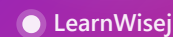


Regole di validazione integrate e l'extender Validation

Guida della lezione · Guida al lab / esame



Guida della lezione

L'extender Validation di Wisej.NET: la proprietà estesa ValidationRules in fase di progettazione, SetValidationRules a runtime, le regole Required, Email, Integer, Decimal, Currency, Telephone e Regex, l'ordine delle regole che si ferma al primo errore e le destinazioni di visualizzazione ErrorProvider o Label. 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

L'extender Validation di Wisej.NET: la proprietà estesa ValidationRules in fase di progettazione, SetValidationRules a runtime, le regole Required, Email, Integer, Decimal, Currency, Telephone e Regex, l'ordine delle regole che si ferma al primo errore e le destinazioni di visualizzazione ErrorProvider o Label.

1. Il problema in produzione

Alla fine del Modulo 2 il form di accettazione di ValidationClinic funziona, ma guarda quanto codice ha richiesto: un helper per il nome, un helper per l'email, un helper per il telefono, e ognuno ripete le stesse verifiche "è vuoto, ha la forma giusta". Aggiungi età, fido e codice cliente e il form si arricchisce di un'altra manciata di metodi quasi identici. Le verifiche non sono sbagliate; vengono solo riscritte su ogni campo e su ogni form. Questo modulo sostituisce quella ripetizione con l'extender `Validation`, in cui descrivi come deve essere un campo valido e Wisej.NET esegue le verifiche al posto tuo.

2. Concetti chiave

Concetto	Perché è importante
extender <code>Validation</code>	Un componente non visuale che aggiungi a un form; si aggancia agli eventi <code>Validating</code> e <code>Validated</code> di ogni controllo che ha delle regole, così non scrivi più un gestore per ogni campo.
proprietà estesa	Una proprietà che un componente aggiunge ad altri controlli; quando l'extender è sul form, ogni controllo compatibile mostra una voce <code>ValidationRules</code> nella finestra Properties.
<code>SetValidationRules</code>	L'equivalente in codice del designer: <code>validation.SetValidationRules(txtEmail, new ValidationRule[] { ... })</code> assegna a un controllo il suo elenco di regole a runtime.
regole integrate	Verifiche già pronte per l'input più comune: <code>RequiredValidationRule</code> , <code>EmailValidationRule</code> , <code>IntegerValidationRule</code> , <code>DecimalValidationRule</code> , <code>CurrencyValidationRule</code> , <code>TelephoneValidationRule</code> e <code>RegexValidationRule</code> .
ordine delle regole	Le regole vengono eseguite nell'ordine in cui le elenchi; la prima che fallisce ferma le altre e annulla l'evento <code>Validating</code> del controllo.
destinazione di visualizzazione	Dove compare l' <code>InvalidMessage</code> della regola che fallisce: sul controllo stesso, tramite un'icona di <code>ErrorProvider</code> oppure come testo in una Label, tutte raggiunte tramite <code>validation.ErrorProvider</code> .
helper di configurazione delle regole	Un piccolo metodo che costruisce gli array di regole per i campi comuni, così un secondo form ottiene le stesse verifiche e la stessa formulazione semplicemente chiamandolo.

3. Il filo della lezione

Un componente, due modi per configurarlo

L'extender `Validation` è un componente, come un timer o l' `ErrorProvider` del Modulo 2: non ha una superficie visibile propria. Trascinalo dalla Toolbox sul form di accettazione nel designer di Visual Studio e compare nella barra dei componenti sotto il form. Da quel momento ogni controllo compatibile del form ottiene nella finestra Properties una nuova voce chiamata `ValidationRules`. Seleziona `txtEmail`, apri quell'editor di collezione e aggiungi le regole di cui il campo ha bisogno. Il designer le scrive nel file designer del form, così chiunque apra il form vede quali campi sono obbligatori senza leggere un solo gestore di eventi.

