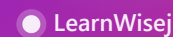


Validazione con data binding ed errori del modello

Guida della lezione · Guida al lab / esame



Guida della lezione

Validare il modello, non solo i controlli: BindingSource ed EndEdit, DataAnnotations eseguite con Validator.TryValidateObject, la mappatura dei MemberNames sui controlli, IDataErrorInfo con un ErrorProvider con data binding, BindToDataAndErrors e l'ordine sicuro del salvataggio. 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

Validare il modello, non solo i controlli: BindingSource ed EndEdit, DataAnnotations eseguite con Validator.TryValidateObject, la mappatura dei MemberNames sui controlli, IDataErrorInfo con un ErrorProvider con data binding, BindToDataAndErrors e l'ordine sicuro del salvataggio.

1. Il problema in produzione

Tutto ciò che ValidationClinic verifica finora vive nel form di inserimento: qui un gestore `Validating`, là una regola sull'extender `Validation`, una chiamata a `ContactValidator` in `btnSave_Click`. Ma un contatto non arriva solo da quel form. Può essere modificato nella griglia dei contatti, importato da un file o creato da un servizio, e nessuno di questi percorsi esegue i gestori del form. Questo modulo sposta le regole di base sul `ContactEditModel` stesso, così che i dati portino con sé la propria definizione di "valido" e ogni schermata possa porre loro la stessa domanda.

2. Concetti chiave

Concetto	Perché è importante
BindingSource	Un componente che si colloca tra i controlli e il tuo oggetto: il suo <code>DataSource</code> è il modello, e ogni controllo si collega a una proprietà attraverso di esso invece di essere copiato a mano.
EndEdit()	Trasferisce nel modello il valore che l'utente sta ancora modificando nel controllo, così la validazione legge ciò che è a schermo e non l'ultimo valore confermato.
DataAnnotations	Attributi come <code>[Required]</code> , <code>[StringLength]</code> , <code>[EmailAddress]</code> e <code>[Range]</code> applicati alle proprietà del modello, ciascuno con un <code>ErrorMessage</code> scritto per l'utente.
Validator.TryValidateObject	Il metodo .NET che esegue quegli attributi su un oggetto e riempie una lista di oggetti <code>ValidationResult</code> ; non sa nulla di controlli né di Wisej.NET.
MemberNames	I nomi delle proprietà a cui appartiene un <code>ValidationResult</code> , che sono l'unico collegamento tra un messaggio di errore e il controllo che dovrebbe mostrarlo.
IDataErrorInfo	Un'interfaccia che permette a un modello di rispondere a "che cosa non va in questa proprietà?" tramite un indicizzatore, così il livello di binding può chiedere gli errori invece di essere il tuo codice a spingerli.
ErrorProvider con data binding	Un <code>ErrorProvider</code> il cui <code>DataSource</code> è la stessa binding source dei controlli, così mostra gli errori del modello accanto ai controlli collegati senza alcuna chiamata a <code>SetError</code> .

3. Il filo della lezione

Collega il form, poi valida il modello

Il data binding collega una proprietà di un controllo a una proprietà del modello. Crea un `BindingSource`, imposta il suo `DataSource` su un `ContactEditModel` e chiama `txtName.DataBindings.Add("Text", contactBindingSource, nameof(ContactEditModel.Name))` per ogni campo. Da quel momento il form non copia più il testo in un oggetto quando si fa clic su Save; il modello sono i dati del form, e `contactBindingSource.Current` te lo restituisce.

C'è una lacuna in cui un principiante cade quasi sempre. Il binding non aggiorna il modello a ogni tasto premuto. Se l'utente digita un'email e fa subito clic su Save, il nuovo testo può trovarsi ancora nella TextBox mentre `Email` sul modello contiene il valore precedente. Validare il modello in quel momento significa verificare dati che l'utente ha già sostituito. La correzione è una sola chiamata: `contactBindingSource.EndEdit()` conclude la modifica in corso e trasferisce nel modello i valori digitati. Rendila la prima riga di ogni percorso di salvataggio, prima che qualsiasi regola legga il modello.

DataAnnotations e Validator.TryValidateObject

Le DataAnnotations mettono le regole semplici proprio accanto ai dati che proteggono. Su `ContactEditModel`, `Name` riceve `[Required]` e `[StringLength(80)]`, `Email` riceve `[Required]` ed

`[EmailAddress]`, e `Age` riceve `[Range(0, 120)]`. Ogni attributo ha un `ErrorMessage` come "Inserisci un indirizzo email valido." Tratta quei messaggi come testo di prodotto che l'utente leggerà, non come note per sviluppatori.

