

Widget personalizzati, estensioni, temi e capstone

Guida della lezione · Guida al lab / esame

● LearnWisej

Guida della lezione

Controlli nativi contro widget, componenti, extender provider e pacchetti di icone; i Packages del controllo Widget, InitScript, WidgetEvent e il collegamento CallAsync; temi e documentazione; il capstone di Operations Console. Che cosa significa per uno sviluppatore Wisej.NET — e come affrontare questo modulo.



Focus di produzione

Controlli nativi contro widget, componenti, extender provider e pacchetti di icone; i Packages del controllo Widget, InitScript, WidgetEvent e il collegamento CallAsync; temi e documentazione; il capstone di Operations Console.

1. Il problema in produzione

Prima o poi un prodotto ha bisogno di qualcosa che la libreria nativa non offre: una capacità specializzata del browser, una libreria JavaScript di terze parti, una visualizzazione su misura o un'integrazione riutilizzabile. La tentazione è buttarsi nel JavaScript e perdere il modello di Wisej.NET: stato sul server, temi, designer e un flusso di eventi che un altro sviluppatore riesce a seguire. Questo modulo integra un widget client senza rinunciare a nulla di tutto ciò, e poi porta a termine il capstone di Operations Console.

2. Concetti chiave

Concetto	Perché è importante
controllo nativo	Un controllo che implementa sia il lato server sia il proprio widget nel browser, quindi ha già supporto a design time, temi, eventi e stato sul server; è sempre la prima tappa.
tipi di estensione	Control, Widget, Component, Extender Provider e Icon Pack: dare un nome al tipo evita di confondere "riutilizzabile" con "nuovo controllo nativo".
controllo Widget	Un contenitore client generico che carica package JavaScript e CSS, esegue uno script di inizializzazione e scambia eventi e dati con il codice sul server.
Packages e InitScript	L'elenco dei package dichiara i file client da cui il widget dipende; l'InitScript crea l'oggetto JavaScript, collega i suoi eventi client ed espone solo le funzioni che servono al server.
WidgetEvent	L'evento server sollevato quando il codice client chiama <code>fireWidgetEvent</code> ; il suo tipo e il payload arrivano dal browser e vanno validati come qualsiasi altro input dell'utente.
CallAsync / Eval	La metà server-verso-client del collegamento: chiama una funzione esposta dal widget, oppure valuta uno script, senza uscire da C#.
contratto del widget	Le proprietà, gli eventi, il formato del payload, le dipendenze dai package e le regole di tema documentati, che permettono a un altro sviluppatore — o a un assistente IA ben istruito — di mantenere il widget.

3. Il filo della lezione

Prima il nativo, poi cinque tipi di estensione

La libreria di controlli è la prima tappa, sempre. Un controllo nativo implementa insieme il lato server e quello client, ed è per questo che partecipa gratuitamente al designer di Visual Studio, ad `Application.Theme`, al modello a eventi e allo stato sul server per sessione. Il codice personalizzato è riservato a ciò che la libreria non può coprire: una capacità del browser senza equivalente nativo, una libreria JavaScript di terze parti, una visualizzazione su misura o un'integrazione che vuoi riutilizzare.

Quando il codice personalizzato è giustificato, dai un nome al tipo di estensione che stai costruendo. Un **Control** è un controllo nativo completo, con una classe server e un widget client. Un **Widget** è il contenitore client generico per un widget JavaScript di terze parti o su misura. Un **Component** può vivere solo sul server oppure anche sul client. Un **Extender Provider** aggiunge proprietà e comportamenti ai controlli esistenti, come fanno `ToolTip` o `ErrorProvider`. Un **Icon Pack** confeziona risorse visive per il progetto e per il designer. Questa tassonomia previene la confusione progettuale: non tutto ciò che è riutilizzabile è uno `UserControl`, e non ogni integrazione JavaScript richiede un nuovo controllo nativo.

Per la Operations Console le regole di scelta sono brevi. Usa prima i controlli nativi. Usa uno `UserControl` per una composizione riutilizzabile lato server come l'editor cliente del Modulo 2. Usa `Widget` per interfacce JavaScript di terze parti che possono vivere dentro un contenitore Wisej.NET. Costruisci un'estensione completa solo quando l'integrazione va riutilizzata tra applicazioni diverse e deve comparire come componente di prima classe.

