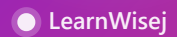


Programmatische Validierung und Feedback mit ErrorProvider

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Ein ErrorProvider pro Formular, SetLastError und das Zurücksetzen von Fehlern pro Control, InvalidMessage für einfaches lokales Feedback, ein kleiner FieldValidation-Ergebnistyp, der Regeln aus der UI heraushält, und eine sichtbare Validierungszusammenfassung, damit kein Fehler davon abhängt, dass jemand mit der Maus über ein Icon fährt. Was das für Entwickler bedeutet, die grundlegendes C# beherrschen und mit der Formularentwicklung in Wisej.NET beginnen – und wie Sie dieses Modul am besten durcharbeiten.



Fokus: Produktivbetrieb

Ein ErrorProvider pro Formular, SetLastError und das Zurücksetzen von Fehlern pro Control, InvalidMessage für einfaches lokales Feedback, ein kleiner FieldValidation-Ergebnistyp, der Regeln aus der UI heraushält, und eine sichtbare Validierungszusammenfassung, damit kein Fehler davon abhängt, dass jemand mit der Maus über ein Icon fährt.

1. Das Problem im Produktivbetrieb

Modul 1 hat dem Aufnahmeformular von ValidationClinic zwei **Validating**-Handler gegeben, und für zwei Felder funktioniert das. Ein echtes Kontaktformular hat einen Namen, eine E-Mail-Adresse, eine Telefonnummer, eine Kundennummer und mehr, und wenn jedes Feld seinen eigenen handgeschriebenen Handler mit seiner eigenen Art bekommt, eine Meldung anzuzeigen, wird das Formular schnell uneinheitlich: Ein Feld zeigt ein Icon, ein anderes gar nichts, ein drittes lässt einen alten Fehler stehen, obwohl der Benutzer ihn schon korrigiert hat. Dieses Modul ersetzt diesen Wildwuchs durch ein einziges Muster: Kleine Hilfsmethoden entscheiden, was falsch ist, eine Methode zeigt das Ergebnis an, und eine sichtbare Zusammenfassung sagt jedem Benutzer, was zu korrigieren ist, ohne dass er mit der Maus über etwas fahren muss.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
<code>ErrorProvider</code>	Eine Extender-Komponente, die neben jedem Control, dem Sie eine Meldung geben, ein Fehler-Icon zeichnet und diese Meldung als Tooltip anzeigt; eine Instanz bedient das gesamte Formular.
<code>ContainerControl</code>	Das Formular oder User-Control, zu dem der Provider gehört; setzen Sie es auf <code>this</code> , weiß der Provider, auf welcher Oberfläche er seine Icons zeichnet.
<code>SetError(control, message)</code>	Zeigt <code>message</code> neben <code>control</code> an; ein Aufruf mit einem leeren String oder <code>null</code> entfernt den Fehler nur für dieses eine Control.
<code>GetError</code> , <code>HasErrors</code> , <code>Clear()</code>	Die aktuelle Meldung eines Controls lesen, abfragen, ob im Formular noch ein Fehler besteht, oder alle Fehler auf einmal entfernen, wenn das gesamte Formular zurückgesetzt wird.
<code>InvalidMessage</code>	Eine Eigenschaft an Editor-Controls, die das Interface <code>IValidation</code> unterstützen; das Control zeigt die Meldung selbst als Ungültig-Tooltip an, ohne dass ein separater Provider beteiligt ist.
<code>FieldValidation</code> -Record	Ein winziger Ergebnistyp mit Control, Feldname und Meldung, wobei eine leere Meldung „gültig“ bedeutet; damit liefern Regeln Antworten, statt den Bildschirm zu verändern.
<code>validationSummaryLabel</code>	Ein einfaches Label, das jedes aktuelle Problem als Text auflistet und nur sichtbar ist, solange Fehler bestehen, damit Feedback nie davon abhängt, ein Icon zu sehen oder mit der Maus darüberzufahren.

