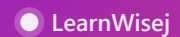


AI Coding Exercise

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.



Same module, built the way you will actually build things.

Every other section of this module explains how "Validation, ErrorProvider, and User Feedback" works. This one asks you to produce it — by directing an assistant with the module's own specification, then verifying the result runs and does what the lab brief describes. It is the workflow you will use on real work, practised on something whose right answer you already know.

What you are asking for

Add validation to the Support Desk ticket editor: DataAnnotations on TicketEditModel, a TicketValidator that returns field-level and summary errors, an ErrorProvider cleared before each run and set for Title, Customer, Category and DueDate, a summary label for cross-field rules, Save disabled while the model is invalid, a DbUpdateException handler that shows a friendly message for a duplicate ticket number, and at least five negative test cases.

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

- ✓ DataAnnotations on TicketEditModel and a TicketValidator returning field-level and summary errors
- ✓ ErrorProvider cleared before each run and set for Title, Customer, Category and DueDate
- ✓ Summary label for cross-field rules and Save disabled while invalid
- ✓ DbUpdateException handled with a friendly message for a duplicate ticket number
- ✓ At least five negative test cases

Why it is shaped this way

The Module 4 ticket editor saves whatever the user typed. An empty title, a ticket with no customer, or a closed ticket due next week all travel through SaveChangesAsync and are stopped, if at all, by the database, which answers with a provider exception that a support agent cannot read. Users need immediate, friendly messages before the database rejects a record, and the application needs a last line of defence that stays in place when the UI is bypassed by an import, a second screen or a bug.

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
three validation layers	UI validation gives immediate feedback, domain validation enforces rules that must hold everywhere, and database constraints guarantee final consistency; each layer catches what the one above it missed.
DataAnnotations on the edit model	Attributes such as [Required] and [StringLength] on TicketEditModel describe user-input rules in one declarative place, separate from the EF Core entity.
explicit validation	Validator.TryValidateObject runs the annotations on demand; EF Core does not run them for you before SaveChangesAsync, so the form must call it.
TicketValidator and ValidationMessage	A service that returns ValidationMessage(FieldName, Message) items, so the rule set is testable and the form only maps messages to controls.
ErrorProvider	The Wisej.NET extender that shows an icon and tooltip beside the invalid control through SetError, and can also be bound to data errors on the current item.
validation summary	A label or MessageBox that lists cross-field errors which belong to no single control, such as a closed ticket with a future due date.
DbUpdateException	The exception EF Core raises when the database rejects the change; it is the final safety layer and must be translated into a friendly message, never shown raw.

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 5 is about. You can open [module-05.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 5 – "Validation, ErrorProvider, and User Feedback" – 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-05.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: Add validation to the Support Desk ticket editor: DataAnnotations on `TicketEditModel`, a `TicketValidator` that returns field-level and summary errors, an `ErrorProvider` cleared before each run and set for Title, Customer, Category and DueDate, a summary label for cross-field rules, Save disabled while the model is invalid, a `DbUpdateException` handler that shows a friendly message for a duplicate ticket number, and at least five negative test cases.
3. Produce these deliverables, and tell me where each one lives in the solution:
 - DataAnnotations on `TicketEditModel` and a `TicketValidator` returning field-level and summary errors
 - `ErrorProvider` cleared before each run and set for Title, Customer, Category and DueDate
 - Summary label for cross-field rules and Save disabled while invalid
 - `DbUpdateException` handled with a friendly message for a duplicate ticket number
 - At least five negative test cases
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:

- If you removed every DataAnnotation from `TicketEditModel` but kept them on the `Ticket` entity, which of your five negative tests would still fail before `SaveChangesAsync`, and why?
- The closed-ticket rule reports DueDate as its field. What would change for the user if it reported no field at all, and when would that be the better choice?
- Two agents create a ticket in the same second and the second save throws `DbUpdateException`. What does the second agent see, what does the server log, and which of the two would you show to a customer?

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:

- ✓ If you removed every `DataAnnotation` from `TicketEditModel` but kept them on the `Ticket` entity, which of your five negative tests would still fail before `SaveChangesAsync`, and why?
- ✓ The closed-ticket rule reports `DueDate` as its field. What would change for the user if it reported no field at all, and when would that be the better choice?
- ✓ Two agents create a ticket in the same second and the second save throws `DbUpdateException`. What does the second agent see, what does the server log, and which of the two would you show to a customer?

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



[Open the Module 5 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 5`.



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.

- ✓ **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."*
- ✓ **Prove it actually works.** *"Go through this project deliverable by deliverable against the module specification at <https://learnwisej.com/assets/courses/ef-core-binding/specs/module-05.md> and show me the evidence that each one is genuinely done — the code path, what to click, and what I should see. Where a deliverable is only partly met, say so plainly instead of arguing it through. Then add whatever is missing."*



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.