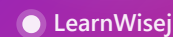


# UX di validazione in produzione, error provider personalizzati e capstone

Guida della lezione · Guida al lab / esame



## Guida della lezione

Mettere insieme tutti i livelli: una pipeline di salvataggio noiosa e ripetibile, un `IErrorProvider` personalizzato che alimenta un pannello di riepilogo, provider a icona e provider di riepilogo fianco a fianco, dieci regole di UX per la validazione e il capstone `ValidationClinic` che dimostra dove va ogni verifica. Che cosa significa per uno sviluppatore che conosce le basi di C# e sta iniziando a sviluppare form con Wisej.NET — e come affrontare questo modulo.



### Focus di produzione

Mettere insieme tutti i livelli: una pipeline di salvataggio noiosa e ripetibile, un `IErrorProvider` personalizzato che alimenta un pannello di riepilogo, provider a icona e provider di riepilogo fianco a fianco, dieci regole di UX per la validazione e il capstone `ValidationClinic` che dimostra dove va ogni verifica.

## 1. Il problema in produzione

Un utente che compila il form di accettazione di `ValidationClinic` non si chiede mai quale API abbia trovato un errore; nota solo se la schermata gli dice che cosa non va e gli permette di finire. Dopo sei moduli l'app ha gestori dei campi, regole dell'extender, un servizio di validazione, modelli annotati e regole della griglia, ma ognuno è stato aggiunto per conto suo, e niente garantisce ancora che vengano eseguiti tutti, nell'ordine giusto, ogni volta che si fa clic su Save. Questo modulo unisce i livelli in un'unica pipeline di salvataggio prevedibile e in un unico riepilogo visibile, poi ti chiede di dimostrare il design in un capstone.

## 2. Concetti chiave

| Concetto                               | Perché è importante                                                                                                                                                                                                                     |
|----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Pipeline di salvataggio                | Una sequenza fissa dentro <code>btnSave_Click</code> che conferma le modifiche in sospeso, esegue a turno ogni livello di validazione e termina al primo livello che fallisce, così nulla arriva al repository senza essere verificato. |
| Uscita anticipata                      | Ogni fase chiude il gestore con <code>return</code> quando trova errori, così le fasi successive non vengono mai eseguite su dati che una fase precedente ha già rifiutato.                                                             |
| <code>IErrorProvider</code>            | La piccola interfaccia a cui il sistema di validazione di Wisej.NET invia i messaggi: <code>SetError(Control, string)</code> per impostare o cancellare un messaggio e <code>GetError(Control)</code> per rileggerlo.                   |
| <code>SummaryErrorProvider</code>      | Il tuo <code>IErrorProvider</code> che mantiene un dizionario di coppie controllo-messaggio e riscrive da esso una label di riepilogo ogni volta che un errore viene impostato o cancellato.                                            |
| Doppio provider                        | Un unico metodo helper <code>SetError</code> che scrive lo stesso messaggio nell' <code>ErrorProvider</code> integrato (l'icona accanto al campo) e nel provider di riepilogo (l'elenco in alto).                                       |
| Errore dell'utente o errore di sistema | Uno sbaglio che l'utente può correggere, come un nome duplicato, diventa un messaggio sul campo; un'eccezione imprevista viene registrata nel log per gli sviluppatori e mostrata come un messaggio generico e tranquillo.              |
| Test del valore limite                 | Un valore di test che sta esattamente sul bordo di una regola, come un'età di 0 o 120, e che scopre gli errori off-by-one che i normali valori validi e non validi non colgono.                                                         |

## 3. Il filo della lezione

### Un pulsante Save volutamente noioso

Un buon gestore di Save esegue gli stessi passi nello stesso ordine a ogni clic e si ferma non appena qualcosa non va. In ValidationClinic l'ordine è: azzera il riepilogo, chiama `bindingSource.EndEdit()` così che il valore ancora in digitazione venga trasferito nel modello, chiama `ValidateChildren()` così che vengano eseguite le regole di campo di ogni controllo, esegui la verifica DataAnnotations con `Validator.TryValidateObject`, esegui `ContactValidator.Validate` per le regole di business, controlla che nessuna riga della griglia abbia ancora un errore, e solo allora salva. `EndEdit` viene per primo perché altrimenti ogni verifica del modello successiva esaminerebbe dati non aggiornati.

```

ResetSummary();
bindingSource.EndEdit();
if (!this.ValidateChildren()) return;

var model = (ContactEditModel)bindingSource.Current;
if (!ApplyAnnotationErrors(model)) return;

var businessErrors = contactValidator.Validate(model);
if (businessErrors.Count > 0) { ApplyBusinessErrors(businessErrors); return; }

if (GridHasErrors()) return;

try { SaveContact(model); MessageBox.Show("Salvato.", MessageBoxIcon.Information); }
catch (DuplicateNameException) { SetError(txtName, "Esiste già un contatto con questo nome."); }