3. Der rote Faden der Lektion

Feldfehler mit ErrorProvider anzeigen und entfernen

Legen Sie für das Formular einen `ErrorProvider` an, entweder indem Sie ihn aus der Toolbox in den Designer ziehen oder als Feld, das im Code erzeugt wird, und setzen Sie seinen `ContainerControl` auf das Formular. Zwei optionale Einstellungen machen ihn für Benutzer ruhiger: `BlinkStyle = ErrorBlinkStyle.NeverBlink` verhindert, dass das Icon blinkt, und `SetIconAlignment(txtEmail, ErrorIconAlignment.MiddleRight)` zusammen mit `SetIconPadding(txtEmail, 4)` platziert das Icon direkt rechts neben dem Feld, wo das Auge es erwartet.

Einen Fehler anzuzeigen ist ein einziger Aufruf: `errorProvider.SetError(txtName, "Name is required.")`. Denken Sie daran, dass Wisej.NET das auf dem Server ausführt; der Browser erhält einfach das aktualisierte Icon und den Tooltip. Entfernen ist derselbe Aufruf mit einem leeren String. Dieses Detail ist wichtiger, als es aussieht. Korrigiert der Benutzer den Namen, entfernen Sie den Fehler an `txtName` und sonst nichts. `Clear()` entfernt jeden Fehler im Formular; heben Sie es daher für die Momente auf, in denen das gesamte Formular wirklich neu beginnt: beim Laden eines anderen Kontakts, beim Zurücksetzen des Formulars oder beim Start eines vollständigen Validierungsdurchlaufs, der sofort jeden aktuellen Fehler

wieder setzt. `GetError(control)` sagt Ihnen, was ein einzelnes Control gerade anzeigt, und `HasErrors` ist eine schnelle letzte Prüfung, bevor Speichern fortfährt.

InvalidMessage oder SetError?

Wisej.NET bietet Ihnen einen zweiten, leichteren Weg für Feedback. Editor-Controls, die `IValidation` implementieren, haben eine Eigenschaft `InvalidMessage`. Setzen Sie sie im `Validating`-Handler des Controls, wenn der Wert falsch ist, setzen Sie sie auf einen leeren String zurück, wenn der Wert stimmt, und das Control zeigt die Meldung als eigenen Ungültig-Tooltip an. Kein Provider, keine Icon-Platzierung, nichts Gemeinsames.

Damit passt `InvalidMessage` gut zu einem sehr einfachen Formular oder zu einem einzelnen Feld, dessen Feedback lokal bleiben soll. `SetError` passt besser, wenn ein Formular mehrere Felder hat, denn dann sehen alle Fehler gleich aus, lassen sich mit `GetError` zurücklesen und mit `HasErrors` zählen. Im Lab verwenden Sie bewusst beides: Das Feld für die Kundennummer verwendet `InvalidMessage`, jedes andere Feld läuft über den `ErrorProvider`, und Sie schreiben eine kurze Notiz, die vergleicht, was der Benutzer sieht.

Ein Zusammenfassings-Label, das jeder lesen kann

Ein Fehler-Icon ist klein, es besteht nur aus einer Farbe und einer Form, und sein Text erscheint erst, wenn der Mauszeiger darauf ruht. Ein Tastaturbenutzer, ein Benutzer mit Touchscreen oder ein Benutzer mit eingeschränktem Sehvermögen kann es völlig übersehen. Deshalb zeigt jedes Formular in diesem Kurs seine Fehler zusätzlich als einfachen Text an. Die einfachste Variante ist ein Label namens `validationSummaryLabel` oberhalb des Speichern-Buttons, mit einer Zeile pro Problem, etwa *Email: Enter a valid email address*.

