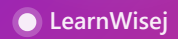


Loading Data into BindingSource, DataGridView, and Lookup Controls

Lesson Guide · Lab / Exam Guide



Lesson Guide

Async LINQ execution, no-tracking projection to list items, BindingSource as the UI bridge, never binding an IQueryable, loading flags around async handlers, and lookup ComboBoxes with DisplayMember and ValueMember. What this means for a Wisej.NET developer working with real databases — and how to read this module.



Production focus

Async LINQ execution, no-tracking projection to list items, BindingSource as the UI bridge, never binding an IQueryable, loading flags around async handlers, and lookup ComboBoxes with DisplayMember and ValueMember.

1. The production problem

Support Desk operators need a ticket grid that searches quickly without pulling the whole database into server memory. In a Wisej.NET application every session keeps its own forms, BindingSources and selected rows on the server, so a screen that loads thousands of tracked **Ticket** entities per user multiplies that cost by the number of open browsers. This module builds the ticket browser properly: compose the query, execute it asynchronously, materialise one small projected page, and hand that list to the UI through a BindingSource.

2. Key concepts

Concept	Why it matters
async execution method	<code>ToListAsync</code> , <code>CountAsync</code> and <code>SingleOrDefaultAsync</code> are the calls that actually send SQL; awaiting them keeps the server thread free while the database works.
projection	A <code>Select</code> into a <code>TicketListItem</code> record returns exactly the columns the grid shows, instead of the Ticket entity with its Customer, Agent and Comments graph.
no-tracking query	<code>AsNoTracking</code> tells the context not to snapshot the results: less memory, no accidental updates, and a list independent of any editing unit of work.
BindingSource	The Wisej.NET component between a data collection and its bound controls; it manages the current item, navigation and list operations and keeps the grid and detail controls synchronised.
IQueryable	A recipe for a database query, not a stable list: it must be executed and materialised before anything is assigned to the UI.
loading guard	A flag, disabled buttons and a status label around every awaited database call, restored in <code>finally</code> , so one operation runs at a time and the page stays usable after an error.
lookup binding	A ComboBox with <code>DisplayMember</code> for the name shown and <code>ValueMember</code> for the key stored, loaded before the first search runs or the edit model is assigned.

3. Lesson narrative

Compose the query, then execute it once

An `IQueryable<Ticket>` describes work the database has not done yet: every `Where` , `OrderBy` , `Skip` , `Take` and `Select` is folded into that description, and nothing runs until an execution method is awaited. The Support Desk `SearchTicketsAsync` starts from `db.Tickets.AsNoTracking()` , appends one `Where` per filter the operator actually set, orders by `UpdatedAt` , pages with `Skip` and `Take` , projects, and only then awaits `ToListAsync` . A separate `CountAsync` on the filtered query, before the page is cut, supplies the total: two SQL statements shaped by the same filters.

The async variants matter because a Wisej.NET event handler runs on a server thread shared with every session: awaiting `ToListAsync` releases that thread while SQL runs, and `SingleOrDefaultAsync` does the same for one ticket loaded by key. The abridged service method:

```

await using var db = await _dbFactory.CreateDbContextAsync(token);
var query = db.Tickets.AsNoTracking();
if (!string.IsNullOrEmpty(criteria.Text))
    query = query.Where(t => t.Number.Contains(criteria.Text) ||
t.Title.Contains(criteria.Text));
if (criteria.CustomerId is not null)
    query = query.Where(t => t.CustomerId == criteria.CustomerId.Value);

var total = await query.CountAsync(token); // total before the page is cut
var page = await query
    .OrderByDescending(t => t.UpdatedAt)
    .Skip(criteria.PageIndex * criteria.PageSize).Take(criteria.PageSize)
    .Select(t => new TicketListItem(t.Id, t.Number, t.Title, t.Customer.Name, /* ... */
t.UpdatedAt))
    .ToListAsync(token); // the only line that runs the page
query
return new PagedResult<TicketListItem>(page, total);

```



Common mistake: calling ToList before filtering or paging

A `ToList()` followed by `Where`, `Skip` or `Take` pulls the whole ticket table over the wire and then filters and pages it in server memory. Compose every clause on the `IQueryable`; make `ToListAsync` the last call.

