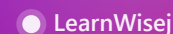


Integrazione JavaScript con Widget

Controlli nativi e Widget a confronto · C# pulito e sicurezza dei Widget



Controlli nativi e Widget a confronto

Quando usare i controlli nativi di Wisej.NET e quando un Widget JavaScript personalizzato, il file JavaScript che crea il widget (un esempio completo di init/update) e come collegarlo da C# tramite Packages, InitScript e Options.



Obiettivo della lezione

Decidi quando una schermata deve usare i controlli nativi di Wisej.NET e quando un **Widget** JavaScript personalizzato è la scelta migliore — poi scrivi il file JavaScript che è il widget (le sue funzioni `init` e `update`) e collega quel file a C# tramite Packages, InitScript e Options.

1. Usa prima i controlli nativi di Wisej.NET

La maggior parte delle schermate di business dovrebbe partire dai controlli predefiniti di Wisej.NET — label, textbox, pulsanti, elenchi, griglie, pannelli, schede, finestre di dialogo e menu. Sono più facili da mantenere e da debuggare, e si inseriscono già nel modello a eventi di Wisej.NET. Un **Widget** serve per i casi in cui ti occorre una funzionalità visuale lato browser che i controlli normali non offrono in modo pulito.

Esigenza	Scelta consigliata	Motivo
Form dei ticket, schermata delle impostazioni, pulsanti normali	Controlli nativi di Wisej.NET	Rapidi da costruire e facili da capire per gli sviluppatori junior.
Elenco dei ticket o tabella di business	DataGridView o controlli nativi di elenco/griglia	Funziona direttamente con il data binding C# e con gli eventi di selezione.
Grafico, indicatore, mappa, timeline o visualizzazione particolare	Widget	Le librerie JavaScript sono spesso migliori per le visualizzazioni specializzate.
Un effetto visivo isolato senza valore per il business	Evitalo	Non aggiungere rischi di manutenzione solo per rendere l'interfaccia vistosa.

2. Come si inserisce un Widget in un'applicazione Wisej.NET

Tieni la normale logica di business in C#. Metti dentro il wrapper del Widget *solo* l'integrazione JavaScript della parte visuale. Il collegamento tra i due deve essere stretto e intenzionale.

```
Pagina Wisej.NET → controlli C# e layout
Wrapper del Widget → carica JS/CSS ed espone le opzioni
Interfaccia JS → grafico / indicatore di stato / timeline
Eventi server → aggiornamenti di stato e di dati
```

La pagina Wisej.NET e i servizi governano ancora i dati di business; il Widget governa un piccolo elemento visuale lato browser. Invia solo i dati di cui il widget ha bisogno, ascolta solo gli eventi che contano e lascia il resto dell'app in normale C#.

3. Caricare JavaScript e CSS

Un'integrazione in stile Widget di solito ha bisogno di file di script, file di stile e dati di configurazione. Lo script può essere un file locale o una libreria di terze parti affidabile; il CSS controlla l'aspetto; la configurazione dice al widget che cosa disegnare.

Parte	Che cosa fa	Regola per chi inizia
File JavaScript	Crea o aggiorna l'elemento visuale personalizzato.	Tienilo piccolo, con un nome chiaro e concentrato sul widget.
File CSS	Dà stile all'elemento del widget.	Evita stili che cambiano per errore l'intera app.
Configurazione / opzioni	Passa valori da C# al widget.	Invia dati semplici: numeri, etichette e stringhe di stato.
Eventi / callback	Permettono al widget di comunicare al server che è successo qualcosa.	Collega solo gli eventi che servono a un flusso reale.

Lato C# queste parti corrispondono a tre membri del controllo **Widget : Packages** (i file di script e di stile da caricare, in ordine), **InitScript** (il JavaScript che crea la parte visuale) e **Options** (l'oggetto dati serializzato verso il browser). Il resto di questa lettura costruisce ciascuno di essi per un piccolo *indicatore di stato* — lo stesso widget che costruirai nel lab.

