the lab measures before fixing. Its `SearchTicketsAsync` loads tracked `Ticket` entities without paging, and `CellFormatting` reads `ticket.Customer.Name`, `ticket.Agent?.DisplayName` and `ticket.Category.Name`. With lazy loading on, every first access to an unloaded navigation runs its own query: fifty rows cost one statement for the tickets plus a hundred and fifty for the names. Nothing throws; only the log shows it.

The rule that removes it: display code must never cause a query. Rows bound to a Wisej.NET `DataGridView` through the `BindingSource` are `TicketListItem` records that already carry the names they show, and `CellFormatting` only formats. The same rule governs tree and list screens: a `TreeView` loads a node's children once, when it is expanded, never by walking navigation properties per node.



Common mistake: lazy loading from row formatting or display code

A navigation property read inside `CellFormatting` or a tree node painter is a query in disguise that runs once per row. Ship the names in the projection instead.

The related-data decision table

EF Core offers four ways to bring related data into a screen; the choice depends on what the screen will do with it.

- Use **projection** when the UI needs a flattened read model: the ticket browser, dashboards, exports.
- Use **Include** when an editor needs a complete aggregate and its size is controlled: a ticket with its customer, category and agent.
- Use **filtered Include** when only a small subset of a collection is required: the last five public comments.
- Use **explicit loading** when the user expands a detail section on demand: the comment history behind a collapsed panel.
- Never rely on **lazy loading** in grids, lists or trees: it silently produces one query per row or per navigation access.

In Support Desk the browser projects to `TicketListItem` ; the editor's `LoadEditorAsync` uses `Include` for the three references and a filtered `Include` for the comments it previews; the comments panel loads the full history with `db.Entry(ticket).Collection(t => t.Comments).LoadAsync()` inside the editor's own context.

Tracking versus no-tracking

Tracking is correct when the same context will save changes to the same entity instances, as the editor does when `SaveChangesAsync` writes the difference. No-tracking is correct for read-only grids, dashboards, lookup lists, exports and autocomplete results. `AsNoTracking` skips the change tracker: less memory per session, faster materialisation, no accidental update from a screen that only reads.

`AsNoTrackingWithIdentityResolution` sits between the two. A no-tracking query that returns entities can hand back a separate `Customer` object for every ticket of the same customer; identity resolution returns one instance per key. Use it when a read-only screen builds an object graph; skip it when you project to flat rows.



Common mistake: using no-tracking entities and later expecting automatic update detection

Editing a `Ticket` from an `AsNoTracking` query, or a `TicketListItem` from the grid, then calling `SaveChangesAsync` changes nothing. The editor loads a fresh tracked entity by key in its own context and maps the approved values onto it.

Paging and indexed filters at the query level

A Wisej.NET grid should not load 100,000 rows just because it can display them. Compose the filters, the sort, `Skip` , `Take` and `Select` at the query level so the database does the work, and return a total count when the UI shows page numbers, from a separate `CountAsync` on the filtered query:

```

var total = await query.CountAsync(token);

var page = await query
    .AsNoTracking()
    .OrderByDescending(t => t.UpdatedAt)
    .Skip(criteria.PageIndex * 50)
    .Take(50)
    .Select(t => new TicketListItem(
        t.Id, t.Number, t.Title, t.Customer.Name,
        t.Agent == null ? null : t.Agent.DisplayName,
        t.Category.Name, t.Status, t.Priority, t.DueDate, t.UpdatedAt))
    .ToListAsync(token);

```

The Module 2 indexes on `Status`, `DueDate`, `CustomerId` and `UpdatedAt` match these filters and this sort. Keep text search index-friendly: a `StartsWith` on the ticket number becomes a prefix `LIKE` an index can serve, while a `Contains` on the title becomes a leading wildcard that scans. When the log shows a frequent filter without an index, add one and measure again.

Async coordination with loading states and background tasks

EF Core provides async methods for every I/O operation. Await each one before starting the next on the same context, because a `DbContext` does not support parallel operations; the two awaits above run in sequence by design. In Wisej.NET an async event handler returns control to the browser while the query runs, so the loading guard matters: show a loading state with the grid's `ShowLoader` or a status `Label`, disable the Search, Next and Previous buttons that would start competing work, and re-enable them in a `finally` block.

Long-running import, export and reporting jobs do not belong in an event handler. Start them with `Application.StartTask`, create the `DbContext` inside the task from the factory rather than capturing the page's context or entities, and push progress only through `Application.Update`, Wisej.NET's context-aware update mechanism. A raw thread that touches a control has no session context, and the cancellation token you pass in is how a job stops when the session ends.