```

I nomi degli helper qui ( `ResetSummary` , `ApplyAnnotationErrors` , `GridHasErrors` ) sono tuoi piccoli metodi, non membri del framework. Il salvataggio vero e proprio sta dentro `try / catch` , e il `catch (Exception ex)` finale (non mostrato) scrive l'eccezione con `System.Diagnostics.Debug.WriteLine` e mostra un testo comprensibile con `MessageBox.Show` . La pipeline è noiosa di proposito: quando un tester segnala "Save ha lasciato passare un'email errata", puoi indicare la fase esatta che avrebbe dovuto fermarla.

## Un pannello di riepilogo tramite `IErrorProvider`

Al sistema di validazione di Wisej.NET non importa che cosa mostri i suoi messaggi, ma solo che la destinazione implementi `IErrorProvider` . L' `ErrorProvider` integrato è un'implementazione e una `Label` può esserne un'altra. Il tuo `SummaryErrorProvider` è la terza: il costruttore riceve la label di riepilogo, `SetError` rimuove la voce del controllo quando il messaggio è vuoto o altrimenti la memorizza, poi unisce i messaggi distinti nel `Text` della label e imposta `Visible` solo finché esiste almeno un errore. `GetError` restituisce il messaggio memorizzato o una stringa vuota.

Lo colleghi con due righe: crea il provider attorno a `validationSummaryLabel` , poi assegna `validation.ErrorProvider = summaryProvider` . Da quel momento ogni regola eseguita dall'extender `Validation`, compresa la tua `MinimumAgeValidationRule` del Modulo 4, riporta i suoi messaggi nel pannello senza alcun codice aggiuntivo nella regola stessa.

## Errori di campo, della griglia e del modello in un unico posto

Ora gli errori arrivano da tre direzioni. Gli errori di campo vengono dai gestori `Validating` e dalle regole dell'extender. Gli errori del modello vengono dai nomi dei membri di `ValidationResult` e dai messaggi di `ContactValidator` , che riconduci ai controlli con la mappa dei campi del Modulo 5. Gli errori della griglia stanno su celle e righe come `ErrorText` , impostato in `CellValidating` , `ValidateRow` e `DataError` . L'utente non dovrebbe aver bisogno di sapere quale di questi ha prodotto un messaggio.

Per ottenerlo, instrada gli errori del form e del modello attraverso un unico metodo helper `SetError(Control, string)` nel form, che chiama `errorProvider.SetError` per l'icona e `summaryProvider.SetError` per l'elenco. Gli errori della griglia restano sulla griglia, dove l'utente vede la cella rossa, e la fase della griglia nella pipeline aggiunge una riga di riepilogo come "Riga 3: i contatti chiusi

richiedono una data di chiusura." così che il problema sia visibile anche in alto. La cancellazione funziona allo stesso modo al contrario: un messaggio vuoto rimuove insieme l'icona e la riga di riepilogo.

## Dieci regole per una validazione che le persone sanno usare

Buona parte di una buona UX di validazione è disciplina, non codice. Le dieci regole qui sotto, raggruppate in sei righe, sono la checklist con cui viene valutato il capstone:

- Metti ogni messaggio accanto al suo campo e ripetilo in un riepilogo per l'intero form.
- Scrivi con un linguaggio semplice che dica che cosa fare dopo, e non dare mai la colpa all'utente.
- Non mostrare mai stack trace, testo SQL o eccezioni di conversione.
- Non salvare mai dati non validi e non intrappolare mai l'utente: non bloccare Cancel o Close senza un motivo reale.
- Tieni le verifiche lente o remote fuori dalla validazione a ogni tasto premuto; esegui al Save.
- Esegui le stesse regole del modello dal form, dalla griglia e da qualsiasi servizio, e testa i casi obbligatorio, non valido, valido e valore limite.

Un pannello di riepilogo conta anche per l'accessibilità: un errore che compare solo quando qualcuno passa il mouse su un'icona è invisibile per chi usa la tastiera e facile da perdere per tutti. Un messaggio che uno screen reader può raggiungere come testo fa parte del design, non è decorazione.

