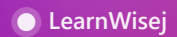


Two-Way Binding and CRUD Editors

Lesson Guide · Lab / Exam Guide



Lesson Guide

DataBindings and BindingSource in editors, EndEdit, CancelEdit, AddNew and RemoveCurrent, direct entity binding versus edit-model binding, and save, cancel, delete and reload flows that end in SaveChangesAsync. What this means for a Wisej.NET developer working with real databases — and how to read this module.



Production focus

DataBindings and BindingSource in editors, EndEdit, CancelEdit, AddNew and RemoveCurrent, direct entity binding versus edit-model binding, and save, cancel, delete and reload flows that end in SaveChangesAsync.

1. The production problem

Support Desk operators want Add Ticket and Edit Ticket screens that behave like desktop forms: a modal dialog, Save, Cancel and Delete buttons, and a grid that shows the change the moment the dialog closes. Two-way binding makes that screen fast to build, but Save is a deliberate operation, not a side effect of typing. If the editor reads a bound value too early, saves a no-tracking grid row as if it were tracked, or lets a double click submit twice, the form still looks correct while it silently loses or duplicates data. This module makes save, cancel, delete and reload explicit, so the editor feels like a desktop form and still follows web-safe data access rules.

2. Key concepts

Concept	Why it matters
two-way binding	A control property such as <code>TextBox.Text</code> is bound to a property of the current item, so the control shows the model and the model receives what the user typed.
<code>DataSourceUpdateMode</code>	The argument to <code>DataBindings.Add</code> that decides when a control pushes its value into the model: on validation (when focus leaves) or on every property change.
EndEdit and CancelEdit	<code>BindingSource.EndEdit</code> commits pending control edits into the current item; <code>CancelEdit</code> discards them. Save always starts with EndEdit.
AddNew and RemoveCurrent	List operations on the BindingSource that create a new current item or remove the selected one from the bound list; neither touches the database until you save.
direct entity binding	Binding controls straight to a tracked EF Core entity, so the change tracker already knows what changed. Productive for small, short-lived, private-context forms.
edit model	A UI-only class such as <code>TicketEditModel</code> that is bound, validated and then mapped to the entity, giving validation, formatting and security trimming a boundary of their own.
fresh-context save and delete	Every Save and Delete creates its own DbContext from the factory, loads the entity by key, changes it and calls <code>SaveChangesAsync</code> , so the database, not the grid row, is the source of truth.

3. Lesson narrative

Wiring the editor: DataBindings and BindingSource

A typical editor opens by assigning an edit model or an entity to a `BindingSource` and binding each control to a property of the current item. In the `TicketEditorForm` the title box binds `Text` to `Title`, the customer `ComboBox` binds `SelectedValue` to `CustomerId`, the due-date picker binds `Value` to `DueDate` and a `CheckBox` binds `Checked` to a boolean flag. Each call to `DataBindings.Add` names the control property, the BindingSource, the member name (use `nameof` so a rename breaks the build, not the screen), enables formatting, and picks a `DataSourceUpdateMode`. `OnValidation` suits text and dates because the value moves when focus leaves the control; `OnPropertyChanged` suits lookups, where the choice should reach the model immediately.

Order matters for lookups. Set `DisplayMember` and `ValueMember` and load the customer, agent and category lists before the edit model becomes the `DataSource`, otherwise the bound foreign key has nothing to select. `LoadEditorAsync` builds the model: an empty `TicketEditModel` with defaults for a new ticket, or a no-tracking read mapped into the model, including the `RowVersion` the concurrency check will need later.



Common mistake

Reading `TextBox.Text` or the model before `EndEdit` while the `BindingSource` has pending edits. The last field the user typed in is not committed yet, so `Save` writes the previous value.

EndEdit, CancelEdit, AddNew and RemoveCurrent

When the user clicks `Save`, the sequence is fixed: `ValidateChildren` so every `Validating` handler runs, `BindingSource.EndEdit` so the last pending control value lands in the current item, domain validation on the model, a copy of the approved values to the EF Core entity if you use an edit model, then `SaveChangesAsync`. Read the model through `BindingSource.Current`, not the controls.

The other `BindingSource` members manage the current item, not the database. `CancelEdit` throws away pending edits on the current item when the item supports editable-object semantics; a plain model is simply reloaded or the form disposed. `AddNew` creates a new item of the list's element type and makes it current, so an `Add Ticket` flow sharing a list with a grid can start from it. `RemoveCurrent` takes the selected item out of the bound list so the grid updates instantly. Neither has spoken to EF Core; a removed row returns on the next search unless a fresh context deletes it.

Save, cancel, delete and reload flows

The `save` method should prevent duplicate clicks, end binding edits, validate, create a `DbContext`, load the entity or create a new one, map fields, call `SaveChangesAsync`, and close or refresh the editor. Guard the method with a saving flag and a disabled `Save` button, restore both in `finally`, and keep the success path short:

```
editBindingSource.EndEdit();
var model = (TicketEditModel)editBindingSource.Current!;
var errors = Validator.Validate(model);
if (errors.Count > 0) { ShowValidation(errors); return; }

await using var db = await DbFactory.CreateDbContextAsync();
var ticket = model.Id == 0
    ? db.Tickets.Add(new Ticket { Number = await NextNumberAsync(db) }).Entity
    : await db.Tickets.SingleAsync(t => t.Id == model.Id);
ticket.Title = model.Title.Trim();
ticket.CustomerId = model.CustomerId!.Value;
ticket.DueDate = model.DueDate;
await db.SaveChangesAsync();
DialogResult = DialogResult.OK; Close();
```

The error path is explicit. Catch `DbUpdateConcurrencyException` first: another operator saved the same ticket and the user needs a conflict dialog, not a generic failure. Catch `DbUpdateException` separately: the database rejected the change (a missing lookup, a duplicate number), so log the detail on the server and show a friendly `AlertBox`. `Cancel` is the simplest path: the context was disposed at the end of `LoadEditorAsync`, so closing with `DialogResult.Cancel` persists nothing. `Reload` belongs to the parent: when `ShowDialog` returns `DialogResult.OK`, the ticket browser runs its search again.