Measure before you optimise

Do not guess. Enable EF Core logging in development with `LogTo` or `Microsoft.Extensions.Logging`, read the SQL and duration of every command, and turn on `EnableSensitiveDataLogging` only on a developer machine, since it prints parameter values. For a slow statement, inspect the database execution plan, where a missing index or a wildcard scan shows up. The lab's before-and-after log is this discipline made concrete.

Only after profiling shows the bottleneck do the specialised tools earn their complexity. Compiled queries help when compilation overhead is measurable on a hot query. Context pooling with `AddPooledDbContextFactory` helps when context construction dominates, and needs care when context state varies per user or tenant, since a pooled context is reset and reused. Split queries help when several collection `Include`s multiply rows. Raw SQL is the last resort. Each answers a number in the log, never a default.



Common mistake: optimising without measuring

Compiled queries, pooling or raw SQL on an unprofiled screen trade readability for an improvement that may not exist. Record, change one thing, record again.



Data-binding habit

Keep the two lifetimes apart. Wisej.NET forms, BindingSources and the selected row live for the session; a DbContext lives for one operation. Create it from the factory, query or save, dispose it, and never let it become session state.



Learning objectives

By the end of this module you'll be able to:

- ✓ Avoid lazy-loading N+1 behavior in grids and tree/list screens.
- ✓ Use Include, filtered Include, explicit loading, and projection intentionally.
- ✓ Use AsNoTracking, AsNoTrackingWithIdentityResolution, paging, indexes, and measured performance improvements.
- ✓ Coordinate EF Core async operations with Wisej.NET loading indicators and background tasks.

Lab / Exam Guide

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



Hands-on lab

Optimise the Support Desk ticket browser: enable EF Core logging in development, record the SQL and timing of the unoptimised search, replace entity loading with projection, add `AsNoTracking` to grid queries, add paging with a page size of 50, replace lazy navigation access with projection or `Include`, and document the measured before-and-after results in your lab notes.

Starting point

Open the Support Desk Data Console solution from Module 5 and the ticket browser page from Module 3. For this lab start from a deliberately naive branch of `TicketQueryService.SearchTicketsAsync`: one that loads tracked `Ticket` entities without paging and lets the grid's `CellFormatting` read the customer, agent and category names through navigation properties, so that there is a real N+1 to measure before you remove it.

Suggested approach

- ✓ Enable EF Core logging in development with `LogTo` or `Microsoft.Extensions.Logging` on `SupportDeskContext`, keep `EnableSensitiveDataLogging` behind a development check, and run the naive search once; count the statements, note the milliseconds and paste the SQL into your lab notes.
- ✓ Rewrite the grid query as a no-tracking projection: `AsNoTracking`, the composed `Where` filters, `OrderByDescending(t => t.UpdatedAt)`, then `Select` into `TicketListItem` with the customer, agent and category names joined by the database, ending in `ToListAsync`.
- ✓ Add paging with a page size of 50: a `CountAsync` on the filtered query for the total, then `Skip` and `Take` before the projection; make sure every filter and the sort use the indexes on `Status`, `DueDate`, `CustomerId` and `UpdatedAt`, and keep the ticket-number search as a prefix match.
- ✓ Remove every navigation access from display code: `CellFormatting` only formats the fields already in the row, and any related data the editor needs is loaded with `Include`, a filtered `Include` or explicit loading inside the editor's own context.
- ✓ Keep the async guard honest: await each database operation before the next on the same context, show the loading state and disable Search, Next Page and Previous Page while a page loads, re-enable in `finally`, then run the optimised search and record the after numbers next to the before numbers.

Required deliverables

- ✓ EF Core logging enabled in development with the unoptimised SQL and timing recorded
- ✓ Grid query rewritten as a no-tracking projection
- ✓ Paging with a page size of 50 and indexed filters
- ✓ No lazy-loading N+1 behaviour left in display code

- ✓ Lab notes documenting the measured before-and-after improvement

Student review questions

- ✓ Your log showed 151 statements for one screen of 50 tickets. Which line of code caused the 150 extra ones, and why did nothing in the application warn you?
- ✓ The editor needs the ticket, its customer and category, and the last five comments; the comments panel opens the full history on demand. Which loading strategy serves each of those three needs, and why is projection the wrong answer for the editor?
- ✓ A teammate proposes compiled queries and context pooling for the ticket browser before looking at the log. What would you measure first, and what would have to be true of the numbers before either change is worth making?



Data-binding habit

Keep the two lifetimes apart. Wisej.NET forms, BindingSources and the selected row live for the session; a DbContext lives for one operation. Create it from the factory, query or save, dispose it, and never let it become session state.