## Il capstone ValidationClinic

Il capstone non è un'applicazione grande. È la prova che sai dove va ogni verifica. ValidationClinic deve avere nome ed email obbligatori, un `ErrorProvider` con un riepilogo visibile, l'extender Validation con almeno tre regole integrate, una classe `ValidationRule` personalizzata, un servizio di validazione usato da Save, validazione del modello con DataAnnotations o `IDataErrorInfo`, la validazione delle celle di email ed età più un gestore `DataError` sulla griglia, il riepilogo con l' `IErrorProvider` personalizzato e un breve README che spieghi il design.

Poi lo dimostri con uno script breve e fisso: lascia un campo non valido, fai clic su Save e guarda `ValidateChildren` fermarlo, mostra un errore del modello che compare solo dopo `EndEdit`, digita un'email errata in una cella della griglia, indica il pannello di riepilogo che elenca tutto, poi correggi ogni problema e salva con successo. Scrivi per prima la tabella dei test del README, con i casi obbligatorio, non valido, valido e valore limite per ogni regola, e la demo diventa questione di eseguire ad alta voce i tuoi stessi test.



### La regola dei livelli di validazione

Dai un nome al livello prima di scrivere la verifica. La validazione dei campi protegge la modifica e viene eseguita quando l'utente lascia un controllo; la validazione del modello protegge i dati ed esegue le stesse regole dal form, dalla griglia e da qualsiasi servizio; la validazione al salvataggio protegge il database ed esegue di nuovo tutti i livelli prima che venga scritto qualsiasi dato. Mostra ogni errore vicino al suo campo e in un riepilogo visibile, con un linguaggio semplice che dica che cosa fare, e tieni gli errori correggibili dall'utente separati dagli errori di sistema.



## **Obiettivi di apprendimento**

Alla fine di questo modulo sarai in grado di:

- ✓ Costruire una pipeline di salvataggio completa per un form Wisej.NET di livello base.
- ✓ Creare un `IErrorProvider` personalizzato per un pannello di riepilogo.
- ✓ Combinare errori di campo, errori della griglia ed errori del modello.
- ✓ Progettare una UX di validazione chiara, accessibile e testabile.
- ✓ Completare l'applicazione capstone del corso.

## Guida al lab / esame

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



### Lab pratico

Completa il capstone ValidationClinic. Scrivi `btnSave_Click` come una pipeline fissa: azzera il riepilogo, chiama `bindingSource.EndEdit()`, fermati se `ValidateChildren()` fallisce, fermati se la verifica `DataAnnotations` fallisce, fermati se `ContactValidator` restituisce errori di business, fermati se la griglia dei contatti ha ancora errori di riga, e solo allora salva dentro un `try/catch` che trasforma un nome duplicato in un errore di campo e qualsiasi altra eccezione in un messaggio comprensibile. Crea `SummaryErrorProvider` che implementa `IErrorProvider`, assegnalo a `validation.ErrorProvider` e instrada gli errori attraverso un metodo helper `SetError` che scrive sia nell'`ErrorProvider` a icona sia nel provider di riepilogo. Testa i casi obbligatorio, non valido, valido e valore limite e scrivi un breve README che spieghi il design della validazione.

## Punto di partenza

ValidationClinic come l'hai lasciato dopo il Modulo 6: il form di accettazione ha gestori dei campi, regole dell'extender, `MinimumAgeValidationRule`, un servizio `ContactValidator` e un `ContactEditModel` collegato tramite un `BindingSource`, e la griglia dei contatti valida le celle di email, età e data di chiusura. Ogni livello funziona da solo, ma Save non li esegue ancora tutti in ordine, e gli errori compaiono in punti diversi a seconda del livello che li ha trovati.