Gli attributi da soli non fanno nulla; qualcosa deve eseguirli. Crea una `List<ValidationResult>` e un `ValidationContext` per il modello, poi chiama `Validator.TryValidateObject(model, context, results, validateAllProperties: true)`. Restituisce `false` quando qualcosa non è andato a buon fine e aggiunge un risultato per ogni regola violata. Non tralasciare l'ultimo argomento: senza `validateAllProperties: true` viene verificato solo `[Required]`, e un'email malformata o un'età di 300 passano senza dire nulla. Nota anche che `[EmailAddress]` e `[Range]` considerano valido un valore null, ed è proprio per questo che la regola dell'obbligatorietà è un attributo separato.

IDataErrorInfo e l'ErrorProvider con data binding

`IDataErrorInfo` ribalta il rapporto. Invece di essere il tuo codice a eseguire un validatore e a spingere i messaggi sui controlli, è il modello a rispondere alle domande. Ha due membri: `Error`, un messaggio sull'oggetto nel suo insieme (spesso una stringa vuota), e l'indicizzatore `this[string columnName]`, che restituisce il messaggio per una proprietà oppure una stringa vuota quando quella proprietà è corretta. Nel secondo modello del lab, l'indicizzatore restituisce "Il nome è obbligatorio." quando `columnName` è `Name` e il nome è vuoto.

L'interfaccia diventa utile quando un `ErrorProvider` è collegato agli stessi dati. Imposta `errorProvider.ContainerControl` sul form ed `errorProvider.DataSource` su `contactBindingSource`. Il provider chiede allora all'elemento corrente l'errore di ogni proprietà collegata e mostra ciascun messaggio accanto al controllo collegato a quella proprietà. Nel tuo codice non compare nessuna chiamata a `SetError`. .NET offre anche `INotifyDataErrorInfo`, che può segnalare più errori per proprietà e cambiarli in modo asincrono, ma per un form da principiante `IDataErrorInfo` è il punto di partenza più semplice e ben supportato.

BindToDataAndErrors a runtime

Un provider con data binding ha bisogno che due impostazioni siano coerenti: `DataSource`, l'oggetto o la lista, e `DataMember`, il percorso al suo interno. Nel designer le imposti entrambe una volta e non le tocchi più. A runtime, per esempio quando `ValidationClinic` passa l'editor dal modello annotato al modello `IDataErrorInfo`, impostarle una dopo l'altra lascia un istante in cui la nuova sorgente è abbinata al vecchio membro, e il provider può tentare di collegarsi a qualcosa che non esiste.

`BindToDataAndErrors(source, member)` le imposta entrambe con una sola chiamata, così il provider vede sempre e solo una coppia coerente. Quando i dati sono l'intero oggetto, passa `string.Empty` come membro: `errorProvider.BindToDataAndErrors(contactBindingSource, string.Empty)`. Usa le due proprietà nel designer, e il metodo ogni volta che la sorgente cambia mentre il form è aperto.

Mappa gli errori del modello sui controlli e sul riepilogo

`Validator.TryValidateObject` non sa volutamente nulla di Wisej.NET, quindi è il form a dover tradurre i suoi risultati. Costruisci una mappa dei campi nel costruttore, un `Dictionary<string, Control>` da `nameof(ContactEditModel.Email)` a `txtEmail` e così via, poi scorri i risultati: per ogni nome in `result.MemberNames`, cerca il controllo e chiama `errorProvider.SetError(control, result.ErrorMessage)`. Infine unisci tutti i messaggi in `validationSummaryLabel` e mostralo, perché

un'icona da sola passa facilmente inosservata. Un risultato il cui membro non è nella mappa arriva comunque al riepilogo, così nessun errore va mai perso. La mappa ha la stessa forma di

`ApplyModelValidation` del Modulo 4, ed è per questo che i due possono condividere un unico riepilogo.

Con tutti i livelli al loro posto, il gestore del salvataggio segue un ordine fisso: concludi la modifica, valida i controlli figli, valida il modello, esegui le regole tra campi, salva solo se nulla è fallito e intercetta a parte le eccezioni di sistema, così un'interruzione del database non sembra mai un errore di digitazione.

```
private void btnSave_Click(object sender, EventArgs e)
{
    contactBindingSource.EndEdit();           // 1. valori digitati nel modello
    if (!ValidateChildren()) return;          // 2. validazione dei campi
    var model = (ContactEditModel)contactBindingSource.Current;
    if (!ApplyAnnotationErrors(model)) return; // 3. DataAnnotations
    if (!ApplyModelValidation(model)) return;  // 4. ContactValidator, regole tra campi
    try { _repository.Save(model); ShowSaved(); } // 5. solo ora
    catch (Exception ex) { ShowSystemError(ex); } // 6. non un messaggio di validazione
}
```



