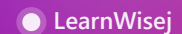


Integración de JavaScript con Widget

Controles nativos frente a Widget · C# limpio y seguridad de los widgets



Controles nativos frente a Widget

Cuándo usar controles nativos de Wisej.NET frente a un Widget JavaScript personalizado, el archivo JavaScript que crea el widget (un ejemplo init/update completo) y cómo conectarlo desde C# mediante Packages, InitScript y Options.



Objetivo de la lección

Decide cuándo una pantalla debe usar controles nativos de Wisej.NET y cuándo un **Widget** JavaScript personalizado encaja mejor; después escribe el archivo JavaScript que es el widget (sus funciones `init` y `update`) y conecta ese archivo con C# mediante Packages, InitScript y Options.

1. Usa primero los controles nativos de Wisej.NET

La mayoría de las pantallas de negocio deben empezar con los controles integrados de Wisej.NET: etiquetas, cuadros de texto, botones, listas, cuadrículas, paneles, pestañas, diálogos y menús. Son más fáciles de mantener y depurar, y ya encajan con el modelo de eventos de Wisej.NET. Un **Widget** es para los casos en los que necesitas una funcionalidad visual del lado del navegador que los controles normales no ofrecen de forma limpia.

Necesidad	Opción recomendada	Motivo
Formulario de ticket, pantalla de ajustes, botones normales	Controles nativos de Wisej.NET	Rápidos de construir y fáciles de entender para desarrolladores junior.
Lista de tickets o tabla de negocio	DataGridView o controles nativos de lista/cuadrícula	Funcionan directamente con el enlace de datos de C# y los eventos de selección.
Gráfico, indicador, mapa, línea de tiempo o visualización especial personalizados	Widget	Las bibliotecas JavaScript suelen ser mejores para elementos visuales especializados.
Un adorno visual puntual sin valor de negocio	Evítalo	No añadas riesgo de mantenimiento solo para que la interfaz sea llamativa.

2. Cómo encaja un Widget en una aplicación Wisej.NET

Mantén la lógica de negocio normal en C#. Pon *solo* la integración visual JavaScript dentro del contenedor Widget. El puente entre ambos debe ser estrecho e intencionado.

Página Wisej.NET	→ controles C# y diseño
Contenedor Widget	→ carga JS/CSS y expone las opciones
Interfaz JavaScript	→ gráfico / indicador de estado / línea de tiempo
Eventos del servidor	→ actualización, estado y cambios de datos

La página Wisej.NET y los servicios siguen siendo dueños de los datos de negocio; el Widget se encarga de un pequeño elemento visual del lado del navegador. Envía solo los datos que necesita el widget, escucha solo los eventos que importan y mantén el resto de la aplicación en C# normal.

3. Cargar JavaScript y CSS

Una integración de tipo Widget suele necesitar archivos de script, archivos de estilo y datos de configuración. El script puede ser un archivo local o una biblioteca de terceros de confianza; el CSS controla la apariencia; la configuración le dice al widget qué dibujar.

Parte	Qué hace	Regla para principiantes
Archivo JavaScript	Crea o actualiza el elemento visual personalizado.	Mantenlo pequeño, con nombre y centrado en el widget.
Archivo CSS	Da estilo al elemento del widget.	Evita estilos que cambien por accidente toda la aplicación.
Configuración / opciones	Pasa valores de C# al widget.	Envía datos simples: números, etiquetas y cadenas de estado.
Eventos / callbacks	Permiten que el widget avise al servidor de que ha pasado algo.	Conecta solo los eventos que respaldan un flujo real.

En el lado de C# estas partes corresponden a tres miembros del control `Widget : Packages` (los archivos de script y hojas de estilo que se cargan, en orden), `InitScript` (el JavaScript que crea el elemento visual) y `Options` (el objeto de datos que se serializa al navegador). El resto de esta lectura construye cada uno de ellos para un pequeño *indicador de estado*, el mismo widget que construirás en el laboratorio.

4. Escribir el archivo JavaScript: el propio widget

El archivo JavaScript es donde realmente se crea el elemento visual. Wisej.NET ejecuta tu `InitScript` una vez en el navegador con `this` apuntando al objeto widget del lado del cliente, y espera que definas **dos funciones** en él:

Función	Cuándo la llama Wisej.NET	Qué haces en ella
<code>this.init(options)</code>	Una vez, después de que se hayan cargado los Packages y exista el elemento DOM del widget.	Crea el elemento visual dentro de <code>this.container</code> y dibuja los primeros valores a partir de <code>options</code> .
<code>this.update(options, old)</code>	Cada vez que C# cambia <code>Options</code> y llama a <code>Update()</code> .	Vuelve a dibujar solo lo que ha cambiado. Sin ella, cambiar las opciones obligaría a recrear el widget.

Tres nombres que debes recordar dentro de ese archivo: `this.container` es el elemento que Wisej.NET reserva para tu contenido (crea tu marcado *dentro* de él y nunca sustituyas la raíz propia del widget); `options` es el objeto JavaScript simple que C# asignó a `Options`, con los nombres de propiedad en camelCase (`Percent` pasa a ser `percent`); y `this.fireWidgetEvent(name, data)` envía un pequeño evento al servidor, que llega como el `WidgetEvent` de C#.

