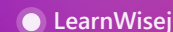


Editor, pulsanti, validazione e feedback

Guida della lezione · Guida al lab / esame



Guida della lezione

Scegliere l'editor adatto al valore, un modello a comandi per i pulsanti, una validazione con `ErrorProvider` che aiuta invece di bloccare, e feedback non bloccante con `Toast` e `AlertBox`. Che cosa significa per uno sviluppatore Wisej.NET — e come affrontare questo modulo.



Focus di produzione

Scegliere l'editor adatto al valore, un modello a comandi per i pulsanti, una validazione con `ErrorProvider` che aiuta invece di bloccare, e feedback non bloccante con `Toast` e `AlertBox`.

1. Il problema in produzione

Gli editor sono il primo punto in cui una libreria di controlli diventa un sistema gestionale. Una maschera che sembra corretta ma accetta input ambigui o non validi genera ticket di assistenza e script di bonifica molto tempo dopo il rilascio. Gli editor di Wisej.NET seguono il familiare modello a componenti di .NET ma girano nel browser, quindi una stessa regola di business può essere applicata su più livelli: proprietà degli editor, extender di validazione, gestori di eventi, data binding, validazione nei servizi e vincoli del database. Il compito del livello UI è impedire gli errori evidenti e spiegare chiaramente quelli che restano; non deve mai essere l'unico livello che protegge l'applicazione.

2. Concetti chiave

Concetto	Perché è importante
editor coerente con il valore	L'editor il cui tipo vincola già l'input: <code>DateTimePicker</code> per le date, <code>NumericUpDown</code> per i numeri con limiti, <code>ComboBox</code> per i valori da elenco, così il server non deve mai interpretare testo ambiguo.
modello a comandi	Una distinzione voluta tra comandi primari (Save, Approve, Search, Send, Export), comandi secondari (ToolBar, MenuBar, SplitButton, menu contestuale) e comandi distruttivi, che richiedono conferma o una collocazione prudente.
ErrorProvider	L'extender che associa un messaggio specifico e utile al controllo che è sbagliato, invece di un generico "Valore non valido" da qualche altra parte nella pagina.
feedback bloccante e non bloccante	<code>MessageBox</code> interrompe per una decisione o una conferma critica; <code>AlertBox</code> e <code>Toast</code> comunicano lo stato senza fermare il flusso di lavoro.
validatori con un nome	Metodi come <code>ValidateEmail</code> o <code>ValidateCreditLimit</code> che contengono una regola ciascuno, così la regola è testabile e spiegabile invece di essere sepolta in un gestore del click.
stato di attesa	Disabilitare Save e mostrare un avanzamento mentre un'operazione è in corso: evita gli invii doppi ed è parte dell'affidabilità, non decorazione.
UserControl editor	Una composizione di label, editor, un <code>ErrorProvider</code> e un metodo di validazione che restituisce un oggetto risultato, così che il resto dell'applicazione non sappia mai come viene controllato il campo email.

3. Il filo della lezione

Abbina l'editor al valore

La prima decisione su qualsiasi schermata di input è quale editor rappresenta ciascun valore: scegli il controllo il cui tipo esclude già l'input impossibile. La `TextBox` serve per il testo ordinario. Imposta `Multiline` solo quando l'utente deve scrivere paragrafi, non perché un campo è largo, e usa `Watermark` come esempio o suggerimento, mai come sostituto di una label. Quando la forma dell'input è prevedibile — un codice, un numero simile a un telefono, un numero di riferimento — `MaskedTextBox` impone quella forma mentre l'utente digita. Quando il cuore del campo è interpretare e formattare un valore tipizzato, `TypedTextBox` fa la conversione al posto tuo.

Date e numeri meritano i propri editor. `DateTimePicker` elimina l'ambiguità di interpretazione e valida la scelta di una data molto più naturalmente del testo libero. `NumericUpDown`, `DomainUpDown` e `TimeUpDown` esprimono valori vincolati con un minimo, un massimo e un passo. `ComboBox` è la scelta giusta quando un valore proviene da un elenco; il suo `DropDownStyle` decide se l'utente può anche digitare un valore che nell'elenco non c'è. È invece una scelta scadente quando l'utente deve cercare o esplorare un insieme di dati molto grande: esiste per una selezione compatta da un elenco piccolo o medio.

Nella Operations Console il `CustomerEditor` applica esattamente questa mappatura: nome ed email sono controlli `TextBox`, stato e tipo cliente sono `ComboBox` con voci fisse, la data di inizio è un `DateTimePicker` e il fido è un `NumericUpDown` con un intervallo. Con gli elenchi conta un'abitudine in particolare: tieni distinti il testo visualizzato e la chiave memorizzata, così che "Attivo" a schermo non diventi mai il valore che salvi.

