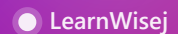


JavaScript Integration with Widget

Native Controls vs the Widget · Clean C# & Widget Security



Native Controls vs the Widget

When to use native Wisej.NET controls vs a custom JavaScript Widget, the JavaScript file that creates the widget (a complete init/update example), and how to connect it from C# through Packages, InitScript and Options.



Lesson goal

Decide when a screen should use native Wisej.NET controls and when a custom JavaScript **Widget** is the better fit — then write the JavaScript file that is the widget (its `init` and `update` functions), and connect that file to C# through Packages, InitScript and Options.

1. Use native Wisej.NET controls first

Most business screens should start with built-in Wisej.NET controls — labels, text boxes, buttons, lists, grids, panels, tabs, dialogs and menus. They're easier to maintain and debug, and already fit the Wisej.NET event model. A **Widget** is for the cases where you need a browser-side visual feature that normal controls don't provide neatly.

Need	Recommended choice	Reason
Ticket form, settings screen, normal buttons	Native Wisej.NET controls	Fast to build and easy for junior developers to understand.
Ticket list or business table	DataGridView or native list/grid controls	Works directly with C# data binding and selection events.
Custom chart, gauge, map, timeline or special visualization	Widget	JavaScript libraries are often better for specialized visuals.
One-off visual flourish with no business value	Avoid it	Don't add maintenance risk just to make the UI flashy.

2. How a Widget fits into a Wisej.NET application

Keep normal business logic in C#. Put *only* the visual JavaScript integration inside the Widget wrapper. The bridge between them should be narrow and intentional.

```
Wisej.NET Page      → C# controls and layout
Widget Wrapper     → loads JS/CSS and exposes options
JavaScript UI      → chart / status gauge / timeline
Server Events      → refresh, status and data updates
```

The Wisej.NET page and services still own the business data; the Widget owns a small browser-side visual element. Send only the data the widget needs, listen only for events that matter, and keep the rest of the app in normal C#.

3. Loading JavaScript and CSS

A Widget-style integration usually needs script files, style files and configuration data. The script may be a local file or a trusted third-party library; the CSS controls appearance; the configuration tells the widget what to draw.

Part	What it does	Beginner rule
JavaScript file	Creates or updates the custom visual element.	Keep it small, named and focused on the widget.
CSS file	Styles the widget element.	Avoid styles that accidentally change the whole app.
Configuration / options	Passes values from C# into the widget.	Send simple data: numbers, labels and status strings.
Events / callbacks	Let the widget tell the server something happened.	Only wire events that support a real workflow.

On the C# side these map to three members of the `Widget` control: **Packages** (the script and stylesheet files to load, in order), **InitScript** (the JavaScript that creates the visual) and **Options** (the data object serialized to the browser). The rest of this reading builds each one for a small *status gauge* — the same widget you'll build in the lab.

4. Writing the JavaScript file — the widget itself

The JavaScript file is where the visual actually gets created. Wisej.NET runs your `InitScript` once in the browser with `this` set to the client-side widget object, and it expects you to define **two functions** on it:

Function	When Wisej.NET calls it	What you do in it
<code>this.init(options)</code>	Once, after the Packages have loaded and the widget's DOM element exists.	Create the visual inside <code>this.container</code> and draw the first values from <code>options</code> .
<code>this.update(options, old)</code>	Every time C# changes <code>Options</code> and calls <code>Update()</code> .	Redraw only what changed. Without it, changed options would need the widget recreated.

Three names to remember inside that file: `this.container` is the element Wisej.NET reserves for your content (create your markup *inside* it, never replace the widget's own root); `options` is the plain JavaScript object that C# assigned to `Options`, with property names camel-cased (`Percent` becomes

`percent`); and `this.fireWidgetEvent(name, data)` sends a small event back to the server, which arrives as the C# `WidgetEvent` .

