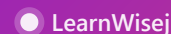


AI Coding Exercise

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



You are not going to type this sample. You are going to direct it.

Almost nobody writes integration code from a blank file any more — an assistant writes the first draft and the developer decides whether it is right. That only works if you know the vocabulary, the contract and the framework rules, which is exactly what the rest of this module taught you. In this exercise you hand that knowledge to ChatGPT or Claude, get the Module 1 sample back as a working Visual Studio solution, and then review it the way you would review a colleague's pull request.

What you'll get out of it

The Module 1 sample is a **Sensor Monitor**: a `TemperatureGauge : Widget` that hosts a third-party gauge library inside a `Wisej.NET` widget container, with `Value` as a validated server property and `ThresholdExceeded` as an ordinary .NET event. Next to the gauge sits a live **Server ⇌ Client trace**, so every message that crosses the wire can be read against the contract you wrote.

Building it by prompt takes fifteen minutes. The value is not the typing you saved — it is the three things you have to be able to do afterwards:

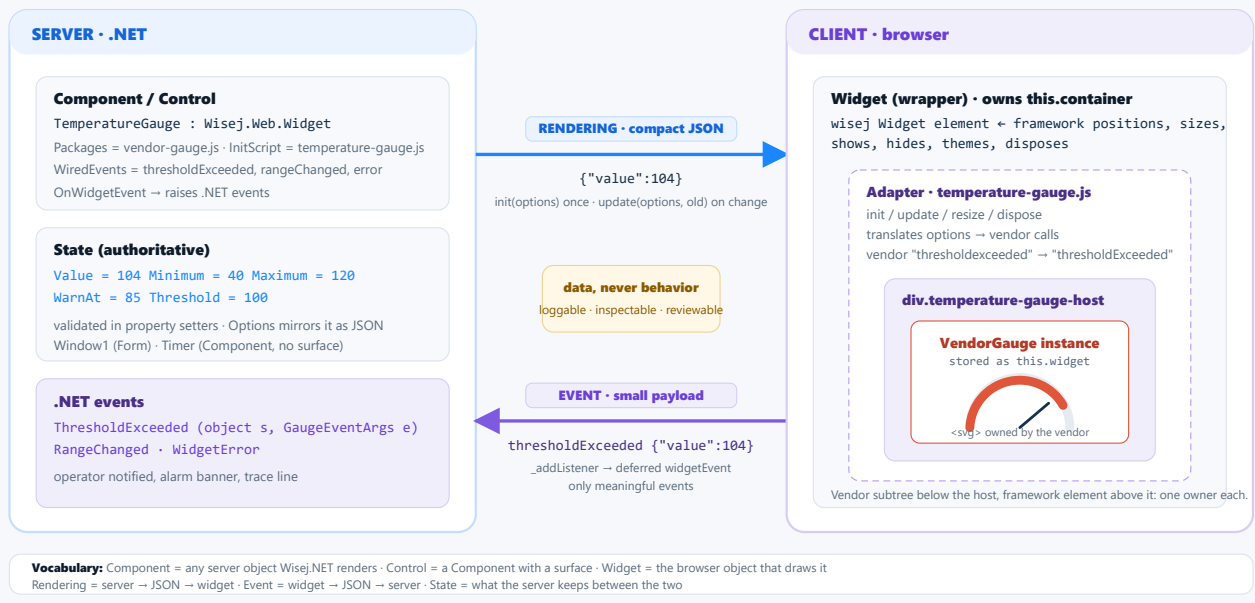
- ✓ **Brief** the assistant with the contract and the framework rules, so the first draft already uses `Wisej.Web.Timer` and hosts the vendor inside a child element of `this.container`.
- ✓ **Verify** what comes back. Code that compiles is not yet code that survives a real page load.
- ✓ **Extend** code you did not write, which is what the rest of your career now looks like.

The architecture you are asking for

Before you prompt, be able to say this in one breath: *the server owns the application state, the browser owns the DOM and the vendor instance, and only data crosses between them* — compact JSON out on rendering, small payloads back on events. The gauge value, its limits and its threshold are server state because they are auditable. The needle animation is not. When the assistant hands you something that keeps the authoritative value in JavaScript, you already know it is wrong.

IntegrationLab · Integration architecture (Module 1)

The server owns the state. The browser owns the DOM and the vendor instance. Only data crosses the wire.



Lesson term	What the assistant has to produce
Component / Control	<code>TemperatureGauge</code> on the server, owning <code>Value</code> , <code>Minimum</code> , <code>Maximum</code> , <code>WarnAt</code> , <code>Threshold</code> . The <code>Wisej.Web.Timer</code> that streams readings is a Component with no surface.
Widget	<code>wwwroot/temperature-gauge.js</code> — the browser adapter that creates the vendor instance in a <i>child</i> element of <code>this.container</code> and keeps it in <code>this.widget</code> .
Package	<code>wwwroot/vendor-gauge.js</code> , the stand-in third-party library, loaded once per page.
Rendering	<code>Options</code> → compact JSON → <code>init(options)</code> on first render, <code>update(options, old)</code> for changed first-level fields.
Event	<code>_addListener</code> / <code>_getEventData</code> → <code>OnWidgetEvent</code> → <code>ThresholdExceeded</code> , <code>RangeChanged</code> , <code>WidgetError</code> .

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 1 is about. You can open `module-01.md` yourself to see exactly what it will read. Read the prompt before you send it; it is the part of this exercise worth learning.