I pulsanti come comandi, e i controlli di feedback

I pulsanti non sono semplici rettangoli cliccabili: una schermata gestionale ha bisogno di un modello a comandi. I comandi primari (Save, Approve, Search, Send, Export) sono normali controlli `Button` collocati dove l'occhio se li aspetta. I comandi secondari stanno in una `ToolBar`, in una `MenuBar`, in uno `SplitButton` o in un menu contestuale. Uno `SplitButton` è la scelta giusta quando un comando ha un'azione predefinita e delle alternative, per esempio Save accanto a Save and Close. I comandi distruttivi richiedono una conferma, uno stato disabilitato o una collocazione chiara. `CheckBox` e `RadioButton` sono controlli di stato, non comandi: stanno dove l'utente fa una scelta, mai dove parte un'azione.

Ciascun controllo di feedback ha un solo ruolo. `ToolTip` e `HelpTip` portano indicazioni su che cosa si aspetta un campo; non sono un canale per gli errori. `ErrorProvider` segnala il controllo che ha un errore di validazione, e per i controlli in binding va associato al contenitore giusto, altrimenti le sue icone compaiono nel posto sbagliato. `MessageBox` è riservato a una decisione bloccante o a una conferma critica. `AlertBox` e `Toast` servono per messaggi di stato che non devono interrompere il flusso di lavoro e, poiché un `Toast` può essere riutilizzato e aggiornato mentre è visibile, funziona anche come indicatore di avanzamento per un salvataggio lungo, insieme a una `ProgressBar` quando l'avanzamento reale è noto.

Una validazione che aiuta senza bloccare

La validazione deve essere visibile, specifica e recuperabile, e deve arrivare abbastanza presto da aiutare senza intralciare la digitazione. Il flusso pulito ha sei passi. Primo: imposta le proprietà degli editor (`MaxLength`, `Minimum`, `Maximum`, voci fisse) così che i valori impossibili non si possano proprio inserire. Secondo: usa un parsing specifico per i valori tipizzati. Terzo: alla validazione del campo, nel gestore `Validating` del controllo, imposta o azzera il messaggio dell' `ErrorProvider` per quel singolo controllo. Quarto: al Save esegui la validazione a livello di maschera e poi quella a livello di servizio. Quinto: se qualcosa fallisce, tieni l'utente sulla schermata, porta il focus sul primo controllo non valido e mostra errori specifici. Sesto: quando il salvataggio riesce, mostra una conferma non bloccante e aggiorna l'area di stato.

Il messaggio deve dire che cosa correggere: "Inserisci un fido tra 0 e 250.000" invece di "Valore non valido". Evita di ammassare tutta la validazione nei gestori del click: rende la schermata difficile da testare e difficile da spiegare per un collega o per un assistente IA. Metti ogni regola vicino al suo controllo o in un validatore con un nome come `ValidateRequiredName`, `ValidateEmail` e `ValidateCreditLimit`, e lascia che sia Save a chiamare quei metodi. Tieni esplicito il percorso di salvataggio — valida, salva, aggiorna lo stato, notifica — e disabilita i comandi mentre è in corso, così un doppio clic non può inviare due volte.

```

public EditorResult ValidateContent()
{
    var result = new EditorResult();
    this.errorProvider.SetError(this.txtEmail, "");
    this.errorProvider.SetError(this.numCreditLimit, "");
    if (!this.txtEmail.Text.Contains("@"))
        result.Add(this.txtEmail, "Inserisci un indirizzo email come nome@azienda.com.");
    if (this.numCreditLimit.Value < 0 || this.numCreditLimit.Value > 250000)
        result.Add(this.numCreditLimit, "Inserisci un fido tra 0 e 250.000.");
    foreach (var error in result.Errors)
        this.errorProvider.SetError(error.Control, error.Message);
    result.FirstInvalid?.Focus();
    return result;
}

```

Lo UserControl CustomerEditor

È il punto in cui un insieme di controlli diventa un componente di livello applicativo. Il **CustomerEditor** del lab è uno **UserControl** che possiede le proprie label, i propri editor, un solo **ErrorProvider**, i pulsanti Save, Reset e Validate e un metodo di validazione che restituisce un oggetto risultato chiaro: quali controlli non sono validi e quali messaggi sono stati mostrati. La pagina shell che lo ospita chiede soltanto "è valido?" e "dammi il cliente"; non viene mai a sapere come funziona la regola sull'email. Il metodo si chiama **ValidateContent** per non confonderlo con i membri di validazione che un controllo contenitore espone già.

Le regole di progettazione per le schermate di input discendono da quel confine e sono la checklist su cui il docente valuta:

- Ogni campo modificabile ha una label visibile o un equivalente accessibile.
- L'editor corrisponde al tipo di valore; date e numeri non vengono interpretati da normali **TextBox** senza una ragione forte.
- I messaggi di validazione dicono che cosa fare, **ErrorProvider** porta gli errori di campo e **Toast** o **AlertBox** il feedback sul flusso.
- I comandi sono disabilitati mentre un'operazione è in corso, e il percorso di salvataggio è esplicito: valida, salva, aggiorna lo stato, notifica.