BindingSource: the bridge between a list and the grid

WiseJ.NET supports two kinds of binding: simple binding connects one control property to one property of the current item (`TextBox.Text` to `Title`), and complex binding connects a list control, a `DataGridView` or a `ComboBox`, to a whole collection. `BindingSource` is the bridge for both: it wraps the collection, tracks the current item, supports navigation through `Position` and `MoveNext`, exposes list operations and raises the change notifications that keep every bound control in step. The grid binds to the `BindingSource`; the `BindingSource`'s `DataSource` is the materialised list.

That list can be a plain `List<TicketListItem>` or DTO collection, a `BindingList<T>` when controls must react to items being added or removed, or a small lookup collection for a `ComboBox`. In the ticket browser, `ticketsDataGridView.DataSource = ticketBindingSource` is set once in the constructor with `AutoGenerateColumns = false`, and each column's `DataPropertyName` names a `TicketListItem` property. After every search only the `BindingSource`'s `DataSource` changes; the grid and its columns stay as designed.



Common mistake: binding IQueryable directly to a grid

Assigning `db.Tickets` or the query itself to a control hands the UI a live recipe tied to a context about to be disposed. Execute with `ToListAsync`, give the list to the `BindingSource`, and bind the grid to the `BindingSource`, never to a `DbSet`.

No-tracking projection for grids, tracked entities for editors

The grid needs seven columns, not the ticket's Description, RowVersion, Comments collection, or Customer and Agent as full objects. Loading them anyway, entity flooding, costs wire bytes, session memory and a change-tracker snapshot per entity, and invites a subtle bug: a grid full of tracked entities looks editable, and a stray `SaveChangesAsync` would write whatever was touched. Projecting with `Select` into `TicketListItem` and marking the query `AsNoTracking` removes all of that; navigation properties remain usable inside the projection (`t.Customer.Name` , `t.Agent == null ? null : t.Agent.DisplayName`) and EF Core translates them into joins.

The editing workflow is the mirror image: the Module 4 editor creates its own context from the factory, loads the tracked entity with `SingleOrDefaultAsync` by key, binds or maps it, and saves through that same context. The list item in the grid is never attached, updated or saved; when the editor closes, the page runs the search again. That keeps the UI list independent of the editing unit of work.



Common mistake: loading entire entity graphs for list screens

A chain of `Include` calls followed by `ToList` materialises tickets, customers, agents and comments the grid never displays, once per session. Project to the list item; reserve tracked, included graphs for the editor that will save them.

Search, paging, lookups and progress feedback

Wisej.NET event handlers can be `async void` , so `searchButton_Click` resets `_pageIndex` and awaits `LoadTicketsAsync` ; the paging buttons do the same with the index adjusted. That method holds the loading guard. If `_loading` is already true it returns at once; otherwise it sets the flag, disables the three buttons, writes "Loading tickets..." to the status label, builds a `TicketSearchCriteria` from the controls, awaits one service call, assigns `result.Items.ToList()` to the BindingSource and reports "Showing 50 of 312 tickets". The `catch` block shows a friendly `AlertBox` ; `finally` re-enables the buttons and clears the flag on every path. EF Core does not support parallel operations on one context; the guard enforces that from the UI.

A genuinely long job, an export or a bulk reassignment, should not hold the request:

`Application.StartTask` runs it in the background and `Application.Update` pushes UI changes to the browser out of band, with the DbContext created inside the task and disposed there, never borrowed from the page.