Use ChatGPT

Opens chatgpt.com with the Module 1 prompt loaded. Works with GPT-5 or



Use Claude

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

Codex; any plan that can browse the web will fetch the spec.



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 `IntegrationLab.slnx` in Visual Studio, press F5 and the Sensor Monitor comes up on `localhost`. You need the .NET SDK and the `Wisej-4` NuGet package; nothing else is downloaded, because the prompt makes the assistant write the "vendor" gauge library itself.

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

I am working through Module 1 – "Integration Architecture and Vocabulary" – of the LearnWisej Application Integration 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/app-integration/specs/module-01.md>

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

1. Project "IntegrationLab", C#, net8.0 or later (the reference build uses net10.0), NuGet package Wisej-4 4.1.x, Kestrel host calling `app.UseWisej()`.
2. A `TemperatureGauge : Wisej.Web.Widget` that hosts a third-party gauge library. Ship the "vendor" library yourself as `wwwroot/vendor-gauge.js` and register it as a Package, so nothing has to be downloaded.
3. The server owns the state: `Value`, `Minimum`, `Maximum`, `WarnAt`, `Threshold`. Validate in the property setters and mirror each one into Options as a first-level field so Wisej.NET renders only what changed.
4. A client adapter `wwwroot/temperature-gauge.js`, embedded in the assembly and assigned to `InitScript`. It must create the vendor instance inside a `CHILD` element of this.container, keep it in `this.widget`, and wire events through `_addListener` / `_getEventData` – never call `fireWidgetEvent` synchronously from inside `update()`, and never define a member that shadows a qooxdoo method such as `getWidth`.
5. `WiredEvents = thresholdExceeded, rangeChanged, error`, surfaced on the server as ordinary .NET events with typed `EventArgs` carrying data only.
6. A "Sensor Monitor" `Window1` with the gauge on the left and a live Server-to-Client trace log on the right, plus buttons that exercise five paths: a valid reading, a streamed series driven by a `Wisej.Web.Timer`, a rejected out-of-range value, a deliberately corrupt payload the adapter must catch and report as one error event, and a resync that re-renders the authoritative server state.
7. A `docs/` folder with `ClientServerContract.md`, `WidgetTriage.md`, `IntegrationDecisionRecord.md` and an architecture diagram.

Write `docs/ClientServerContract.md` before you write any code, and show it to me first.

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

The mistakes that cost time here are the ones the compiler cannot catch: code that builds cleanly and then misbehaves on a real page load. These two cost us a debugging session when we built the reference sample, and they are the first things to check in yours:

- ✓ The adapter called `fireWidgetEvent` synchronously from inside a vendor callback. Events raised while a server update is being applied are dropped — the wiring belongs in `_addListener` / `_getEventData` .
- ✓ The adapter defined `this.getWidth` , shadowing a qooxdoo method. The widget rendered zero pixels wide until the helper was renamed.

Then run the five paths the prompt asked for — valid reading, streamed series, rejected value, corrupt payload, resync — and compare each trace line with `docs/ClientServerContract.md` . If a message crosses the wire that the contract does not describe, the contract or the code is wrong, and it is your call which.



We already built this one — compare yours with ours

The Module 1 reference implementation lives in the public samples repository, with the [docs/](#) deliverables, the server class, the adapter and the trace window. Build yours first, then open ours: the differences are the interesting part.



[Open the Module 1 sample on GitHub](#)

Or clone the whole set with `git clone https://github.com/iceteagroup/LearnWisej-Samples` and open `Application Integration Course/Module 1/IntegrationLab.slnx`.



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. The vendor gauge is a stand-in, there are no unit tests, the designer support is minimal and nothing is hardened for production. 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 the sample further

Point your assistant at the cloned folder (or paste the relevant files in) and try these. Each one is a review-and-extend exercise, not a rewrite.

- ✓ **Swap the stand-in for a real library.** *"In this IntegrationLab project, replace `wwwroot/vendor-gauge.js` with a real third-party gauge — for example an ApexCharts radial bar, or a canvas gauge loaded as a Package. Keep `docs/ClientServerContract.md` unchanged: the server contract, the `Options` fields and the three wired events must stay exactly as they are, and only the adapter may change. Show me the diff, and tell me what the real library forced you to handle that the stand-in did not."*
- ✓ **Add a second widget, reusing the pattern.** *"Add a `HumidityGauge : Widget` to this project following the same architecture as `TemperatureGauge` : server-validated state mirrored into `Options` , an adapter that hosts the vendor in a child element, events wired through `_addListener` / `_getEventData` . Put both gauges in the Sensor Monitor window sharing one trace log, and factor out what is genuinely common into a base class — but only if it earns its place. Update the contract document to cover both."*
- ✓ **Make it survive a bad day.** *"Review this project for the failure modes it does not handle yet: the vendor script failing to load, an event arriving after the widget was disposed, two rapid `Value` changes racing the first render, and a browser without `ResizeObserver` . For each one, tell me first whether it is a real risk in Wisej.NET 4 and why — only then write the fix. Add an entry to `docs/IntegrationDecisionRecord.md` for anything you decided to leave alone."*



Carry it into the lab

The lab that closes this module asks for an architecture diagram, a widget triage table and an integration decision record for three candidate widgets. Whatever you build here — yours or ours — already contains one worked example of all three in `docs/` . Use it as the shape of your answer, not as the answer.