4. Scrivere il file JavaScript — il widget vero e proprio

Il file JavaScript è il posto in cui la parte visuale viene effettivamente creata. Wisej.NET esegue il tuo **InitScript** una volta nel browser con **this** impostato sull'oggetto widget lato client, e si aspetta che tu definisca su di esso **due funzioni**:

Funzione	Quando la chiama Wisej.NET	Che cosa ci fai
<code>this.init(options)</code>	Una volta, dopo che i Packages sono stati caricati e l'elemento DOM del widget esiste.	Crea la parte visuale dentro <code>this.container</code> e disegna i primi valori da <code>options</code> .
<code>this.update(options, old)</code>	Ogni volta che C# cambia <code>Options</code> e chiama <code>Update()</code> .	Ridisegna solo ciò che è cambiato. Senza questa funzione, per applicare le opzioni cambiate bisognerebbe ricreare il widget.

Tre nomi da ricordare dentro quel file: `this.container` è l'elemento che Wisej.NET riserva al tuo contenuto (crea il tuo markup *dentro* di esso, non sostituire mai la radice del widget); `options` è il semplice oggetto JavaScript che C# ha assegnato a `Options`, con i nomi delle proprietà in camelCase (`Percent` diventa `percent`); e `this.fireWidgetEvent(name, data)` invia un piccolo evento al server, che arriva come `WidgetEvent` in C#.

Widgets/statusGauge.js — un file di esempio completo (non serve alcuna libreria di terze parti)

```
// InitScript del Widget Wisej.NET per un semplice indicatore di stato.
// Viene eseguito nel browser. "this" è l'oggetto widget lato client.
// Le opzioni da C# arrivano così: { percent: 0-100, label: "text", status: "ok|warn|error"
}

// Chiamata UNA VOLTA, dopo che i Packages sono stati caricati e l'elemento del widget
esiste.
this.init = function (options) {
    // Tieni un riferimento al widget: dentro le callback del DOM "this" cambia.
    var me = this;

    // Costruisci la parte visuale dentro il container che Wisej.NET ci riserva.
    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>';

    // Memorizza gli elementi che aggiorneremo in seguito.
    me.fillEl = me.container.querySelector('.lw-gauge-fill');
    me.labelEl = me.container.querySelector('.lw-gauge-label');

    // Collega UN SOLO evento che serve a un flusso reale: click → avvisa il server.
    me.container.addEventListener('click', function () {
        // Solo un payload piccolo. Arriva in C# come WidgetEvent con e.Type ==
        "gaugeClick".
        me.fireWidgetEvent('gaugeClick', { percent: me.lastPercent || 0 });
    });

    // Primo disegno con i valori inviati da C#.
    me.render(options);
};

// Chiamata OGNI VOLTA che C# cambia Options e chiama widStatus.Update().
```

```

this.update = function (options, old) {
    // Ridisegna solo se qualcosa che mostriamo è davvero cambiato.
    if (!old || options.percent !== old.percent ||
        options.label !== old.label || options.status !== old.status) {
        this.render(options);
    }
};

// Helper facoltativo: una funzione con nome che il server può chiamare con
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; // riavvia l'animazione CSS
    el.classList.add('lw-gauge-pulse');
};

// La nostra routine di disegno: pura logica di visualizzazione, nessuna regola di business.
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 + '%';

    // Lo stato sceglie solo una classe di colore; QUALE stato si applica l'ha deciso C#.
    this.fillEl.className = 'lw-gauge-fill lw-gauge-' + (options.status || 'ok');
};