Status, category, agent and customer are small lookup lists. Load them in the page `Load` handler, before the first search: bind each ComboBox to its list, set `DisplayMember` to the name and `ValueMember` to the key, and read `SelectedValue` when building the criteria so the query filters by `CustomerId` , not by display text. In an editor, bind `SelectedValue` to the edit model's foreign key and populate the lookup before the edit model is assigned to its BindingSource; a SelectedValue with no matching item silently resolves to nothing. When a lookup grows past a few hundred rows, give it search or paging of its own.



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:

- ✓ Use async LINQ execution methods such as ToListAsync and SingleOrDefaultAsync.
- ✓ Bind lists, BindingList<T>, DTOs, and lookup collections through BindingSource.
- ✓ Use no-tracking projection for read-only grids and tracked entities for edit workflows.
- ✓ Implement search, sort, paging, lookup ComboBox binding, and progress feedback.

Lab / Exam Guide

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



Hands-on lab

Build the Support Desk ticket browser: `TicketListItem` and `TicketSearchCriteria` records, a `SearchTicketsAsync` method that composes `AsNoTracking`, `Where`, `OrderBy`, `Skip`, `Take` and `Select` before `ToListAsync`, a `DataGridView` bound through a `BindingSource` to `TicketListItem`, lookup `ComboBoxes` for status and customer loaded before the first search, `Search`, `Next Page` and `Previous Page` buttons, a total count and page size display, and a loading guard that keeps the UI usable after errors.

Starting point

Open the Support Desk Data Console solution as you left it after Module 2: the Web, Data, Services and Tests projects, `SupportDeskContext` registered through `AddDbContextFactory`, the `InitialCreate` migration applied and the seed data in place (five customers, three agents, categories and at least fifty tickets). The Tickets page is still empty apart from a title label, and `TicketQueryService` only contains the count query from Module 1.

Suggested approach

- ✓ In `SupportDesk.Services` add the `TicketListItem`, `TicketSearchCriteria` and `PagedResult<T>` records, then write `SearchTicketsAsync`: a context from the factory, `AsNoTracking`, one `Where` per filter that is set, `CountAsync` for the total, then `OrderByDescending(t => t.UpdatedAt)`, `Skip`, `Take`, `Select` and `ToListAsync`.
- ✓ On `TicketBrowserPage` add a `BindingSource` and a `DataGridView`, set `AutoGenerateColumns` to false, define the columns with `DataPropertyName` for Number, Title, CustomerName, AgentName, Status, DueDate and UpdatedAt, and bind the grid to the `BindingSource` in the constructor.
- ✓ In the page `Load` handler populate the status and customer `ComboBoxes` with `DisplayMember` and `ValueMember` set, including an "All" entry that yields a null criterion, before the first search runs.
- ✓ Write `LoadTicketsAsync` with the loading guard: an early return on `_loading`, buttons disabled, a status label, one awaited service call, the materialised list assigned to the `BindingSource`, an `AlertBox` in `catch`, and the buttons and flag restored in `finally`; wire `Search` (page index reset), `Next Page` and `Previous Page` to it.
- ✓ Show the returned count, the total count and the page size in the status label, disable `Previous Page` on the first page and `Next Page` on the last, then point the connection string at a stopped server to prove the page stays usable after a failure.

Required deliverables

- ✓ `TicketListItem` and `TicketSearchCriteria` records with `SearchTicketsAsync`
- ✓ `DataGridView` bound to `TicketListItem` through a `BindingSource`

- ✓ Filtering and paging composed in the query so the database does the work
- ✓ Lookup ComboBoxes loaded before the first search, showing names and storing keys
- ✓ Search, Next Page and Previous Page buttons with a total count and a loading guard

Student review questions

- ✓ Which lifetime did you choose for the DbContext inside `SearchTicketsAsync`, and what would break if the page kept one context alive for its whole life?
- ✓ After a search, which data is held on the server for this session and which stays only in the database? What does a second concurrent operator cost?
- ✓ If the customer lookup grew to fifty thousand rows, which parts of the page would you change and which parts of the query service would stay exactly as they are?



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.