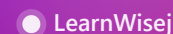


Integration Architecture and Vocabulary

Lesson Guide · Lab / Exam Guide



Lesson Guide

Understand the component/widget/control model and the Wisej.NET rendering and event pipeline. What this means for a Wisej.NET integration developer — and how to read this module.



Production focus

Understand the component/widget/control model and the Wisej.NET rendering and event pipeline.

1. The integration problem

Developers often treat JavaScript widgets as isolated browser objects. In Wisej.NET, the integration must be designed as a server-owned component that renders state, receives events, and delegates browser-specific work to a client widget.

2. Key concepts

Concept	How to use it
Component	Server-side .NET object that owns state and participates in rendering.
Widget	Browser-side JavaScript object that creates and updates the DOM/widget instance.

3. Production lesson

Wisej.NET integration starts with a simple architectural rule: the server owns the application state, while the browser owns the DOM and the third-party widget instance. A Wisej.NET component sends a compact representation of its relevant state to a client-side widget. That client-side widget creates, configures, updates, and wires the third-party JavaScript object. When something important happens on the client, the Wisej.NET widget sends an event payload back to the server component. This model is the same idea behind built-in Wisej.NET controls. The difference is that, for third-party integrations, the client widget usually becomes an adapter around another JavaScript library. The adapter receives state from .NET, translates it to the vendor's options object or method calls, listens to vendor events, and forwards only meaningful information back to the server. A good integration does not try to hide the fact that the third-party control lives in the browser. It keeps the vendor object inside the client adapter, exposes only the state and events the .NET side actually needs, and documents that contract so the server never has to guess what the widget is doing.

The vocabulary, one word at a time

Integration conversations go wrong when two developers use the same word for different things. A Component is any server-side .NET object that Wisej.NET knows how to render to the browser; a timer or a data source is a Component with no visual surface. A Control is a Component that occupies space on a page and has a visual representation. A Widget is the JavaScript object in the browser that draws that surface and reacts to the user. Every Control is a Component, but not every Component is a Control, and neither of them is the Widget.

Server and Client name the two sides of the wire. Rendering is the moment the server serializes a Component's relevant state and sends it to its Widget. An Event is the moment the Widget sends something back. State is whatever the server keeps between those two moments: property values, selection, data, and permissions. In the lab's `TemperatureGauge`, `Value` is server state, the vendor gauge is the Widget, and `ThresholdExceeded` is an Event.

Use these words precisely in decision records and code reviews. If a sentence cannot be written with them, the design is probably unclear.

JSON out, JSON back

The rendering pipeline is deliberately simple. When a Component's state changes, Wisej.NET collects the properties that changed, serializes them as a compact JSON object, and delivers that object to the Widget. The Widget applies each value: for a built-in control that means updating its own DOM, and for an integration it means translating the value into the vendor's options object or a vendor method call. The server never touches the DOM, and the browser never holds the authoritative state.

The return path is the same shape in reverse. When the Widget decides that something matters, it sends a named event with a small JSON payload to the server, and the server raises a .NET event on the Component. In the lab, setting `Value` to 104 on the server renders a small JSON change to the gauge, and when the gauge crosses its threshold one event with the current value comes back and `ThresholdExceeded` fires in C#.

Everything crossing the wire is data, never behavior. That is what keeps the integration reviewable: the JSON in both directions can be logged, inspected in the browser tools, and compared against a written contract.

Why the vendor object lives inside a container

A Wisej.NET Widget owns a DOM element that the framework positions, sizes, shows, hides, themes, and disposes. A third-party library also wants to own an element: it creates children inside it, attaches handlers to it, and sometimes replaces it. If the vendor object were handed the Wisej.NET element itself, the two owners would fight. Layout changes would be lost, disposal would leave handlers behind, and a vendor call that rebuilds the element would break the framework's bookkeeping.

Hosting the vendor object inside a container element that the Widget provides avoids the conflict. The framework keeps its element, the vendor keeps its own subtree below it, and the Widget mediates between them. Resize becomes a call to the vendor's resize method, disposal becomes a call to the vendor's destroy method followed by clearing the container, and theming becomes a style applied to the container.

The exception is a library designed to enhance a single element in place, such as a masked input. Even then, the enhancement targets an element inside the container, never the framework's root.

Choosing the integration depth

Not every widget deserves the same investment. A one-off integration drops a `Wisej.Web.Widget` on a page with a script and options; it is right for a prototype, an internal dashboard, or a widget used exactly once. A reusable integration wraps the same widget in a class with typed properties so that it can be placed on many screens without copying setup. A themeable integration goes further and connects the widget's appearance to the application theme. A native-feeling integration is a full custom control with a server class and a client class, designed, themed, and event-driven like a built-in control.

The lab makes this decision for three candidates. A small gauge used on one dashboard is a one-off. A rich text editor that appears in several forms and must match the theme deserves a reusable, themeable wrapper. A data grid that becomes the application's standard list surface justifies a native-feeling control, because dozens of screens and years of maintenance depend on it.

- How many screens will use it, now and in two years.
- Must it match the theme and work in the Designer.
- Who maintains the vendor library version and its breaking changes.
- What is the cost of getting the depth wrong later.



Production habit

Keep Wisej.NET state ownership and client widget behavior clear. Document the client/server contract before relying on the integration in production.



Common mistake to avoid

Copying demo JavaScript into screens without documenting lifecycle, resources, event contracts, cleanup, or security boundaries.



