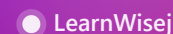


Events, Handlers, and Widget Event Contracts

Lesson Guide · Lab / Exam Guide



Lesson Guide

Turn vendor callbacks into stable .NET events without leaking vendor complexity into the application. What this means for a Wisej.NET integration developer — and how to read this module.



Production focus

Turn vendor callbacks into stable .NET events without leaking vendor complexity into the application.

1. The integration problem

Third-party widgets produce many browser events, but most application screens only need a few meaningful server-side events with clean payloads.

2. Key concepts

Concept	How to use it
fireWidgetEvent	Forward important widget events to the server.
Payload contract	Send small meaningful event data to .NET code.

3. Production lesson

Event integration is not a mechanical forwarding exercise. A grid may produce dozens of browser events. A chart may generate hover, click, selection, zoom, render, export, and layout callbacks. Sending all of them to the server creates noise and unnecessary traffic. A good wrapper chooses the events that represent business decisions or server-owned state changes. For `Wisej.Web.Widget`, `fireWidgetEvent(name, data)` is the direct path to the server-side `WidgetEvent`. For custom controls, `fireEvent` and `fireDataEvent` can be handled through `OnWebEvent`, provided the server config wires the event. The difference matters because custom controls may share event names with built-in events and must pass unhandled events to the base implementation. The event payload should be deliberately shaped. Instead of sending a complete vendor event object, send `rowKey`, `fieldName`, `value`, `selection`, or another compact object that the server can act on directly. The server should never have to dig through a vendor-specific structure to find the one value it cares about, and the payload should stay small enough that raising it on every meaningful interaction does not become a performance problem.

Wiring vendor events at the right moment

A vendor callback can only be attached once the vendor instance exists, so the natural place is inside `init`, immediately after creation, while `this` still refers to the Wisej.NET widget. Some libraries expose

an event API on the instance, others accept callbacks in the options object, and a few require attaching to DOM elements they create. Each approach works, provided the handler captures the widget reference as Module 2 taught.

Libraries that recreate their instance on certain option changes need the handlers re-attached after each recreation, which is why the lab keeps the wiring in one function called from both `init` and the recreate path in `update`. Attaching in `init` only would leave the second instance silent.

Every attach should have a detach in the widget's disposal path. A vendor handler that outlives its widget holds the widget in memory, and after enough screen changes the leak becomes visible as a slow browser.

fireWidgetEvent to the server

For a `Wisej.Web.Widget`, the client method `fireWidgetEvent(name, data)` is the only path an integration needs. The name is a short string chosen by the integration, such as `thresholdCrossed`, and the data is any serializable object. On the server, the widget's `WidgetEvent` fires with the name and the deserialized data, and the handler switches on the name to decide what to do.

```
me.gauge.on('change', function (v) {
    if (v > options.threshold && !me.above) {
        me.above = true;
        me.fireWidgetEvent('thresholdCrossed', { value: v });
    }
});
```

The example shows the important discipline: the client decides when an event is meaningful and fires once, rather than forwarding every change and letting the server filter. A wrapper class derived from `Wisej.Web.Widget` can handle `WidgetEvent` internally and re-raise a typed .NET event such as `ThresholdExceeded`, so application code never sees the string names.

fireEvent and fireDataEvent for custom controls

A custom control built on the client class model raises events differently. `fireEvent(name)` raises an event with no payload and `fireDataEvent(name, data)` raises one with a payload. Because these are ordinary client-side events, the server only receives them when the control's server class has declared an interest, which the lab describes as wiring the event in the server configuration.

On the server, the events arrive in `OnWebEvent`, which receives the event name and data. The override handles the names it owns and must pass everything else to the base implementation, because a custom control shares the event pipeline with the built-in events its base class already handles, such as focus and resize. Swallowing an unknown event breaks behavior that has nothing to do with the integration.

The difference between the two mechanisms is the module's self-check. `WidgetEvent` is a single catch-all event on a Widget; `OnWebEvent` is the general event entry point of a control, shared with the framework, and therefore needs the base call.

Payloads that .NET code can trust

An event payload is a contract, and the lab records it in a table: event name, each field with its type, when the event fires, and which .NET event it becomes. The chart's `pointClicked` carries a series name, an x

value, and a y value. The knob's `valueChanged` carries the new value only. Nothing in either payload is a vendor object, a DOM node, or a whole data row.

Small payloads are stable across vendor upgrades, because they are built from a few values the wrapper extracts, not from the vendor's event argument shape. They are also safe: the server treats every field as untrusted, validates it, and looks up anything sensitive itself. A row key in a payload is a lookup key, never proof that the user may edit that row.

- Does the payload contain only primitives and small objects.
- Could the same payload be produced by a different vendor library.
- Is every field validated on the server before it is used.



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:

- ✓ Wire vendor events during initialization or after instance creation.
- ✓ Use `fireWidgetEvent` for `Wisej.Web.Widget` event forwarding.
- ✓ Use `fireEvent` and `fireDataEvent` with wired events for custom controls.
- ✓ Design event payloads that are small, stable, and meaningful to .NET application code.

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

Add server-side events for gauge threshold crossing, knob value changed, and chart point clicked. Normalize all event payloads into simple DTOs.

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.

Required deliverables

- ✓ Three client event handlers
- ✓ Three server-side event handlers
- ✓ Payload contract table

Self-check before submitting

- ✓ Why not forward every vendor event?
- ✓ What is the difference between `WidgetEvent` and `OnWebEvent`?
- ✓ What makes an event payload stable?