Widgets/statusGauge.js — a complete example file (no third-party library needed)

```
// Wisej.NET Widget InitScript for a simple status gauge.
// Runs in the browser. "this" is the client-side widget object.
// Options from C# arrive as: { percent: 0-100, label: "text", status: "ok|warn|error" }

// Called ONCE, after Packages have loaded and the widget element exists.
this.init = function (options) {
    // Keep a reference to the widget: inside DOM callbacks "this" changes.
    var me = this;

    // Build the visual inside the container Wisej.NET reserves for us.
    me.container.innerHTML =
        '<div class="lw-gauge">' +
        '  <div class="lw-gauge-track"><div class="lw-gauge-fill"></div></div>' +
        '  <div class="lw-gauge-label"></div>' +
        '</div>';

    // Cache the elements we will update later.
    me.fillEl = me.container.querySelector('.lw-gauge-fill');
    me.labelEl = me.container.querySelector('.lw-gauge-label');

    // Wire ONE event that supports a real workflow: click → tell the server.
    me.container.addEventListener('click', function () {
        // Small payload only. Arrives in C# as WidgetEvent with e.Type == "gaugeClick".
        me.fireWidgetEvent('gaugeClick', { percent: me.lastPercent || 0 });
    });

    // First draw with the values C# sent.
    me.render(options);
};

// Called EVERY TIME C# changes Options and calls widStatus.Update().
this.update = function (options, old) {
    // Redraw only if something we show actually changed.
    if (!old || options.percent !== old.percent ||
        options.label !== old.label || options.status !== old.status) {
        this.render(options);
    }
};

// Optional helper: a named function the server can call with widStatus.Call("pulse").
this.pulse = function () {
    var el = this.container.querySelector('.lw-gauge');
    if (!el) return;
    el.classList.remove('lw-gauge-pulse');
    void el.offsetWidth; // restart the CSS animation
    el.classList.add('lw-gauge-pulse');
};

// Our own drawing routine: pure display logic, no business rules.
```

```

this.render = function (options) {
    options = options || {};
    var percent = Math.max(0, Math.min(100, Number(options.percent) || 0));
    this.lastPercent = percent;

    this.fillEl.style.width = percent + '%';
    this.labelEl.textContent = (options.label || 'Status') + ' - ' + percent + '%';

    // Status only picks a colour class; C# decided WHICH status applies.
    this.fillEl.className = 'lw-gauge-fill lw-gauge-' + (options.status || 'ok');
};

```

Widgets/statusGauge.css — scoped to the widget's own class names

```

.lw-gauge      { font: 13px/1.4 inherit; padding: 8px; }
.lw-gauge-track { height: 14px; border-radius: 7px; background: #e6e6ee; overflow: hidden; }
.lw-gauge-fill  { height: 100%; width: 0; border-radius: 7px; transition: width .4s ease; }
.lw-gauge-ok    { background: #1f9d57; }
.lw-gauge-warn  { background: #e8a317; }
.lw-gauge-error { background: #d64545; }
.lw-gauge-label { margin-top: 6px; color: #444; }
.lw-gauge-pulse { animation: lwPulse .6s ease; }
@keyframes lwPulse { 50% { opacity: .5; } }

```

🔗 Why var me = this?

Inside `init`, `this` is the Wisej.NET widget. Inside a browser callback such as the click listener, `this` is the element that raised the event, so `this.fireWidgetEvent` would be undefined. Capture the widget in `me` first and use `me` inside every callback. Also notice that the file never decides whether a value is "warn" or "error" — C# decides the status and the JavaScript only picks a colour for it.

Prefix your CSS class names (here `lw-gauge-...`) so the stylesheet can't accidentally restyle the rest of the application. Keep the file free of business rules, secrets and server URLs — it should be readable by any developer as "draw these numbers".

5. Connecting the file to C#

Add a `Widget` control to the page (name it `widStatus`), then wire the two files and the first options in the page's constructor or `Load` handler. The CSS goes in **Packages**; the JavaScript file becomes the **InitScript**; the data goes in **Options**.

```

// C# - StatusPage.cs (both files live under the web project's /Widgets folder)
private void StatusPage_Load(object sender, EventArgs e)
{
    // 1. Stylesheet (and any third-party scripts) load in list order, once per session.
    widStatus.Packages.Add(new Package {
        Name     = "statusGauge-css",
        Source   = "Widgets/statusGauge.css"
    });
};