Errori tipici

Usare **MessageBox** per ogni piccolo messaggio di stato. Salvare uno stato parzialmente non valido perché la validazione ha girato solo sui controlli visibili. Nascondere la validazione dentro gestori di eventi anonimi. Leggere il valore di una **ComboBox** senza distinguere il testo visualizzato dalla chiave memorizzata. Dimenticare che pulsanti disabilitati e feedback di attesa fanno parte dell'affidabilità, non della decorazione.



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:

- ✓ Scegliere l'editor corretto per input testuale, numerico, di data/ora, con maschera e di selezione.
- ✓ Usare in modo appropriato Button, SplitButton, CheckBox, RadioButton, ToolTip, HelpTip, ErrorProvider, MessageBox, Toast e ProgressBar.
- ✓ Realizzare una validazione abbastanza immediata da aiutare l'utente ma non così aggressiva da bloccare il normale inserimento dei dati.
- ✓ Creare uno UserControl editor riutilizzabile per i dati di clienti e contatti.

Guida al lab / esame

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



Lab pratico

Costruisci uno UserControl CustomerEditor per la Operations Console con campi etichettati per nome, email, stato, data di inizio, fido e tipo cliente, usando per ogni valore l'editor adatto invece di una TextBox per tutto. Aggiungi messaggi ErrorProvider per i campi obbligatori, la forma dell'email, l'intervallo del fido e le date non valide; comandi Save, Reset e Validate con comportamento disabilitato e di attesa durante il salvataggio; e un Toast o un MessageBox quando la validazione o il salvataggio si conclude.

Punto di partenza

Apri la soluzione `OperationsConsole` del Modulo 1: la pagina shell con le sue aree di navigazione, contenuti, comandi e stato è già funzionante e la pagina segnaposto Editors è vuota. Tieni il lavoro di questo modulo in una cartella e in un namespace `Editors` dedicati, così che il capstone possa riutilizzare il controllo senza modifiche.

Approccio suggerito

- ✓ Abbozza i sei campi e annota l'editor di ciascuno prima di aprire il designer: `TextBox` per nome ed email, `ComboBox` per stato e tipo cliente, `DateTimePicker` per la data di inizio, `NumericUpDown` per il fido.
- ✓ Aggiungi uno UserControl `CustomerEditor`, inserisci le label e gli editor con nomi chiari (`txtName`, `cboStatus`, `dtpStartDate`, `numCreditLimit`), imposta `Minimum`, `Maximum`, `MaxLength` e voci fisse in modo che i valori impossibili non si possano digitare, e dai a ogni editor un `ToolTip` che descrive che cosa si aspetta.
- ✓ Aggiungi un solo `ErrorProvider` e dei validatori con un nome (`ValidateRequiredName`, `ValidateEmail`, `ValidateCreditLimit`, `ValidateStartDate`) che impostano o azzerano un messaggio specifico; chiama il validatore corrispondente dall'evento `Validating` di ciascun editor e tutti quanti da un metodo `ValidateContent` che restituisce un oggetto risultato.
- ✓ Aggiungi i pulsanti Save, Reset e Validate; fai in modo che Save esegua `ValidateContent`, poi la chiamata al servizio, poi un aggiornamento dello stato, e disabilita tutti e tre i comandi mostrando un feedback di attesa mentre il salvataggio è in corso.
- ✓ Mostra un `Toast` o un `MessageBox` quando la validazione o il salvataggio si conclude, aggiorna l'area di stato dello shell e ospita il controllo finito nella pagina Editors.

Deliverable richiesti

- ✓ UserControl CustomerEditor con campi etichettati per nome, email, stato, data di inizio, fido e tipo cliente
- ✓ Editor coerenti con il valore: TextBox, DateTimePicker, NumericUpDown e ComboBox invece di testo libero ovunque

- ✓ Messaggi `ErrorProvider` per campi obbligatori, forma dell'email, intervallo del fido e date non valide
- ✓ Comandi `Save`, `Reset` e `Validate` con comportamento disabilitato e di attesa durante il salvataggio
- ✓ Feedback con `Toast` o `AlertBox` quando la validazione o il salvataggio si conclude

Domande di autovalutazione

- ✓ Quali errori di validazione non possono più verificarsi grazie all'editor che hai scelto, e quali hanno ancora bisogno di un messaggio `ErrorProvider`?
- ✓ Se l'utente fa clic su `Save` due volte in un secondo, che cosa fa la tua schermata, e in quale punto del codice è garantito?
- ✓ Che cosa sa la pagina shell su come viene validato il campo email, e che cosa dovrebbe cambiare se quella regola si spostasse in un servizio?



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.