```

Widgets/statusGauge.css — limitato ai nomi di classe 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; } }

```



Perché var me = this?

Dentro `init`, `this` è il widget Wisej.NET. Dentro una callback del browser, come il listener del click, `this` è l'elemento che ha generato l'evento, quindi `this.fireWidgetEvent` sarebbe undefined. Cattura prima il widget in `me` e usa `me` dentro ogni callback. Nota anche che il file non decide mai se un valore è "warn" o "error" — lo stato lo decide C#, e il JavaScript sceglie solo il colore corrispondente.

Aggiungi un prefisso ai nomi delle classi CSS (qui `Lw-gauge-...`) così che il foglio di stile non possa ridefinire per errore lo stile del resto dell'applicazione. Tieni il file libero da regole di business, segreti e URL del server — qualsiasi sviluppatore deve poterlo leggere come "disegna questi numeri".

5. Collegare il file a C#

Aggiungi un controllo `Widget` alla pagina (chiamalo `widStatus`), poi collega i due file e le prime opzioni nel costruttore della pagina o nel gestore `Load`. Il CSS va in **Packages**; il file JavaScript diventa l'**InitScript**; i dati vanno in **Options**.

```
// C# - StatusPage.cs (entrambi i file stanno nella cartella /Widgets del progetto web)
private void StatusPage_Load(object sender, EventArgs e)
{
    // 1. Il foglio di stile (e gli eventuali script di terze parti) si caricano nell'ordine
    // dell'elenco, una volta per sessione.
    widStatus.Packages.Add(new Package {
        Name    = "statusGauge-css",
        Source  = "Widgets/statusGauge.css"
    });

    // 2. Il file JavaScript È il widget: caricalo come InitScript.
    // (Puoi anche incollare il file nella proprietà InitScript nel designer.)
    widStatus.InitScript = File.ReadAllText(Application.MapPath("Widgets/statusGauge.js"));

    // 3. Prime opzioni - solo valori di visualizzazione sicuri, preparati da un servizio
    C#.
    var data = statusService.GetStatus();
    widStatus.Options = new { percent = data.Percent, label = data.Label, status =
    data.Status };

    // 4. Ascolta solo l'evento che abbiamo collegato nel file JS.
    widStatus.WidgetEvent += widStatus_WidgetEvent;
}

private void btnRefresh_Click(object sender, EventArgs e)
{
    var data = statusService.GetStatus(); // C# governa i dati e le regole
    widStatus.Options = new { percent = data.Percent, label = data.Label, status =
    data.Status };
    widStatus.Update(); // esegue this.update(options, old) nel
    browser
    widStatus.Call("pulse"); // esegue this.pulse() nel browser
    AddLog("Widget refreshed with server data");
}

private void widStatus_WidgetEvent(object sender, WidgetEventArgs e)
{
    if (e.Type == "gaugeClick")
    {
        // e.Data è il piccolo oggetto inviato dal JS: { percent: 42 }
        int percent = e.Data.percent;
        AddLog($"Gauge clicked at {percent}%");
    }
}
```

