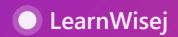


AI Coding Exercise

Build this module's TicketOps 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 TicketOps 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

Refactor the application to use injected services for tickets, users, permissions, notifications, and audit logging. Add a fake service-registration profile and a production registration profile.

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

- ✓ Service registration method
- ✓ Five interfaces and fake implementations
- ✓ Injected MainPage/Form
- ✓ Presenter or workflow service for one screen
- ✓ Service lifetime table

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
Dependency injection (DI)	a class names what it needs by interface and is handed an implementation from outside, rather than creating one itself.
Service container / <code>IServiceProvider</code>	the registry that maps a contract to its implementation and lifetime, and resolves instances on demand.
Service lifetime	how long a resolved instance lives: application/singleton (shared), session-scoped (per user), or transient (per resolve).
Session-scoped service	one instance per user session; the correct home for per-user context (Module 2).
Constructor injection	dependencies supplied as constructor parameters; used for classes you or the container create.
Property injection	a dependency assigned to a marked property after construction; used for framework-created Forms/Pages you don't build yourself.
Presenter / application service	a plain, browser-free class that holds the decision logic a Form delegates to, making that logic unit-testable.
Fake / test double	a stand-in implementation of an interface used in a test in place of the real service.

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 8 is about. You can open [module-08.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 TicketOps 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 8 – "Services, Dependency Injection & Testable UI Patterns" – of the LearnWisej Production Architecture Deep Dive 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/intermediate/specs/module-08.md>

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

1. Project "TicketOps", 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: Refactor the application to use injected services for tickets, users, permissions, notifications, and audit logging. Add a fake service-registration profile and a production registration profile.
3. Produce these deliverables, and tell me where each one lives in the solution:
 - Service registration method
 - Five interfaces and fake implementations
 - Injected MainPage/Form
 - Presenter or workflow service for one screen
 - Service lifetime table
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.

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.

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



[Open the Module 8 sample on GitHub](#)

Or clone the whole set with `git clone https://github.com/iceteagroup/LearnWisej-Samples` and open `Production Architecture Deep Dive/Module 8`.



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 TicketOps 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 TicketOps 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/intermediate/specs/module-08.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.