```

```

// 2. The JavaScript file IS the widget: load it as the InitScript.
//     (You can also paste the file into the InitScript property in the designer.)
widStatus.InitScript = File.ReadAllText(Application.MapPath("Widgets/statusGauge.js"));

// 3. First options – safe display values only, prepared by a C# service.
var data = statusService.GetStatus();
widStatus.Options = new { percent = data.Percent, label = data.Label, status =
data.Status };

// 4. Listen only for the event we wired in the JS file.
widStatus.WidgetEvent += widStatus_WidgetEvent;
}

private void btnRefresh_Click(object sender, EventArgs e)
{
    var data = statusService.GetStatus(); // C# owns the data and the rules
    widStatus.Options = new { percent = data.Percent, label = data.Label, status =
data.Status };
    widStatus.Update(); // runs this.update(options, old) in the
browser
    widStatus.Call("pulse"); // runs this.pulse() in the browser
    AddLog("Widget refreshed with server data");
}

private void widStatus_WidgetEvent(object sender, WidgetEventArgs e)
{
    if (e.Type == "gaugeClick")
    {
        // e.Data is the small object the JS sent: { percent: 42 }
        int percent = e.Data.percent;
        AddLog($"Gauge clicked at {percent}%");
    }
}

```

Note how the two sides line up: `Options = new { percent, label, status }` in C# is exactly the `options` object the JavaScript reads; `Update()` triggers `this.update`; `Call("pulse")` runs `this.pulse`; and `fireWidgetEvent('gaugeClick', ...)` arrives as `WidgetEvent` with `e.Type == "gaugeClick"`. Set the options *before* calling `Update()` or `Call()` — Wisej.NET sends them to the browser as one batch, applied in order.

6. What happens at runtime

1. The page renders and the browser creates the widget's element.
2. Wisej.NET loads every entry in **Packages**, in order (here the CSS).
3. It runs the **InitScript**, which defines `init`, `update`, `pulse` and `render` on the widget.
4. It calls `init(options)` with the serialized **Options** — the gauge appears.
5. Each time C# sets `Options` and calls `Update()`, the browser runs `update(options, old)`.
6. A click in the browser fires `gaugeClick`; C# handles it in `WidgetEvent` like any button click.



If the widget stays blank

Open the browser's developer tools. An error at load time naming an undefined symbol means a Packages order problem; "fireWidgetEvent is not a function" inside a callback means a lost `this` (use `me`); an empty container with no error usually means `init` never got the option it needed — log `options` at the top of `init` to see what C# actually sent.



You're ready for the next reading

You can choose native controls vs a Widget, write the widget's JavaScript file with `init` and `update`, and connect it from C# through Packages, InitScript, Options, `Update()`, `Call()` and `WidgetEvent`. Next: keeping the rest of the app clean C#, and the security risks of third-party scripts.

Clean C# & Widget Security

Keep business rules in C# (not JavaScript), wrap integration code, and recognize the security risks of third-party scripts.



Lesson goal

Keep business rules in C# (not in JavaScript), wrap integration code so it doesn't scatter, and recognize the security and maintenance risks of adding third-party scripts to a business app.

1. Keep the rest of the app clean C#

The Widget shouldn't become a hidden second application inside your Wisej.NET app. Don't put business rules, ticket calculations, permissions or database logic in JavaScript — keep those in C# services and pass only the final display data to the widget.

Bad pattern	Better pattern
Place business rules inside a large JavaScript block.	Calculate business values in a C# service, then send display values to the Widget.
Load random scripts from unknown sites.	Use trusted libraries, version them, and document why they're needed.
Let every page manually build its own script strings.	Wrap integration code in one reusable widget helper / control.
Expose secrets or internal server data to JavaScript.	Send only safe display data to the browser.

2. Security and maintenance risks

Third-party scripts add power, but also risk — they can change page behavior, break when the library updates, conflict with styles, or expose data if the wrong information is sent to the browser. Treat every external script as a dependency that must be reviewed and maintained.

- Use trusted libraries; avoid copy-pasting unknown scripts into the project.
- Prefer local, versioned files when you need predictable behavior.
- **Never** send API keys, passwords, connection strings or private internal data to the browser widget.
- Document what the script does and which screen uses it.
- Test the screen after theme changes, layout changes and browser updates.



Never expose secrets to the browser

Anything sent to the widget reaches the browser. Prepare safe display data in C# and send only what the widget needs to draw.

3. The beginner pattern to remember

Native controls are the default. The Widget is for special visuals. C# owns the data and rules; JavaScript owns the visual rendering. The integration boundary should be small, named and easy to test.



You're ready for the lab

Next: watch the widget walkthrough, pass the knowledge check, then build a small JavaScript status/chart widget — fed safe display data from a C# service, refreshed through a server event, with a data card to verify the values.