La stessa cosa si può fare in codice, ed è quello che usa il lab. Crei il componente, punti la sua `ContainerControl` al form e chiami `SetValidationRules` una volta per controllo, di solito da un metodo `ConfigureValidationRules` chiamato nel costruttore del form. Entrambe le strade producono lo stesso risultato: l'extender si aggancia agli eventi di validazione del controllo ed esegue le regole quando l'utente lascia il campo, esattamente il momento che nel Modulo 1 hai imparato a gestire a mano.

```

errorProvider.ContainerControl = this;
validation.ContainerControl = this;
validation.ErrorProvider = errorProvider;

validation.SetValidationRules(txtEmail, new ValidationRule[]
{
    new RequiredValidationRule { InvalidMessage = "L'email è obbligatoria.", Trim = true },
    new EmailValidationRule { InvalidMessage = "Inserisci un indirizzo email valido." }
});
validation.SetValidationRules(txtAge, new ValidationRule[]
{
    new RequiredValidationRule { InvalidMessage = "L'età è obbligatoria." },
    new IntegerValidationRule { InvalidMessage = "L'età deve essere un numero intero." }
});

```

Le sette regole integrate

Ogni regola risponde a una domanda sul testo di un controllo. `RequiredValidationRule` chiede "c'è qualcosa qui?" e ha tre proprietà utili: `Required` attiva la verifica, `Trim` rimuove prima gli spazi così un campo che contiene solo spazi conta comunque come vuoto, e `MinimumLength` richiede almeno un certo numero di caratteri. `EmailValidationRule` verifica che il valore abbia l'aspetto di un indirizzo email. `IntegerValidationRule` accetta numeri interi come un'età, mentre `DecimalValidationRule` accetta numeri con una parte decimale. `CurrencyValidationRule` verifica un importo come il fido e può formattare il valore una volta accettato. `TelephoneValidationRule` verifica un numero di telefono rispetto a una maschera, e `RegexValidationRule` confronta il valore con un'espressione regolare, che è il modo in cui ValidationClinic impone un codice cliente formato dalle lettere CUS, un trattino e quattro cifre.

Ogni regola ha un `InvalidMessage`. Quel testo è ciò che l'utente legge, quindi scrivilo come testo di prodotto che dice che cosa fare: "L'età deve essere un numero intero." invece di "Integer parse failed".

L'ordine delle regole e il primo errore

Le regole di un controllo vengono eseguite dalla prima all'ultima. Appena una fallisce, le regole restanti vengono saltate e l'evento `Validating` del controllo viene annullato, con lo stesso effetto che nel Modulo 1 ottenevi impostando tu `e.Cancel = true`. Viene mostrato un solo messaggio: quello della prima regola che ha fallito.

Ecco perché l'ordine è una scelta di progettazione. Metti `RequiredValidationRule` per prima su ogni campo obbligatorio. Lascia vuota la casella dell'email e l'utente legge "L'email è obbligatoria.", che è il vero problema. Se la regola Email venisse prima, la stessa casella vuota direbbe "Inserisci un indirizzo email valido.", che è tecnicamente vero e molto meno utile. Nel lab lo dimostrerai con un `txtEmail` vuoto: Required fallisce, Email non viene mai eseguita e il focus si comporta secondo la modalità `AutoValidate` che hai scelto in precedenza. I campi facoltativi, come il fido, iniziano semplicemente con la loro regola di formato.

Scegliere dove comparire il messaggio

Senza una destinazione di visualizzazione, una regola che fallisce riscrive il suo messaggio sul controllo stesso come tooltip di valore non valido. Assegna un `ErrorProvider` a `validation.ErrorProvider` e l'extender chiama invece `SetError` su di esso, dandoti l'icona e il tooltip del Modulo 2 senza nessuna

chiamata a `SetError` nel tuo codice. Anche una `Label` può fare da destinazione: mostra il messaggio come testo semplice e si nasconde quando non c'è niente da dire, il che è adatto a un form breve o a un campo che ha bisogno di testo visibile invece di un'icona. Qualsiasi classe che implementa `IErrorProvider` può svolgere questo ruolo; il Modulo 7 ne costruisce una che alimenta un pannello di riepilogo.

L'extender genera anche i propri eventi `Validating` e `Validated` per ogni controllo che ha delle regole. Questo ti dà un unico punto centrale in cui aggiornare `validationSummaryLabel`, così il riepilogo visibile del Modulo 2 resta allineato alle icone anche se le verifiche non le scrivi più tu.

Configurazioni di regole riutilizzabili

Email, età e telefono compaiono in più di una schermata di `ValidationClinic`. Invece di copiare gli array, mettili in un piccolo helper, per esempio una classe statica `ClinicRules` con metodi come `RequiredEmail()` e `RequiredAge()` che restituiscono ciascuno un nuovo `ValidationRule[]`. Il form di accettazione e qualsiasi secondo form chiamano gli stessi metodi, così l'ordine e la formulazione di ogni messaggio vengono decisi una volta sola.