Learning goals

By the end of this module you'll be able to:

- ✓ Distinguish Component, Control, Widget, Client, Server, Rendering, Event, and State.
- ✓ Explain how server components render JSON state to client widgets and receive JSON event data back.
- ✓ Describe why third-party widgets usually live inside a Wisej.NET widget container rather than replacing the Wisej.NET widget itself.
- ✓ Choose whether a widget should be one-off, reusable, themeable, or native-feeling.

Lab / Exam Guide

What you'll build in the hands-on lab, the recommended workflow, the required deliverables, and the self-check before you submit.



Hands-on lab

Create an integration decision record for three candidate widgets: a small gauge, a rich editor, and a data grid. Decide which integration type each one deserves and explain why.

Recommended workflow

- ✓ Open the existing Wisej.NET integration course project and run it locally.
- ✓ Identify the server component, client widget, resource file, or endpoint involved in this module.
- ✓ Make one small change at a time and test it in the browser before moving on.
- ✓ Keep vendor-specific JavaScript inside the adapter or wrapper instead of scattering it across pages.
- ✓ Update the UI so the learner can clearly see success, failure, progress, or communication changes.
- ✓ Take screenshots or notes that prove the integration works and explain the design decision.
- ✓ If you use an AI coding assistant, brief it with the contract and the Wisej.NET rules below before asking for code; the AI Coding Exercise in this module walks you through it end to end.

Required deliverables

- ✓ Integration architecture diagram
- ✓ Widget triage table
- ✓ First ADR: chosen integration approach for each candidate widget

Self-check before submitting

- ✓ Why is a Control always a Component, but not every Component a Control?
- ✓ Why should most third-party widgets be hosted inside a container element?
- ✓ What state belongs on the server and what state can remain local in the browser?

Working with an AI coding assistant (Claude, OpenAI)

Claude, ChatGPT and the coding assistants built on them (Claude Code, Codex, Copilot) will write this integration quickly, and an unbriefed request gets you a generic answer: a `System.Windows.Forms.Timer`, a widget created straight on `this.container`, an event fired with `fireWidgetEvent` from inside `update()`, or a `return` statement inside `EvalAsync`. All of these compile or parse, and all of them fail at runtime. Give the assistant the contract and the framework rules up front and they disappear. The assistant is a fast pair of hands; you remain the architect who owns the contract and verifies every claim.

Brief it the way you would brief a new team member

- ✓ **Vocabulary first.** State that a Component is a server .NET object that owns state, a Control is a Component with a surface, and a Widget is the browser adapter. "Widget" without this definition invites client-only code.
- ✓ **Framework rules in the first message.** Paste them verbatim: the server class derives from `Wisej.Web.Widget` ; vendor scripts are `Packages` ; the adapter is the `InitScript` loaded with `GetResourceString` from an embedded resource; state is written to `((dynamic)this.Options)` and arrives as `init(options)` / `update(options, old)` ; client events are declared in `WiredEvents` , wired on the client through `_addListener` / `_getEventData` , and land in `OnWidgetEvent` .
- ✓ **Contract before code.** Give the assistant the state fields, the event names and payloads, and the failure paths. Ask it to restate the contract in its own words before it writes a line; a wrong restatement is cheaper to fix than wrong code.
- ✓ **Name the versions.** "Wisej.NET 4, NuGet Wisej-4 4.1.0, net10.0, Kestrel host" removes an entire class of Wisej 2 / .NET Framework answers.

Ask for small pieces and review each one

- ✓ Request the server class alone, review it, then request the client adapter alone. One file per turn keeps the review honest and the errors local.
- ✓ Say what must *not* change: "keep the contract; do not add events; do not validate in JavaScript". Assistants happily widen a design while fixing a bug.
- ✓ Paste compiler output and browser-console errors **verbatim**. Paraphrased errors lead to guessed fixes.
- ✓ When an API looks unfamiliar, ask: "Which Wisej.NET class and member is this? Where is it documented?" Check the answer against the Wisej.NET documentation or the framework's XML docs before accepting it.
- ✓ Ask for the failure paths explicitly: "What happens when the vendor throws inside `update()` ? Show the code path that turns it into one `error` event." Unasked, any first draft stays on the happy path.

Verify, then keep what you learned

- ✓ Nothing is done until it builds and the five paths run in the browser: success, progress, an invalid value, a corrupt payload, and recovery. The trace panel is your evidence.
- ✓ Keep a short note of the conventions and corrections you settled on, and paste it into the first message of your next session. That note is worth more than any single prompt.
- ✓ Keep vendor-specific JavaScript inside the adapter even when the assistant offers to inline it into the page; the same rule applies to human and AI-written code.
- ✓ Never paste licence keys, connection strings or customer data into a prompt. Use placeholders and a stand-in vendor library.



Wisej.NET rules to state in the brief

Events fired synchronously from inside `update()` are dropped — wire them through `_addListener`. Defining `this.getWidth`, `this.getHeight` or `this.destroy` on the wrapper shadows a qooxdoo method and can collapse the widget to 0 px. `EvalAsync` takes an expression, not a `return` statement. Anonymous objects assigned to `Options` are camel-cased on the wire; WebMethod return values are not. Use `Wisej.Web.Timer`, never `System.Windows.Forms.Timer`.

The **AI Coding Exercise** section of this module puts all of this to work: it hands you a ready-made prompt that points an assistant at this guide, and our own Sensor Monitor — built exactly this way — needed two corrections after running, both listed above.