## Approccio suggerito

- ✓ Scrivi per primo `SummaryErrorProvider`: una classe che implementa `IErrorProvider`, riceve la `Label` di riepilogo nel costruttore, memorizza coppie controllo-messaggio, rimuove una voce quando il messaggio è vuoto e riscrive testo e visibilità della label a ogni `SetError`. Implementa `GetError` in modo che restituisca una stringa vuota per un controllo senza errori.
- ✓ Crea il provider attorno a `validationSummaryLabel`, assegna `validation.ErrorProvider = summaryProvider` e aggiungi a livello di form un metodo helper `SetError(Control, string)` che chiama `errorProvider.SetError` e `summaryProvider.SetError` con lo stesso messaggio. Sostituisci con l'helper le chiamate dirette a `SetError` nel codice che mappa gli errori del modello e di business.
- ✓ Riscrivi `btnSave_Click` come pipeline fissa: azzera il riepilogo, `bindingSource.EndEdit()`, `ValidateChildren()`, la verifica con `Validator.TryValidateObject`, `ContactValidator.Validate`, poi un controllo della griglia che salta la riga nuova segnaposto e segnala ogni riga che ha ancora un `ErrorText`. Ogni fase termina non appena fallisce.
- ✓ Racchiudi in `try / catch` solo il salvataggio: intercetta prima `DuplicateNameException` e trasformala in un errore su `txtName` tramite il tuo helper, poi intercetta `Exception`, registrala con `System.Diagnostics.Debug.WriteLine` e mostra un testo comprensibile con `MessageBox.Show`. In caso di successo chiama `AlertBox.Show("Salvato.", MessageBoxIcon.Information)`.

- ✓ Scrivi il README: un breve paragrafo per livello (campo, modello, salvataggio), una tabella di casi di test con i casi obbligatorio, non valido, valido e valore limite per ogni regola, e la checklist del capstone. Poi prova la demo finché non riesci a eseguirla in cinque minuti.

Il capstone viene valutato rispetto a dieci funzionalità obbligatorie: nome del cliente ed email obbligatori; un `ErrorProvider` con un riepilogo visibile; l'extender Validation con almeno tre regole integrate; una classe `ValidationRule` personalizzata; un servizio di validazione usato da Save; validazione del modello con DataAnnotations o `IDataErrorInfo`; la validazione delle celle di email ed età nella griglia; un gestore `DataError` della griglia; il riepilogo con l' `IErrorProvider` personalizzato; e un breve README. La demo segue un ordine fisso: mostra un campo non valido, mostra Save fermato da `ValidateChildren`, mostra un errore del modello che compare dopo `EndEdit`, mostra la validazione delle celle della griglia, mostra il provider di riepilogo che elenca tutto, poi correggi i dati e salva con successo. Prima di consegnare, confronta le tue schermate con le dieci regole di UX della lezione: messaggi vicino al campo e in un riepilogo, linguaggio semplice che dica che cosa fare, nessuna colpa all'utente, nessuno stack trace, nessun salvataggio non valido, Cancel e Close mai bloccati senza motivo, nessuna verifica remota a ogni tasto premuto, le stesse regole del modello ovunque e tutti e quattro i tipi di valore di test.

## Deliverable richiesti

- ✓ Pipeline di salvataggio che esegue EndEdit, ValidateChildren e le verifiche di modello, di business e della griglia prima di salvare
- ✓ Classe SummaryErrorProvider che implementa SetError e GetError di IErrorProvider
- ✓ Metodo helper SetError che scrive sia nell'ErrorProvider a icona sia nel provider di riepilogo
- ✓ try/catch attorno al salvataggio che separa l'errore di nome duplicato dagli errori di sistema
- ✓ README con il design della validazione e i casi di test obbligatorio, non valido, valido e valore limite

## Domande di revisione per lo studente

- ✓ Un tester digita una nuova email nel form e fa subito clic su Save; viene salvata la vecchia email. Quale fase della pipeline manca o è nel posto sbagliato, e perché la sua posizione è importante?
- ✓ Il tuo `SummaryErrorProvider` mostra due volte "L'email è obbligatoria." dopo che l'utente corregge il campo e poi lo rende di nuovo non valido. Quale parte di `SetError` dovrebbe impedirlo, e da che cosa dipende?
- ✓ Quando la rete cade, il repository genera un'eccezione con un messaggio del database. Descrivi passo per passo che cosa vede l'utente e che cosa può trovare lo sviluppatore in seguito, e spiega perché questo caso è diverso da quello del nome duplicato.



### **La regola dei livelli di validazione**

Dai un nome al livello prima di scrivere la verifica. La validazione dei campi protegge la modifica e viene eseguita quando l'utente lascia un controllo; la validazione del modello protegge i dati ed esegue le stesse regole dal form, dalla griglia e da qualsiasi servizio; la validazione al salvataggio protegge il database ed esegue di nuovo tutti i livelli prima che venga scritto qualsiasi dato. Mostra ogni errore vicino al suo campo e in un riepilogo visibile, con un linguaggio semplice che dica che cosa fare, e tieni gli errori correggibili dall'utente separati dagli errori di sistema.