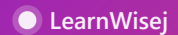


AI Coding Exercise

Build this module's OperationsConsole 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.



The assistant is the fast pair of hands. You are still the architect.

An assistant can produce this module's sample in minutes. What it cannot do is decide whether the result is the right design for your application — that call stays with you. You supply the judgement: brief it properly, then check what comes back against what Module 2 just taught you.

What you are asking for

Build a CustomerEditor UserControl for the Operations Console with labelled fields for name, email, status, start date, credit limit and customer type, using the editor that matches each value instead of a TextBox for everything. Add ErrorProvider messages for required fields, email shape, credit-limit range and invalid dates; Save, Reset and Validate commands with disabled and busy behaviour while saving; and a Toast or MessageBox when validation or save completes.

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

- ✓ CustomerEditor UserControl with labelled fields for name, email, status, start date, credit limit and customer type
- ✓ Matching editors: TextBox, DateTimePicker, NumericUpDown and ComboBox instead of free text everywhere
- ✓ ErrorProvider messages for required fields, email shape, credit-limit range and invalid dates
- ✓ Save, Reset and Validate commands with disabled and busy behaviour while saving
- ✓ Toast or MessageBox feedback when validation or save completes

Why it is shaped this way

Editors are the first place where a control library turns into a business system. A form that looks correct but accepts ambiguous or invalid input creates support tickets and cleanup scripts long after it shipped.

Wisej.NET editors follow the familiar.NET component model but run in the browser, so one business rule can be enforced at several layers: editor properties, validation extenders, event handlers, data binding, service validation and database constraints. The UI layer's job is to prevent obvious mistakes and explain errors clearly; it must never be the only layer that protects the application.

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
value-matched editor	The editor whose type already constrains the input: DateTimePicker for dates, NumericUpDown for bounded numbers, ComboBox for list values, so the server never parses ambiguous text.
command model	A deliberate split between primary commands (Save, Approve, Search, Send, Export), secondary commands (ToolBar, MenuBar, SplitButton, context menu) and destructive commands that need confirmation or careful placement.
ErrorProvider	The extender that attaches a specific, actionable message to the control that is wrong, instead of a generic "Invalid value" elsewhere on the page.
blocking vs non-blocking feedback	MessageBox interrupts for a decision or critical confirmation; AlertBox and Toast report status without stopping the workflow.
named validators	Methods such as ValidateEmail or ValidateCreditLimit that hold one rule each, so the rule is testable and explainable rather than buried in a click handler.
busy state	Disabling Save and showing progress while an operation runs, which prevents duplicate submits and is part of reliability, not decoration.
editor UserControl	A composition of labels, editors, an ErrorProvider and a validate method that returns a result object, so the rest of the app never learns how the email field is checked.

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 2 is about. You can open [module-02.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 OperationsConsole 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 2 – "Editors, Buttons, Validation, and Feedback" – of the LearnWisej Mastering the Control Library 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/mastering-the-control-library/specs/module-02.md>

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

1. Project "OperationsConsole", 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 a CustomerEditor UserControl for the Operations Console with labelled fields for name, email, status, start date, credit limit and customer type, using the editor that matches each value instead of a TextBox for everything. Add ErrorProvider messages for required fields, email shape, credit-limit range and invalid dates; Save, Reset and Validate commands with disabled and busy behaviour while saving; and a Toast or MessageBox when validation or save completes.
3. Produce these deliverables, and tell me where each one lives in the solution:
 - CustomerEditor UserControl with labelled fields for name, email, status, start date, credit limit and customer type
 - Matching editors: TextBox, DateTimePicker, NumericUpDown and ComboBox instead of free text everywhere
 - ErrorProvider messages for required fields, email shape, credit-limit range and invalid dates
 - Save, Reset and Validate commands with disabled and busy behaviour while saving
 - Toast or MessageBox feedback when validation or save completes
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 validation errors can no longer happen at all because of the editor you chose, and which still need an ErrorProvider message?
- If the user clicks Save twice in one second, what does your screen do, and where in the code is that guaranteed?
- What does the shell page know about how the email field is validated, and what would have to change if the rule moved to a service?

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 validation errors can no longer happen at all because of the editor you chose, and which still need an ErrorProvider message?
- ✓ If the user clicks Save twice in one second, what does your screen do, and where in the code is that guaranteed?
- ✓ What does the shell page know about how the email field is validated, and what would have to change if the rule moved to a service?

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



[Open the Module 2 sample on GitHub](#)

Or clone the whole set with `git clone https://github.com/iceteagroup/LearnWisej-Samples` and open `Mastering the Control Library Course/Module 2`.



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.

- ✓ **Add the second case.** *"Extend this OperationsConsole 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/mastering-the-control-library/specs/module-02.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."*
- ✓ **Review it like a pull request.** *"Walk me through this OperationsConsole 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."*



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.