Widgets/statusGauge.js: un archivo de ejemplo completo (sin bibliotecas de terceros)

```
// 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: acotado a los nombres de clase propios del widget

```

.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; } }

```

🔗 ¿Por qué var me = this?

Dentro de `init`, `this` es el widget de Wisej.NET. Dentro de un callback del navegador, como el listener del clic, `this` es el elemento que lanzó el evento, así que `this.fireWidgetEvent` no estaría definido. Guarda primero el widget en `me` y usa `me` dentro de cada callback. Fíjate también en que el archivo nunca decide si un valor es «warn» o «error»: C# decide el estado y el JavaScript solo elige un color para él.

Añade un prefijo a los nombres de clase CSS (aquí `lw-gauge-...`) para que la hoja de estilo no pueda cambiar por accidente el estilo del resto de la aplicación. Mantén el archivo libre de reglas de negocio, secretos y URL del servidor: cualquier desarrollador debería poder leerlo como «dibuja estos números».

5. Conectar el archivo con C#

Añade un control `Widget` a la página (llámalo `widStatus`) y después conecta los dos archivos y las primeras opciones en el constructor de la página o en el manejador `Load`. El CSS va en **Packages**; el archivo JavaScript se convierte en el **InitScript**; los datos van en **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}%");
    }
}
```

Fíjate en cómo encajan los dos lados: `Options = new { percent, label, status }` en C# es exactamente el objeto `options` que lee el JavaScript; `Update()` lanza `this.update`; `Call("pulse")` ejecuta `this.pulse`; y `fireWidgetEvent('gaugeClick', ...)` llega como `WidgetEvent` con `e.Type ==`

"`gaugeClick`". Establece las opciones *antes* de llamar a `Update()` o `Call()`: Wisej.NET las envía al navegador en un único lote, aplicado en orden.

6. Qué ocurre en tiempo de ejecución

1. La página se renderiza y el navegador crea el elemento del widget.
2. Wisej.NET carga cada entrada de **Packages**, en orden (aquí, el CSS).
3. Ejecuta el **InitScript**, que define `init`, `update`, `pulse` y `render` en el widget.
4. Llama a `init(options)` con las **Options** serializadas: aparece el indicador.
5. Cada vez que C# establece `Options` y llama a `Update()`, el navegador ejecuta `update(options, old)`.
6. Un clic en el navegador lanza `gaugeClick`; C# lo gestiona en `WidgetEvent` como cualquier clic de botón.



Si el widget se queda en blanco

Abre las herramientas de desarrollo del navegador. Un error durante la carga que nombra un símbolo no definido indica un problema de orden en Packages; «fireWidgetEvent is not a function» dentro de un callback indica que se ha perdido `this` (usa `me`); un contenedor vacío sin errores suele indicar que `init` no recibió la opción que necesitaba: registra `options` al principio de `init` para ver lo que C# ha enviado realmente.



Ya estás listo para la siguiente lectura

Sabes elegir entre controles nativos y un Widget, escribir el archivo JavaScript del widget con `init` y `update`, y conectarlo desde C# mediante Packages, InitScript, Options, `Update()`, `Call()` y `WidgetEvent`. A continuación: mantener el resto de la aplicación en C# limpio y los riesgos de seguridad de los scripts de terceros.

C# limpio y seguridad de los widgets

Mantén las reglas de negocio en C# (no en JavaScript), encapsula el código de integración y reconoce los riesgos de seguridad de los scripts de terceros.



Objetivo de la lección

Mantén las reglas de negocio en C# (no en JavaScript), encapsula el código de integración para que no se disperse y reconoce los riesgos de seguridad y mantenimiento de añadir scripts de terceros a una aplicación de negocio.

1. Mantén el resto de la aplicación en C# limpio

El Widget no debe convertirse en una segunda aplicación oculta dentro de tu aplicación Wisej.NET. No pongas reglas de negocio, cálculos de tickets, permisos ni lógica de base de datos en JavaScript: mantenlos en servicios C# y pasa al widget solo los datos de visualización finales.

Patrón incorrecto	Patrón mejor
Poner las reglas de negocio dentro de un gran bloque JavaScript.	Calcular los valores de negocio en un servicio C# y después enviar los valores de visualización al Widget.
Cargar scripts cualesquiera desde sitios desconocidos.	Usar bibliotecas de confianza, fijar sus versiones y documentar por qué se necesitan.
Dejar que cada página construya a mano sus propias cadenas de script.	Encapsular el código de integración en un único helper o control de widget reutilizable.
Exponer secretos o datos internos del servidor a JavaScript.	Enviar al navegador solo datos de visualización seguros.

2. Riesgos de seguridad y mantenimiento

Los scripts de terceros aportan potencia, pero también riesgo: pueden cambiar el comportamiento de la página, romperse cuando se actualiza la biblioteca, entrar en conflicto con los estilos o exponer datos si se envía al navegador la información equivocada. Trata cada script externo como una dependencia que debe revisarse y mantenerse.

- Usa bibliotecas de confianza; evita copiar y pegar scripts desconocidos en el proyecto.
- Prefiere archivos locales y con versión cuando necesites un comportamiento predecible.
- **Nunca** envíes claves de API, contraseñas, cadenas de conexión ni datos internos privados al widget del navegador.
- Documenta qué hace el script y qué pantalla lo usa.
- Prueba la pantalla después de cambios de tema, cambios de diseño y actualizaciones del navegador.



Nunca expongas secretos al navegador

Todo lo que se envía al widget llega al navegador. Prepara datos de visualización seguros en C# y envía solo lo que el widget necesita para dibujarse.

3. El patrón para principiantes que debes recordar

Los controles nativos son la opción predeterminada. El Widget es para elementos visuales especiales. C# es dueño de los datos y las reglas; JavaScript se encarga del renderizado visual. El límite de la integración debe ser pequeño, tener un nombre claro y ser fácil de probar.



Ya estás listo para el laboratorio

A continuación: mira el recorrido guiado del widget, supera la evaluación de conocimientos y después construye un pequeño widget JavaScript de estado o gráfico, alimentado con datos de visualización seguros desde un servicio C#, actualizado mediante un evento de servidor y con una tarjeta de datos para verificar los valores.