Zwei Regeln halten die Zusammenfassung ehrlich. Sie ist nur sichtbar, solange mindestens ein Fehler besteht, damit nie ein leerer roter Kasten stehen bleibt. Und sie wird aus denselben Ergebnissen neu aufgebaut, die auch die Icons setzen, sodass Icons und Zusammenfassung sich nie widersprechen können. Eine sichtbare Zusammenfassung macht das Formular außerdem leichter testbar: Sie können den Text des Labels prüfen, statt mit der Maus über Icons zu fahren.

Eine Hilfsmethode für mehrere Felder

Der entscheidende Schritt in diesem Modul ist, das Entscheiden vom Anzeigen zu trennen. Jede Regel wird zu einer kleinen Methode, `ValidateName`, `ValidateEmail` oder `ValidatePhone`, die einen Wert liest und einen `FieldValidation`-Record zurückgibt. Sie ruft nie `SetError` auf und fasst nie ein Label an. Eine einzige `Apply`-Methode nimmt dann alle Ergebnisse entgegen und erledigt die Anzeige an einer Stelle.

```

public sealed record FieldValidation(Control Control, string FieldName, string Message)
{
    public bool IsValid => string.IsNullOrEmpty(Message);
}

private bool Apply(params FieldValidation[] results)
{
    foreach (var result in results)
        errorProvider.SetError(result.Control, result.Message);
    validationSummaryLabel.Text = string.Join("\n", results
        .Where(r => !r.IsValid).Select(r => $"{r.FieldName}: {r.Message}"));
    validationSummaryLabel.Visible = results.Any(r => !r.IsValid);
    return results.All(r => r.IsValid);
}

```

Beachten Sie, dass `Apply` für jedes Ergebnis `SetError` aufruft, ob gültig oder nicht. Ein gültiges Ergebnis hat eine leere Meldung, und wenn Sie es übergeben, wird das Icon dieses Controls entfernt. Genau so entfernt das Korrigieren eines Feldes nur dessen eigenen Fehler: Die anderen Ergebnisse tragen weiterhin ihre Meldungen, also bleiben ihre Icons stehen. Der Speichern-Button ruft `Apply(ValidateName(), ValidateEmail(), ValidatePhone())` auf und kehrt vorzeitig zurück, wenn er `false` erhält.

Feld-Feedback und Feedback zu Geschäftsregeln

Nicht jede Meldung gehört neben ein Feld. *Feld-Feedback* betrifft den Wert in einem Control: Er ist leer, er enthält kein @, die Telefonnummer ist zu kurz. Der Benutzer kann das korrigieren, ohne dieses Control zu verlassen, also steht die Meldung neben dem Control und in der Zusammenfassung. *Feedback zu Geschäftsregeln* betrifft die Bedeutung der Daten: Ein Kontakt mit diesem Namen existiert bereits, oder zwei Datumsangaben passen nicht zusammen. Solche Regeln brauchen oft andere Felder oder gespeicherte Daten und laufen meist beim Speichern, nicht wenn der Benutzer ein Feld mit Tab verlässt.

Halten Sie beides in Ihrem Code und in Ihrer Formulierung auseinander. Feld-Hilfsmethoden bleiben klein und schnell. Eine Geschäftsregel kann mit `SetError` trotzdem auf das relevanteste Control zeigen, aber ihre Meldung erklärt die Regel in verständlicher Sprache und erscheint in der Zusammenfassung, damit der Benutzer versteht, warum das Speichern abgebrochen wurde. Keines von beiden ist eine Exception: Eine fehlerhafte E-Mail-Adresse ist ein erwarteter Fehler mit einer normalen Produktmeldung, während eine verlorene Datenbankverbindung ein Systemfehler ist, der getrennt behandelt wird. Spätere Module verschieben Geschäftsregeln in einen wiederverwendbaren Service; dieses Modul legt die Gewohnheit dafür an.



