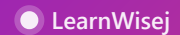


AI Coding Exercise

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.



Brief it, read it, then break it.

This exercise has three parts, and only the first is about prompting. You brief an assistant with the module specification and get a SupportDesk solution back; you read what it wrote and decide whether it really does what the module asked; then you push it further than the module asked. The second and third parts are where the learning is.

What you are asking for

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.

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

- ✓ 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

Why it is shaped this way

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.

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
async execution method	ToListAsync, CountAsync and SingleOrDefaultAsync are the calls that actually send SQL; awaiting them keeps the server thread free while the database works.
projection	A Select into a TicketListItem record returns exactly the columns the grid shows, instead of the Ticket entity with its Customer, Agent and Comments graph.
no-tracking query	AsNoTracking 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 finally, so one operation runs at a time and the page stays usable after an error.
lookup binding	A ComboBox with DisplayMember for the name shown and ValueMember for the key stored, loaded before the first search runs or the edit model is assigned.

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 3 is about. You can open [module-03.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 3 – "Loading Data into BindingSource, DataGridView, and Lookup Controls" – 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-03.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 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.
3. Produce these deliverables, and tell me where each one lives in the solution:
 - 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
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:

- 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?

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:

- ✓ 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?

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



[Open the Module 3 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 3`.



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.

- ✓ **Review it like a pull request.** *"Walk me through this SupportDesk code as if you were reviewing a colleague's pull request and did not write it. Where does state live, what happens end to end on one user action, what would you send back for changes, and which parts would you not be comfortable putting in front of a customer? Be specific and quote the lines you mean — do not summarise the design back to me."*
- ✓ **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."*



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.