Il collegamento del Widget nelle due direzioni

Il controllo `Widget` fa tre cose. La sua collezione `Packages` carica, nell'ordine, i file JavaScript e CSS di cui la libreria client ha bisogno. Il suo `InitScript` viene eseguito quando i package sono presenti: `this.init` crea l'oggetto client dentro `this.container`, collega gli eventi propri della libreria ed espone le poche funzioni che il server può chiamare. Infine trasporta messaggi: il codice client chiama `this.fireWidgetEvent("ratingChanged", { value: 4 })` e il server riceve un `WidgetEvent` il cui `WidgetEventArgs` espone il `Type` e il payload `Data`; il codice server risponde con `CallAsync` per invocare una funzione client, oppure con `Eval` per eseguire uno script.

Il widget di valutazione del lab mostra il giro completo: un clic su una stella emette l'evento, il gestore valida il valore, lo salva tramite un servizio e lo conferma al widget:

```
private async void ratingWidget_WidgetEvent(object sender, WidgetEventArgs e)
{
    if (e.Type != "ratingChanged") return;

    // Il payload arriva dal browser: validalo prima di fidarti.
    int value = Convert.ToInt32(e.Data.value);
    if (value < 1 || value > 5) return;

    await _ratings.SaveAsync(_customerId, value);
    await this.ratingWidget.CallAsync("setSaved", value);
    MessageBox.Show($"Valutazione {value} salvata.");
}
```

Usa il collegamento con attenzione. Il JavaScript resta concentrato su ciò che riguarda solo il browser: rendering, gesti, animazioni. Le regole di business restano sul server, dove sono protette, testate e condivise da tutte le schermate; duplicarne una in JavaScript ha senso solo come ottimizzazione lato client voluta, mai come unica copia. Un bug classico: dentro una callback della libreria `this` è l'oggetto della libreria, non il widget, quindi cattura `var me = this` in `init` e chiama `me.fireWidgetEvent`.

Un contratto che puoi debuggare, tematizzare e documentare

Un wrapper solido ha un contratto. Le proprietà pubbliche lato server corrispondono a configurazioni significative, non a ogni opzione che la libreria espone. Gli eventi rappresentano azioni dell'utente rilevanti per il business: `ratingChanged`, non `mouseup`. Se i payload contengono più di un paio di valori, definisci un DTO o una struttura JSON chiara; il gestore valida e normalizza i dati prima di usarli, perché un payload che arriva dal browser può essere modificato da chiunque abbia gli strumenti per sviluppatori.

Il debug dei widget comincia dalla console del browser, e la documentazione di Wisej.NET lo dice esplicitamente. Quando un widget si mostra vuoto o un evento non arriva mai, controlla prima la console, poi l'ordine di caricamento dei package, i percorsi delle risorse e i tempi di inizializzazione, e infine se il widget viene ricreato dopo un refresh o un cambio di layout — il che significa che `init` deve poter essere eseguito di nuovo senza danni. Gli script inseriti nell'HTML di avvio possono girare troppo presto per i widget Wisej.NET; risorse incorporate, `Packages`, `InitScript` e chiamate a livello di controllo sono i pattern più sicuri.

Temi e comportamento responsive sono la rifinitura finale. Se il widget deve intonarsi alla console, decidi in modo consapevole: esponi proprietà di stile, mappa le appearance del tema, usa gli extender StyleSheet o Animation, oppure distribuisci il CSS nel package del widget. Non scrivere i colori dell'applicazione dentro il JavaScript, a meno che il widget non sia deliberatamente indipendente. Applica le proprietà responsive per profilo client come hai fatto per lo shell, fornisci etichette accessibili e comportamento da tastiera dove è possibile, e scrivi la nota di grounding per l'IA: a che cosa serve il widget, perché nessun controllo nativo era adatto e dove si trova il contratto.

- Le dipendenze dai package e il loro ordine di caricamento sono documentati.
- Proprietà, eventi e formato del payload sono scritti accanto al codice.
- Le regole di business sono applicate sul server e la console del browser è pulita.