```
}  
}
```

Nota come i due lati corrispondono: `Options = new { percent, label, status }` in C# è esattamente l'oggetto `options` che legge il JavaScript; `Update()` attiva `this.update`; `Call("pulse")` esegue `this.pulse`; e `fireWidgetEvent('gaugeClick', ...)` arriva come `WidgetEvent` con `e.Type == "gaugeClick"`. Imposta le opzioni *prima* di chiamare `Update()` o `Call()` — Wisej.NET le invia al browser in un unico blocco, applicato in ordine.

6. Che cosa succede in fase di esecuzione

1. La pagina viene disegnata e il browser crea l'elemento del widget.
2. Wisej.NET carica ogni voce di **Packages**, in ordine (qui il CSS).
3. Esegue l'**InitScript**, che definisce `init`, `update`, `pulse` e `render` sul widget.
4. Chiama `init(options)` con le **Options** serializzate — l'indicatore compare.
5. Ogni volta che C# imposta `Options` e chiama `Update()`, il browser esegue `update(options, old)`.
6. Un click nel browser genera `gaugeClick`; C# lo gestisce in `WidgetEvent` come qualsiasi click su un pulsante.



Se il widget resta vuoto

Apri gli strumenti per sviluppatori del browser. Un errore al caricamento che nomina un simbolo non definito indica un problema di ordine nei Packages; "fireWidgetEvent is not a function" dentro una callback indica un `this` perso (usa `me`); un container vuoto senza errori di solito significa che `init` non ha mai ricevuto l'opzione di cui aveva bisogno — scrivi `options` nella console all'inizio di `init` per vedere che cosa ha inviato davvero C#.



Sei pronto per la prossima lettura

Sai scegliere tra controlli nativi e un Widget, scrivere il file JavaScript del widget con `init` e `update` e collegarlo da C# tramite Packages, InitScript, Options, `Update()`, `Call()` e `WidgetEvent`. Prossimo passo: mantenere il resto dell'app in C# pulito, e i rischi di sicurezza degli script di terze parti.

C# pulito e sicurezza dei Widget

Tieni le regole di business in C# (non in JavaScript), incapsula il codice di integrazione e riconosci i rischi di sicurezza degli script di terze parti.



Obiettivo della lezione

Tieni le regole di business in C# (non in JavaScript), incapsula il codice di integrazione così che non si sparpagli, e riconosci i rischi di sicurezza e di manutenzione legati all'aggiunta di script di terze parti a un'app di business.

1. Mantieni il resto dell'app in C# pulito

Il Widget non deve diventare una seconda applicazione nascosta dentro la tua app Wisej.NET. Non mettere in JavaScript regole di business, calcoli sui ticket, permessi o logica del database — tienili nei servizi C# e passa al widget solo i dati finali da visualizzare.

Pattern sbagliato	Pattern migliore
Mettere le regole di business dentro un grande blocco JavaScript.	Calcolare i valori di business in un servizio C#, poi inviare al Widget i valori da visualizzare.
Caricare script a caso da siti sconosciuti.	Usare librerie affidabili, fissarne la versione e documentare perché servono.
Lasciare che ogni pagina costruisca a mano le proprie stringhe di script.	Incapsulare il codice di integrazione in un unico helper / controllo widget riutilizzabile.
Esporre segreti o dati interni del server a JavaScript.	Inviare al browser solo dati di visualizzazione sicuri.

2. Rischi di sicurezza e di manutenzione

Gli script di terze parti aggiungono potenza, ma anche rischi — possono cambiare il comportamento della pagina, rompersi quando la libreria si aggiorna, entrare in conflitto con gli stili o esporre dati se al browser vengono inviate le informazioni sbagliate. Tratta ogni script esterno come una dipendenza da rivedere e mantenere.

- Usa librerie affidabili; evita di copiare e incollare nel progetto script sconosciuti.
- Preferisci file locali con una versione fissata quando ti serve un comportamento prevedibile.
- **Non inviare mai** al widget nel browser chiavi API, password, stringhe di connessione o dati interni privati.
- Documenta che cosa fa lo script e quale schermata lo usa.
- Prova la schermata dopo cambi di tema, cambi di layout e aggiornamenti del browser.



Non esporre mai segreti al browser

Tutto ciò che viene inviato al widget arriva al browser. Prepara in C# dati di visualizzazione sicuri e invia solo ciò che serve al widget per disegnare.

3. Il pattern da ricordare per chi inizia

I controlli nativi sono la scelta predefinita. Il Widget serve per le visualizzazioni speciali. C# governa i dati e le regole; JavaScript governa il disegno della parte visuale. Il confine dell'integrazione deve essere piccolo, con nomi chiari e facile da testare.



Sei pronto per il lab

Prossimo passo: guarda la videoguia sul widget, supera la verifica delle conoscenze, poi costruisci un piccolo widget JavaScript di stato/grafico — alimentato con dati di visualizzazione sicuri da un servizio C#, aggiornato tramite un evento server, con una scheda dati per verificare i valori.