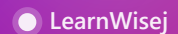


Rapid Integration with Wisej.Web.Widget

Lesson Guide · Lab / Exam Guide



Lesson Guide

Use Wisej.Web.Widget for one-off integrations, prototypes, and quick validation of vendor libraries. What this means for a Wisej.NET integration developer — and how to read this module.



Production focus

Use Wisej.Web.Widget for one-off integrations, prototypes, and quick validation of vendor libraries.

1. The integration problem

A team needs to prove that a JavaScript widget can be hosted in Wisej.NET before investing in a reusable control wrapper.

2. Key concepts

Concept	How to use it
Packages	Register JavaScript and CSS assets required by a one-off widget.
InitScript	Create the vendor widget using options rendered from .NET.

3. Production lesson

`Wisej.Web.Widget` is the fastest integration surface. You drop the widget on a page, attach scripts and styles through Packages, and write an InitScript that creates or updates the third-party object. This approach is ideal when the widget is used once, when the API is still being explored, or when a prototype must be validated before building a full wrapper. The InitScript must create the vendor object from the correct element and store the instance somewhere predictable, usually on `this`. The update function is optional, but a production-quality prototype should include it whenever the .NET side can change options after the first render. Some vendor widgets expose a clean `setOptions` or `option` method. Others require destruction and recreation. The integration code should make that behavior explicit. The Options object is convenient and flexible, but it is also easy to misuse. In early prototypes, JSON-like options are acceptable because they get a widget on screen quickly. Once the wrapper is reused, move the important settings into named .NET properties so they can be validated, versioned, and understood without reading the vendor documentation.

Packages for a single widget

A `Wisej.Web.Widget` declares its dependencies through its Packages collection. Each package names a source, which can be a URL, an application path, or an embedded resource, and Wisej.NET loads it once

per browser session, in list order, before the InitScript runs. Both scripts and stylesheets are declared this way, so the vendor CSS travels with the vendor JavaScript and is not forgotten on the next screen.

For the lab's dashboard, the gauge needs one script and one stylesheet, while the jQuery-style knob needs jQuery, the knob plugin, and its stylesheet. Two widgets on the same page can list the same package, and the framework loads it once. Ordering across widgets is not guaranteed, so a widget must list everything it needs itself rather than relying on another widget having loaded it first.

Keep vendor files under the application's own path, or embed them, rather than pointing at a public CDN. A CDN outage or version change should not be able to break a production dashboard.

init creates, update re-syncs

The InitScript defines two functions on the widget. `init(options)` runs once, after the packages have loaded and the container exists, and creates the vendor object from the serialized Options.

`update(options, old)` runs every time the server changes Options after that first render and receives both the new values and the previous ones. Without an update function, a changed option would require destroying and recreating the whole widget.

```
init: function (options) {
    this.gauge = new VendorGauge(this.container, options);
},
update: function (options, old) {
    if (options.value !== old.value)
        this.gauge.setValue(options.value);
    if (options.max !== old.max)
        this.gauge.setOptions({ max: options.max });
}
```

The comparison against `old` is what makes update cheap. Only the changed values are pushed to the vendor object, an animation is not restarted for an unrelated option, and a vendor that needs recreation for one specific option can do that only when that option changed. Write the update function even for a prototype, because the server button in the lab that changes options is exactly the case it handles.

Serializing options from .NET

The Options property is a dynamic object on the server. Any value assigned to it, including nested objects, arrays, and anonymous types, is serialized to JSON by Wisej.NET and arrives in JavaScript as the `options` argument. A gauge configuration with a value, a range, a list of colored bands, and a label object can be built in C# as ordinary objects and read in the InitScript as ordinary JavaScript.

Two rules keep this predictable. Property names are camel-cased during serialization, so a C# `MaxValue` arrives as `maxValue`, and the InitScript must use the JavaScript spelling. Only what the vendor needs should be assigned; a database entity or a domain object placed in Options is serialized in full, which leaks data to the browser and bloats every render. Build a small object for the widget and assign that.

Nested objects are compared as whole values during update, so replacing the bands array triggers a re-sync of the bands even if only one color changed. Structure the options so that frequently changing values, such as the current reading, sit at the top level.

Calling the client with Call

Some vendor behavior is a method rather than an option: reset an animation, refresh a layout, or scroll to a point. The server reaches those through `Call`, which names a function on the client widget and passes arguments that are serialized like Options. The `InitScript` defines that function next to `init` and `update`, and it forwards to the vendor instance stored on the widget.

`Call` does not execute immediately. Wisej.NET queues it and sends it to the browser with the next response, together with any property changes made in the same request. This is why a button handler can set three options and call two methods, and the browser receives one consistent batch. It also means the server cannot read a result from `Call`; the value-returning variants are the subject of Module 6.

Order the work so the state is right before the method runs. Set the options first and issue the call second, because the batch is applied in order and the vendor method will see the updated options.



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:

- ✓ Configure JavaScript and CSS packages for a Widget instance.
- ✓ Write `init(options)` and optional `update(options, old)` functions.
- ✓ Pass complex option objects from .NET to JavaScript using Wisej.NET serialization.
- ✓ Invoke client methods from the server with `Call` and understand when the call is queued.

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

Build a dashboard page with two one-off widgets: a simple gauge and a jQuery-style knob. Both must update when server-side properties change.

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

- ✓ Widget packages configured
- ✓ InitScript and update implementation
- ✓ Server button that changes widget options and calls the client

Self-check before submitting

- ✓ When is `update` called?
- ✓ What makes Options flexible but error-prone?
- ✓ Why might a widget need to be destroyed and recreated?