Delete should not rely only on the selected grid row. Confirm the action, create a fresh context, load the entity by key, check business rules such as "tickets with comments cannot be deleted", remove it, and call `SaveChangesAsync`. If the load returns null, someone else already deleted it: show a friendly message and refresh the list. Where auditability matters, prefer a soft delete that keeps the row.



Common mistake

Saving no-tracking list items as if they were tracked entities. A `TicketListItem` from the grid was projected with `AsNoTracking`; changing it and calling `SaveChangesAsync` on a new context writes nothing. Load the entity by key in the save context instead.



Common mistake

Not guarding Save against double clicks. Two overlapping `SaveAsync` calls create two tickets or race the same row; the saving flag plus a disabled button, restored in `finally`, is part of correctness, not polish.

Direct entity binding or an edit model?

Direct entity binding is productive for simple screens because EF Core change tracking already knows the original and current values, so Save is little more than `SaveChangesAsync`. Use it when the form edits one aggregate, the context is private to the form, the form is disposed quickly, and the user can cancel by disposing or reloading. Never bind directly to a no-tracking grid item and expect EF Core to save the changes: nothing is tracking it.

An edit model adds a deliberate boundary between the UI and persistence. It is the better choice for validation, formatting, calculated display values, security trimming and partial updates. The flow is: load the tracked entity, map it to `TicketEditModel`, bind the model, validate it, then copy only the approved values back to the tracked entity. The model can leave `CreatedAt` and `Number` out entirely, so a user cannot edit a field the screen never meant to expose, and a validator that takes a plain model needs no database to test. Choose by risk profile: a private, short-lived, single-aggregate form may bind the entity; anything with rules, roles or partial updates gets a model.



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:

- ✓ Bind editor controls to model properties with DataBindings and BindingSource.
- ✓ Use BindingSource.EndEdit, CancelEdit, AddNew, RemoveCurrent, and current-item management correctly.
- ✓ Implement create, edit, delete, save, cancel, and reload workflows.

- ✓ Choose direct entity binding or edit-model binding based on the risk profile of the form.

Lab / Exam Guide

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



Hands-on lab

Build the Add Ticket and Edit Ticket modal forms for the Support Desk Data Console: a `TicketEditModel`, a `TicketEditorForm` with a `BindingSource` and an `ErrorProvider`, `TextBox`, `ComboBox`, `DateTimePicker` and `CheckBox` properties bound to the model, `LoadEditorAsync` for new and existing tickets, `SaveAsync` that calls `EndEdit`, a validation placeholder, mapping and `SaveChangesAsync`, `Delete` with confirmation and a fresh entity load, and a parent grid that refreshes after `DialogResult.OK`.

Starting point

Open the Support Desk Data Console solution from Module 3. The ticket browser page already searches and pages `TicketListItem` rows through a `BindingSource` into a `DataGridView`, the status and customer lookup `ComboBoxes` load before the first search, and `SupportDeskContext` is registered through `AddDbContextFactory`. Nothing can add, edit or delete a ticket yet.

Suggested approach

- ✓ Create `TicketEditModel` in the Services project first, with only the properties the screen may edit plus `Id` and `RowVersion`, and decide which entity fields stay out of it.
- ✓ Add `TicketEditorForm` with an injected `IDbContextFactory<SupportDeskContext>`, a `BindingSource`, an `ErrorProvider` and Save, Cancel and Delete buttons; load the lookups before the model is assigned.
- ✓ Bind each control with `DataBindings.Add` and an explicit `DataSourceUpdateMode`, then write `LoadEditorAsync` for both the new-ticket and the existing-ticket case.
- ✓ Write `SaveAsync` as the pipeline from the reading (guard, `EndEdit`, validate, fresh context, load or create, map, `SaveChangesAsync`, close) and `DeleteAsync` with a confirmation and a fresh load by key.
- ✓ Wire the ticket browser's Add and Edit buttons to `ShowDialog` and re-run the search only when the result is `DialogResult.OK`; then test every path with a second browser session open.

Required deliverables

- ✓ `TicketEditModel` and `TicketEditorForm` with a `BindingSource` and an `ErrorProvider`
- ✓ `TextBox`, `ComboBox`, `DateTimePicker` and `CheckBox` properties bound to the edit model
- ✓ `LoadEditorAsync` for new and existing tickets and `SaveAsync` with `EndEdit`, mapping and `SaveChangesAsync`
- ✓ Delete with confirmation, a fresh entity load and graceful handling of already-deleted records
- ✓ Parent grid refreshed after `DialogResult.OK`, and Cancel persists nothing

Student review questions

- ✓ What lifetime did you choose for the DbContext in the editor, and what would have to be true about the form before a form-scoped context became acceptable?
- ✓ Which values are bound to the UI, which live only in the edit model, and which remain only in the database, and why is `Number` not editable?
- ✓ Which failure modes did you handle explicitly (double click, already-deleted ticket, database rejection, concurrent edit), and what does the operator see for each?



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.