Limita l'helper alla forma dei campi. Le regole che confrontano due campi, cercano qualcosa in un database o esprimono una policy di business non appartengono all'extender, perché verrebbero eseguite solo quando un controllo perde il focus. Il Modulo 4 le sposta in metodi di validazione, in un servizio di validazione e in classi `ValidationRule` personalizzate.



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:

- ✓ Usare l'extender `Validation` di `Wisej.NET` in fase di progettazione e a runtime.
- ✓ Applicare le regole `Required`, `Email`, `Integer`, `Decimal`, `Currency`, `Telephone` e `Regex`.
- ✓ Spiegare l'ordine delle regole e perché la validazione si ferma al primo errore.
- ✓ Assegnare un `ErrorProvider` o una `Label` per visualizzare i messaggi di validazione.
- ✓ Creare configurazioni di regole riutilizzabili per i form più comuni.

Guida al lab / esame

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



Lab pratico

Sostituisci i gestori dei campi scritti a mano in ValidationClinic con l'extender Validation. Aggiungi un componente Validation e un ErrorProvider al form di accettazione, imposta ContainerControl e validation.ErrorProvider, poi chiama SetValidationRules in un metodo ConfigureValidationRules: Required e poi Email su txtEmail, Required e poi Integer su txtAge, Required e poi Telephone su txtPhone, Currency su txtCreditLimit e una regola Regex per il codice cliente. Metti Required per prima ovunque così vince il messaggio più utile, aggancia l'evento Validating dell'extender per aggiornare il riepilogo e sposta gli array di regole in un piccolo helper riutilizzabile così un altro form può applicare la stessa configurazione.

Punto di partenza

Apri ValidationClinic come l'ha lasciato il Modulo 2: il form di accettazione ha `txtName`, `txtEmail`, `txtPhone` e `txtAge`, un `ErrorProvider`, gli helper `ValidateName`, `ValidateEmail` e `ValidatePhone` e una `validationSummaryLabel`. Per questo lab aggiungi un TextBox `txtCreditLimit` e un TextBox `txtCustomerCode` se non sono ancora sul form.

Approccio suggerito

- ✓ Aggiungi un componente `Validation` al form di accettazione nel designer, seleziona `txtEmail` e aggiungi una regola Required e una regola Email tramite l'editor di collezione `ValidationRules`, poi esegui il form una volta per vedere l'extender in azione prima di scrivere codice.
- ✓ Rimuovi quelle regole del designer e scrivi `ConfigureValidationRules`, chiamato dal costruttore: imposta `errorProvider.ContainerControl`, `validation.ContainerControl` e `validation.ErrorProvider`, poi chiama `SetValidationRules` per email, età, telefono, fido e codice cliente.
- ✓ Dai a ogni campo obbligatorio `RequiredValidationRule` come prima regola, con `Trim = true` dove gli spazi non devono contare, e scrivi ogni `InvalidMessage` come una frase che dice all'utente che cosa digitare.
- ✓ Gestisci gli eventi `Validating` e `Validated` dell'extender in un unico punto e ricostruisci `validationSummaryLabel` dagli errori correnti, nascondendola quando non ne restano.
- ✓ Sposta gli array di regole in un helper statico `ClinicRules` i cui metodi restituiscono un nuovo `ValidationRule[]`, chiamalo dal form di accettazione, poi prova un'email vuota, un'email malformata, un'email valida, un'età non numerica e un codice cliente errato.

Deliverable richiesti

- ✓ Componente Validation collegato al form e a un ErrorProvider come destinazione di visualizzazione
- ✓ Metodo ConfigureValidationRules che chiama SetValidationRules per email, età, telefono, fido e codice cliente

- ✓ Regola Required posta per prima su ogni campo obbligatorio, con testo InvalidMessage in linguaggio semplice
- ✓ Gestore Validating dell'extender che mantiene il riepilogo di validazione allineato alle regole
- ✓ Helper di configurazione delle regole riutilizzabile che un secondo form può chiamare

Domande di revisione per lo studente

- ✓ Inverti l'ordine così la regola Email viene eseguita prima di Required su `txtEmail`. Descrivi esattamente che cosa vede un utente dopo aver lasciato il campo vuoto e perché quel messaggio è meno utile.
- ✓ Sia gli helper del Modulo 2 sia l'extender finiscono con un'icona accanto al campo. Che cosa hai smesso di scrivere a mano e che cosa hai dovuto comunque decidere tu?
- ✓ Un collega vuole aggiungere "la data di fine deve essere successiva alla data di inizio" a `ClinicRules`. Perché l'extender è il posto sbagliato per quella verifica, e quando non verrebbe eseguita?



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.