Errori tipici

Sostituire i controlli standard con JavaScript perché sembra più moderno. Mettere regole di business nel JavaScript client, dove sono più difficili da proteggere e da testare. Non gestire la reinizializzazione del widget dopo un refresh o un cambio di layout. Dimenticare di documentare le dipendenze dai package e il contratto degli eventi. Costruire un widget su misura che nessun altro nel team è in grado di mantenere.

Chiudere la Operations Console

Il capstone riunisce tutto il corso in un'unica applicazione, `OperationsConsole`: lo shell navigabile (Moduli 1 e 3), l'editor cliente validato (Modulo 2), l'esploratore con TreeView e ListView (Modulo 4), la `DataGridView` degli ordini (Modulo 5), la dashboard e la schermata dei contenuti (Modulo 6) e la schermata con il widget o l'estensione personalizzata di questo modulo. Ogni schermata viene valutata sulle domande poste dai lab: quale controllo fa il lavoro più importante, dove vive lo stato, che cosa succede con due sessioni o con dieci volte i dati, e quale codice appartiene a uno `UserControl` o a un servizio riutilizzabile.

La griglia di valutazione pesa la scelta dei controlli e i controlli dati al 20% ciascuno, e layout e riuso, validazione e feedback, contratto del widget personalizzato, rifinitura e documentazione al 15% ciascuno. Prepara una demo di cinque minuti che segua lo script del capstone e scrivi `CapstoneNotes.md` partendo dal template del workbook: controlli usati e perché, ipotesi sulla dimensione dei dati, scelte di validazione e layout, codice riutilizzabile estratto, e una cosa che miglioreresti in produzione. Il codice deve essere comprensibile a un altro sviluppatore e spiegabile da un assistente IA che abbia le tue note come base.



La regola da libreria di controlli

Scegli il controllo nativo più semplice che esprime il compito dell'utente prima di scrivere codice personalizzato. I controlli nativi ti danno già supporto a design time, temi, eventi e stato sul server; il passo successivo è un extender o uno `UserControl`, e solo quando nessun controllo nativo è adatto si costruisce un widget JavaScript.



Obiettivi di apprendimento

Alla fine di questo modulo sarai in grado di:

- ✓ Spiegare la differenza tra controlli nativi, widget, componenti, extender provider e pacchetti di icone.
- ✓ Usare il controllo Widget per caricare package JavaScript/CSS, inizializzare un widget client e scambiare eventi e dati con il codice sul server.
- ✓ Applicare temi, proprietà responsive, note di accessibilità e note di grounding per l'IA a un pacchetto di controlli personalizzato.
- ✓ Completare e presentare il capstone di Operations Console usando controlli nativi più almeno un widget o un'estensione personalizzata.

Guida al lab / esame

Che cosa costruirai nel lab pratico, l'approccio suggerito e i deliverable richiesti.



Lab pratico

Incapsula un piccolo widget JavaScript di rating o di stato con il controllo Widget, oppure configura un pacchetto di estensione Wisej.NET: carica i suoi package JavaScript e CSS, solleva un evento lato server dal client con il pattern degli eventi widget, chiama una funzione lato client dal server con CallAsync o Eval, applica il tema della console, poi rifinisci il capstone completo di Operations Console e prepara una demo di 5 minuti.

Punto di partenza

Apri la soluzione `OperationsConsole` che ti sei portato dietro dai Moduli 1 a 6: lo shell, l'editor cliente, l'esploratore documenti, la griglia Orders e la dashboard esistono tutti, e il tab Widgets è ancora la pagina segnaposto del Modulo 1. Tieni il lavoro di questo modulo in una cartella o in un namespace dedicati, così che il capstone possa riutilizzarlo.

Approccio suggerito

- ✓ Decidi, e metti per iscritto, perché nessun controllo nativo è adatto al rating o all'indicatore che stai aggiungendo; se un controllo nativo invece va bene, usalo e scegli un altro obiettivo per il widget.
- ✓ Inserisci un controllo `Widget` nella pagina Widgets, aggiungi i file JavaScript e CSS alla sua collezione `Packages` nell'ordine di caricamento, e scrivi un `InitScript` il cui `init` crea l'oggetto client dentro `this.container` e può essere eseguito di nuovo dopo un refresh senza problemi.
- ✓ Nella callback client cattura `var me = this` e chiama `me.fireWidgetEvent("ratingChanged", { value })`; sul server gestisci `WidgetEvent`, controlla `e.Type`, valida e normalizza `e.Data`, poi salva tramite un servizio.
- ✓ Conferma al browser con `CallAsync("setSaved", value)` o con `Eval`, dai all'utente un Toast o un `AlertBox`, e applica il tema al widget tramite le appearance del tema, il CSS nel package o l'extender `StyleSheet`, invece che con colori scritti dentro il JavaScript.
- ✓ Scrivi il contratto accanto al codice (package, proprietà, eventi, formato del payload, note di debug), poi ripassa ogni schermata del capstone rispetto alla griglia di valutazione e prova lo script della demo da cinque minuti.