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:

- ✓ Collegare i controlli a un modello e validare il modello prima del salvataggio.
- ✓ Usare le `DataAnnotations` con `Validator.TryValidateObject`.
- ✓ Comprendere `IDataErrorInfo` e il comportamento di un `ErrorProvider` con data binding.
- ✓ Usare `BindToDataAndErrors` quando cambi a runtime la sorgente dati di un provider.
- ✓ Mappare gli errori del modello sui controlli Wisej.NET e sui riepiloghi.

Guida al lab / esame

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



Lab pratico

Collega il form di inserimento di ValidationClinic a un ContactEditModel tramite un contactBindingSource e aggiungi le annotazioni Required, StringLength, EmailAddress e Range alle sue proprietà Name, Email e Age. In btnSave_Click chiama per prima cosa contactBindingSource.EndEdit(), esegui Validator.TryValidateObject con validateAllProperties impostato a true e mappa il nome del membro di ogni ValidationResult sul suo controllo tramite una mappa dei campi e il riepilogo di validazione. Poi implementa IDataErrorInfo su un secondo modello, punta l'ErrorProvider alla binding source con BindToDataAndErrors e mostra gli stessi errori che compaiono tramite data binding senza una chiamata a SetError scritta a mano.

Punto di partenza

ValidationClinic dopo il Modulo 4: il form di inserimento ha regole di campo, un servizio

`ContactValidator`, `ApplyModelValidation` e una `MinimumAgeValidationRule`, ma i suoi controlli non sono collegati a nulla, quindi `btnSave_Click` legge ancora i valori direttamente da `txtName`, `txtEmail` e `txtAge`.

Approccio suggerito

- ✓ Crea `ContactEditModel` con `Name`, `Email` e un `Age` nullable, aggiungi le quattro annotazioni con testi `ErrorMessage` in linguaggio semplice e collega a esso le tre TextBox con `DataBindings.Add("Text", contactBindingSource, ...)`.
- ✓ Prima di scrivere qualsiasi validazione, dimostra la lacuna della modifica: digita un'email, fai clic su Save e scrivi nel log `((ContactEditModel)contactBindingSource.Current).Email` con e senza una chiamata a `contactBindingSource.EndEdit()` prima.
- ✓ Scrivi `ValidateModel` attorno a `Validator.TryValidateObject(model, context, results, validateAllProperties: true)`, poi costruisci `_fieldMap` nel costruttore e un metodo `ApplyAnnotationErrors` che imposta l'errore di ogni controllo e riempie `validationSummaryLabel`.
- ✓ Riscrivi `btnSave_Click` nell'ordine sicuro: `EndEdit`, `ValidateChildren`, annotazioni, il `ContactValidator` del Modulo 4, salvataggio e un `catch` separato che mostra un errore di sistema comprensibile.
- ✓ Aggiungi un secondo modello che implementa `IDataErrorInfo`, passa la binding source a quel modello a runtime, chiama `errorProvider.BindToDataAndErrors(contactBindingSource, string.Empty)` e svuota il nome per vedere comparire l'errore senza alcuna `SetError`.

Deliverable richiesti

- ✓ ContactEditModel con DataAnnotations collegato al form tramite un BindingSource
- ✓ Gestore del salvataggio che chiama EndEdit prima di eseguire Validator.TryValidateObject

- ✓ Mappa dei campi che trasforma i nomi dei membri dei ValidationResult in errori dei controlli e righe del riepilogo
- ✓ Modello IDataErrorInfo i cui errori compaiono tramite un ErrorProvider con data binding
- ✓ Chiamata a BindToDataAndErrors usata quando la sorgente dati del provider cambia a runtime

Domande di revisione per lo studente

- ✓ Un tester digita una nuova email e preme Invio per salvare, e a essere verificata e memorizzata è la vecchia email valida. Quale riga manca, e perché il form a schermo sembra corretto mentre il modello è sbagliato?
- ✓ Rimuovi `validateAllProperties: true` e ogni salvataggio "funziona" ancora. Quali delle tue quattro annotazioni hanno smesso di essere eseguite, e quali dati errati possono ora raggiungere il repository?
- ✓ Il servizio di importazione crea contatti senza aprire il form. Quali regole di questo modulo proteggono ancora quel percorso, e quali vengono eseguite solo perché un utente ha fatto clic su Save?



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.