

# AI Coding Exercise

Build this module's SupportDesk sample with ChatGPT or Claude from a ready-made prompt that points the model at this module's specification, then review, run and extend what it gives you — including against our own reference build on GitHub.

● LearnWisej



## Writing the code is the easy half now.

Getting working code for "Two-Way Binding and CRUD Editors" out of a modern assistant takes one good prompt. Judging it, correcting it and extending it takes everything this module covered. That is the split this exercise is built around — you supply the specification and the review, the assistant supplies the typing.

## What you are asking for

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.

The module asks for these deliverables, and the prompt below asks the assistant for exactly them:

- ✓ 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

## Why it is shaped this way

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.

These are the terms the assistant has to get right. If what comes back uses them loosely, it has not understood the module:

| Term                              | What it means here                                                                                                                                                                                                      |
|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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 <code>EndEdit</code> .                                           |
| AddNew and RemoveCurrent          | List operations on the <code>BindingSource</code> 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 <code>DbContext</code> 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. |

## Build it — pick your assistant

Both buttons open a new tab with the **same ready-made prompt** already filled in. The prompt points the model at this module's specification — a single Markdown file holding the lesson, the lab brief and the deliverables — so it works from the real material instead of guessing what Module 4 is about. You can open [module-04.md](#) yourself to see exactly what it will read.



### Use ChatGPT

Opens [chatgpt.com](#) with this module's prompt loaded. Works with GPT-5 or Codex; any plan that can browse the web will fetch the spec.



### Use Claude

Opens [claude.ai](#) with the same prompt loaded. Claude Code users can paste it into a terminal in an empty folder instead.



### Send it as-is and you get a project, not a snippet

The prompt ends by asking for the whole solution as a single `.zip` with build-and-run steps. Download it, open the solution in Visual Studio, press F5 and SupportDesk comes up on [localhost](#). You need the .NET SDK and the [Wisej-4](#) NuGet package; everything else the prompt asks the assistant to write itself.

The prompt, in full — read it, then edit it if you want a different sample

I am working through Module 4 – "Two-Way Binding and CRUD Editors" – of the LearnWisej Data Binding with EF Core course, and I want to build the module sample with you.

Start by reading the module specification. It is a single Markdown file holding the full lesson, the lab brief and the deliverables:

<https://learnwisej.com/assets/courses/ef-core-binding/specs/module-04.md>

Then build a complete Wisej.NET 4 project I can open in Visual Studio:

1. Project "SupportDesk", C#, net8.0 or later, NuGet package Wisej-4 4.1.x, Kestrel host calling `app.UseWisej()`.
2. Build what the specification's lab asks for: 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`.
3. Produce these deliverables, and tell me where each one lives in the solution:
  - `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
4. Follow the specification wherever it names a Wisej.NET class, property or event – use exactly that one. Where it leaves a choice open, tell me which API you picked and why.
5. Keep the server authoritative: state and validation live in .NET, and the browser renders and reports. Never move a decision into JavaScript because it is easier there.
6. Make the result demonstrable: I should be able to run it and see the behaviour this module is about, including what happens when something goes wrong.

Before you write any code, restate in your own words what you are going to build and where each piece will live, so I can correct the design while it is still cheap.

When the code is done, package the whole solution as a single .zip I can download, and give me the exact steps to build and run it.

Finally, answer these questions from the specification, using your own implementation as the evidence:

- 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?

Ask me about anything that is ambiguous before you start.

## Then review what came back

Code that compiles is not yet code that is right. The interesting questions are design questions: is this the Wisej.NET control the job actually calls for, does it follow the lifecycle the framework gives you, is the authoritative state on the server rather than parked in the browser because that was convenient? Read the result as a reviewer, not as a customer.

This module gives you the checklist. Answer each one against the code you just received:

- ✓ 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?

Anything you had to correct is worth writing down. A short note of the conventions and decisions you settled on, pasted into the first message of your next session, is worth more than any single clever prompt.



### We already built this one — compare yours with ours

The Module 4 reference implementation lives in the public samples repository. Build yours first, then open ours: the differences are the interesting part.



[Open the Module 4 sample on GitHub](#)

Or clone the whole set with `git clone https://github.com/iceteagroup/LearnWisej-Samples` and open `Data Binding with EF Core Course/Module 4`.



### Our sample is a starting point, not a finished product

It was built the same way you are building yours — by prompt, then corrected — and it deliberately stops where the module stops. Expect stand-ins where production code would go, thin error handling and no tests. Treat that as an invitation: use what you learned in this module, plus the assistant, to push it further. Improving code you did not write is the skill this course is actually teaching.

### Three prompts to take it further

Point your assistant at the project you just built — or at ours and try these. Each one is a review-and-extend exercise, not a rewrite.

- ✓ **Swap a piece without moving the seams.** *"Pick the part of this SupportDesk project that is most obviously a stand-in for something real — a fake data source, an in-memory store, a simplified service — and replace it with a production-shaped implementation. The contract either side of it must not change: same types, same calls, same behaviour under failure. Tell me what the real implementation forced you to handle that the stand-in did not."*
- ✓ **Make it survive a bad day.** *"Review this SupportDesk project for the failure modes it does not handle yet — bad or missing input, a dependency that is slow or unavailable, work arriving after the user navigated away, and two actions racing each other. For each one, tell me first whether it is a real risk in Wisej.NET 4 and why, and only then write the fix. Where you decide something needs no handling, say so and explain the reasoning rather than adding defensive code."*
- ✓ **Add the second case.** *"Extend this SupportDesk project with a second case that follows the same pattern as the first — another entity, screen or control of the same shape, wired the same way. Factor out what is genuinely shared into a base class or service, but only where it earns its place; two similar things do not always want an abstraction. Show me the diff and justify each thing you extracted."*



#### Carry it into the lab

The hands-on lab that closes this module asks for the same deliverables you just had the assistant produce. Use what you have here as the shape of your answer and the evidence behind it — not as the answer itself. The lab is graded on your reasoning, and that part has to be yours.