Deliverable richiesti

- ✓ Schermata basata su Widget o estensione con i suoi package JavaScript e CSS caricati
- ✓ Evento client-verso-server sollevato con il pattern degli eventi widget o con un evento nativo dell'estensione
- ✓ Chiamata server-verso-client tramite CallAsync, Eval o l'API dell'estensione
- ✓ Contratto client/server documentato: package, proprietà, eventi e formato del payload

- ✓ Capstone di Operations Console rifinito con uno script di demo da 5 minuti

Domande di autovalutazione

- ✓ Che cosa succederebbe se un utente modificasse il payload di `ratingChanged` negli strumenti per sviluppatori del browser prima che arrivi al tuo gestore, e quale riga del tuo codice lo blocca?
- ✓ Il tuo widget si mostra vuoto dopo che il tab Widgets è stato nascosto e riaperto: quali tre cose controlli, in quale ordine, e perché l'HTML di avvio non è la soluzione?
- ✓ Quale regola del tuo widget potresti spostare in JavaScript come ottimizzazione, e quale non deve mai lasciare il server?

Consegna del capstone

Il lab di questo modulo è anche la consegna del capstone. La Operations Console deve comprendere sei schermate: lo shell applicativo con aree di navigazione, comandi e stato; l'editor cliente o contatti validato; l'esploratore TreeView/ListView con ID stabili e una strategia lazy o virtuale; la DataGridView degli ordini con colonne esplicite, formattazione e una strategia per i grandi volumi; la dashboard e la schermata dei contenuti con ChartJS e un'anteprima di documento; e la schermata con il Widget o l'estensione personalizzata di questo lab. Consegna il progetto sorgente, gli screenshot di ogni schermata, `CapstoneNotes.md` scritto a partire dal template del workbook, e una demo di cinque minuti (dal vivo o registrata) che segua questo script:

- ✓ Spiega l'architettura dell'applicazione dal punto di vista della libreria di controlli, poi mostra l'editor e il suo comportamento in validazione.
- ✓ Mostra la navigazione ad albero e lista con il caricamento del dettaglio, poi la griglia e come gestisce i grandi volumi.
- ✓ Mostra la dashboard e i controlli di contenuto, poi il widget o l'estensione personalizzata, inclusa la comunicazione a eventi nelle due direzioni.
- ✓ Spiega che cosa dovrebbe rispondere il tuo assistente IA se gli si chiedesse perché è stato scelto ciascun controllo.

Area di valutazione	Peso	Che cosa significa "eccellente"
Scelta dei controlli	20%	Controlli nativi, extender, UserControl, widget ed estensioni usati in modo appropriato e con una motivazione chiara.
Layout e riuso	15%	Contenitori, docking, ancoraggio, splitter, tab e UserControl usati in modo pulito.
Controlli dati	20%	Pattern di TreeView, ListView e DataGridView con identità, binding e strategia per i grandi volumi corretti.
Validazione e feedback	15%	Proprietà degli editor, ErrorProvider e notifiche usati in modo chiaro e coerente.
Widget o estensione personalizzata	15%	Un contratto client/server documentato, eventi gestiti, nessuna logica di business fuori posto.
Rifinitura e documentazione	15%	Tema, comportamento responsive, commenti e note di grounding per l'IA pensati per la produzione.



La regola da libreria di controlli

Scegli il controllo nativo più semplice che esprime il compito dell'utente prima di scrivere codice personalizzato. I controlli nativi ti danno già supporto a design time, temi, eventi e stato sul server; il passo successivo è un extender o uno UserControl, e solo quando nessun controllo nativo è adatto si costruisce un widget JavaScript.