Gewohnheit: Validierungsebenen

Benennen Sie die Ebene, bevor Sie die Prüfung schreiben. Die Feldvalidierung schützt das Bearbeiten und läuft, wenn der Benutzer ein Control verlässt; die Modellvalidierung schützt die Daten und führt dieselben Regeln aus dem Formular, dem Grid und jedem Service aus; die Validierung beim Speichern schützt die Datenbank und führt jede Ebene noch einmal aus, bevor irgendetwas geschrieben wird. Zeigen Sie jeden Fehler nahe an seinem Feld und in einer sichtbaren Zusammenfassung, in verständlicher Sprache, die sagt, was als Nächstes zu tun ist, und halten Sie Fehler, die der Benutzer selbst korrigieren kann, getrennt von Systemfehlern.



Lernziele

Am Ende dieses Moduls können Sie:

- ✓ mit `ErrorProvider` Fehler auf Feldebene anzeigen und entfernen.
- ✓ den Unterschied zwischen `InvalidMessage` und `ErrorProvider.SetError` verstehen.
- ✓ ein Label für die Validierungszusammenfassung anlegen, das die Barrierefreiheit verbessert.
- ✓ mehrere Felder aus einer einzigen Hilfsmethode heraus validieren.
- ✓ Feld-Feedback von Feedback zu Geschäftsregeln trennen.

Lab- und Prüfungsleitfaden

Was Sie im praktischen Lab bauen, das vorgeschlagene Vorgehen und die geforderten Ergebnisse.



Praxis-Lab

Machen Sie aus dem Kontaktformular von `ValidationClinic` ein Formular mit programmatischer Validierung. Legen Sie für das Formular einen `ErrorProvider` mit gesetztem `ContainerControl`, `NeverBlink` und rechtsbündigem Icon an, schreiben Sie einen `FieldValidation`-Record sowie die Hilfsmethoden `ValidateName`, `ValidateEmail` und `ValidatePhone`, die eine Meldung zurückgeben, statt die UI anzufassen, und eine `Apply`-Methode, die für jedes Ergebnis `SetError` aufruft und ein `validationSummaryLabel` füllt, das nur sichtbar ist, solange Fehler bestehen. Verwenden Sie `InvalidMessage` am Feld für die Kundennummer, um die beiden Feedback-Varianten zu vergleichen, und stellen Sie sicher, dass das Korrigieren eines Feldes nur dessen eigenen Fehler entfernt.

Ausgangspunkt

Sie öffnen das Projekt `ValidationClinic`, das Sie in Modul 1 erstellt haben — oder, wenn Sie hier einsteigen, den Build zu Modul 1 aus dem [öffentlichen Beispiel-Repository](#). Das Formular zur Kundenaufnahme hat bereits `txtName`, `txtEmail`, einen Speichern- und einen Abbrechen-Button, dazu die `Validating`-Handler und den Schutz mit `ValidateChildren()`, die Sie beim letzten Mal geschrieben haben; in diesem Lab fügen Sie `txtPhone`, `txtCustomerCode` und ein Zusammenfassungs-Label hinzu und verschieben die Prüfungen in Hilfsmethoden.

Vorgeschlagenes Vorgehen

- ✓ Fügen Sie dem Formular einen `ErrorProvider` hinzu und setzen Sie in einer Methode `SetupValidation`, die aus dem Konstruktor aufgerufen wird, `ContainerControl = this`, `BlinkStyle = ErrorBlinkStyle.NeverBlink` sowie für jedes Feld `SetIconAlignment(..., ErrorIconAlignment.MiddleRight)` mit einem kleinen `SetIconPadding`.
- ✓ Deklarieren Sie den `FieldValidation`-Record und schreiben Sie dann `ValidateName`, `ValidateEmail` und `ValidatePhone` so, dass jede Methode den Wert trimmt, ihn prüft und einen Record mit einer verständlichen Meldung zurückgibt, oder mit einer leeren Meldung, wenn der Wert in Ordnung ist. Keine davon darf `SetError` aufrufen.
- ✓ Platzieren Sie ein Label namens `validationSummaryLabel` mit `Visible = false` oberhalb der Buttons und schreiben Sie `Apply(params FieldValidation[] results)` so, dass die Methode den Fehler jedes Controls setzt oder entfernt, das Label mit einer Zeile pro Problem füllt, das Label nur anzeigt, solange Probleme bestehen, und zurückgibt, ob alles gültig ist.
- ✓ Rufen Sie `Apply(ValidateName(), ValidateEmail(), ValidatePhone())` am Anfang von `btnSave_Click` auf und kehren Sie zurück, wenn das Ergebnis `false` ist. Lassen Sie den `Validating`-Handler jedes Feldes anhand des Ergebnisses seiner Hilfsmethode nur den Fehler seines eigenen Controls setzen oder entfernen, und führen Sie, solange die Zusammenfassung nach einem fehlgeschlagenen Speichern sichtbar ist, das vollständige `Apply` erneut aus, damit die Liste schrumpft, während Felder korrigiert werden.

- ✓ Verwenden Sie in `txtCustomerCode_Validating` `InvalidMessage` statt des Providers, starten Sie dann das Formular mit mehreren ungültigen Werten, korrigieren Sie ein Feld nach dem anderen und schreiben Sie eine kurze Notiz darüber, wie sich die beiden Feedback-Varianten für den Benutzer unterscheiden.

Geforderte Ergebnisse

- ✓ ErrorProvider auf Formularebene mit konfiguriertem ContainerControl, BlinkStyle und Icon-Ausrichtung
- ✓ FieldValidation-Record und die Hilfsmethoden ValidateName, ValidateEmail und ValidatePhone
- ✓ Apply-Methode, die den Fehler jedes Controls setzt oder entfernt und zurückgibt, ob alle Felder gültig sind
- ✓ validationSummaryLabel, das jedes aktuelle Problem auflistet und ausgeblendet ist, wenn es keine gibt
- ✓ Ein Feld, das InvalidMessage statt ErrorProvider verwendet, mit einer Notiz, die beide vergleicht

Wiederholungsfragen

- ✓ Der `txtName_Validating` -Handler eines Teamkollegen ruft `errorProvider.Clear()` auf, sobald der Name gültig wird. Was geht für einen Benutzer schief, der außerdem eine ungültige E-Mail-Adresse eingegeben hat, und was sollte der Handler stattdessen aufrufen?
- ✓ Das Feld für die Kundennummer zeigt seine Meldung über `InvalidMessage` an. Was müsste das Zusammenfassungs-Label über dieses Feld wissen, um es ebenfalls aufzulisten, und was sagt Ihnen das darüber, wann ein gemeinsamer Provider die bessere Wahl ist?
- ✓ Ein Benutzer gibt eine korrekt aufgebaute E-Mail-Adresse ein, die bereits zu einem anderen Kontakt gehört. Ist das Feld-Feedback oder Feedback zu einer Geschäftsregel, wann sollte die Prüfung laufen, und wo sollte die Meldung erscheinen?



Gewohnheit: Validierungsebenen

Benennen Sie die Ebene, bevor Sie die Prüfung schreiben. Die Feldvalidierung schützt das Bearbeiten und läuft, wenn der Benutzer ein Control verlässt; die Modellvalidierung schützt die Daten und führt dieselben Regeln aus dem Formular, dem Grid und jedem Service aus; die Validierung beim Speichern schützt die Datenbank und führt jede Ebene noch einmal aus, bevor irgendetwas geschrieben wird. Zeigen Sie jeden Fehler nahe an seinem Feld und in einer sichtbaren Zusammenfassung, in verständlicher Sprache, die sagt, was als Nächstes zu tun ist, und halten Sie Fehler, die der Benutzer selbst korrigieren kann, getrennt von Systemfehlern.