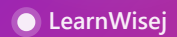


Validierungs-UX für den Produktivbetrieb, eigene Error-Provider und Abschlussprojekt

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Alle Ebenen zusammenführen: eine unspektakuläre, wiederholbare Speicher-Pipeline, ein eigener IErrorProvider, der ein Zusammenfassungs-Panel füllt, Icon- und Zusammenfassungs-Provider nebeneinander, zehn UX-Regeln für die Validierung und das ValidationClinic-Abschlussprojekt, das zeigt, wohin jede Prüfung gehört. 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

Alle Ebenen zusammenführen: eine unspektakuläre, wiederholbare Speicher-Pipeline, ein eigener IErrorProvider, der ein Zusammenfassungs-Panel füllt, Icon- und Zusammenfassungs-Provider nebeneinander, zehn UX-Regeln für die Validierung und das ValidationClinic-Abschlussprojekt, das zeigt, wohin jede Prüfung gehört.

1. Das Problem im Produktivbetrieb

Ein Benutzer, der das Aufnahmeformular von ValidationClinic ausfüllt, fragt nie, welche API einen Fehler gefunden hat; er merkt nur, ob die Maske ihm sagt, was falsch ist, und ob er fertig werden kann. Nach sechs Modulen hat die App Feld-Handler, Extender-Regeln, einen Validator-Service, annotierte Modelle und Grid-Regeln, aber jedes davon wurde für sich hinzugefügt, und noch garantiert nichts, dass bei jedem Klick auf Speichern alle laufen, und zwar in der richtigen Reihenfolge. Dieses Modul fügt die Ebenen zu einer vorhersehbaren Speicher-Pipeline und einer sichtbaren Zusammenfassung zusammen und lässt Sie das Design dann in einem Abschlussprojekt belegen.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
Speicher-Pipeline	Eine feste Abfolge in <code>btnSave_Click</code> , die offene Bearbeitungen übernimmt, jede Validierungsebene der Reihe nach ausführt und bei der ersten fehlschlagenden Ebene zurückkehrt, sodass nichts ungeprüft das Repository erreicht.
Frühe Rückkehr	Jede Stufe beendet den Handler mit <code>return</code> , wenn sie Fehler findet, sodass spätere Stufen nie mit Daten laufen, die eine frühere Stufe bereits abgelehnt hat.
<code>IErrorProvider</code>	Das kleine Interface, an das das Validierungssystem von Wisej.NET Meldungen schickt: <code>SetError(Control, string)</code> , um eine Meldung zu setzen oder zu entfernen, und <code>GetError(Control)</code> , um sie zurückzulesen.
<code>SummaryErrorProvider</code>	Ihr eigener <code>IErrorProvider</code> , der ein Dictionary aus Control-Meldungs-Paaren führt und daraus bei jedem Setzen oder Entfernen eines Fehlers ein Zusammenfassungs-Label neu schreibt.
Doppelte Provider	Eine einzige <code>SetError</code> -Hilfsmethode, die dieselbe Meldung in den integrierten <code>ErrorProvider</code> (das Icon neben dem Feld) und in den Zusammenfassungs-Provider (die Liste oben) schreibt.
Benutzerfehler und Systemfehler	Ein Fehler, den der Benutzer beheben kann, etwa ein doppelter Name, wird zur Feldmeldung; eine unerwartete Exception wird für Entwickler protokolliert und als ruhige, allgemeine Meldung angezeigt.
Grenzwertest	Ein Testwert genau an der Grenze einer Regel, etwa Alter 0 oder 120, der die Off-by-one-Fehler aufdeckt, die gewöhnliche gültige und ungültige Werte übersehen.

3. Der rote Faden der Lektion

Ein Speichern-Button, der absichtlich langweilig ist

Ein guter Speichern-Handler führt bei jedem Klick dieselben Schritte in derselben Reihenfolge aus und stoppt, sobald etwas falsch ist. In ValidationClinic lautet die Reihenfolge: die Zusammenfassung leeren, `bindingSource.EndEdit()` aufrufen, damit der noch in Eingabe befindliche Wert in das Modell übertragen wird, `ValidateChildren()` aufrufen, damit die Feldregeln jedes Controls laufen, die DataAnnotations-Prüfung mit `Validator.TryValidateObject` ausführen, `ContactValidator.Validate` für Geschäftsregeln ausführen, prüfen, dass keine Grid-Zeile mehr einen Fehler trägt, und erst dann speichern. `EndEdit` kommt zuerst, weil jede spätere Modellprüfung sonst veraltete Daten untersuchen würde.

```

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("Saved.", MessageBoxIcon.Information); }
catch (DuplicateNameException) { SetError(txtName, "A contact with this name already exists."); }

```

Die Hilfsmethoden hier (`ResetSummary` , `ApplyAnnotationErrors` , `GridHasErrors`) sind Ihre eigenen kleinen Methoden, keine Member des Frameworks. Das eigentliche Speichern steht in einem `try / catch` , und das abschließende `catch (Exception ex)` (nicht abgebildet) schreibt die Exception mit `System.Diagnostics.Debug.WriteLine` und zeigt einen verständlichen Text mit `MessageBox.Show` an. Die Pipeline ist absichtlich langweilig: Meldet ein Tester „Speichern hat eine fehlerhafte E-Mail-Adresse durchgelassen“, können Sie genau auf die Stufe zeigen, die sie hätte stoppen müssen.

Ein Zusammenfassungs-Panel über IErrorProvider

Dem Validierungssystem von Wisej.NET ist egal, was seine Meldungen anzeigt, solange das Ziel `IErrorProvider` implementiert. Der integrierte `ErrorProvider` ist eine Implementierung, und ein `Label` kann eine weitere sein. Ihr `SummaryErrorProvider` ist eine dritte: Sein Konstruktor erhält das Zusammenfassungs-Label, `SetError` entfernt den Eintrag des Controls, wenn die Meldung leer ist, und speichert ihn andernfalls, fügt dann die unterschiedlichen Meldungen zum `Text` des Labels zusammen und setzt `Visible` nur, solange mindestens ein Fehler besteht. `GetError` gibt die gespeicherte Meldung oder einen leeren String zurück.

Sie verbinden ihn mit zwei Zeilen: den Provider rund um `validationSummaryLabel` erzeugen und dann `validation.ErrorProvider = summaryProvider` zuweisen. Ab diesem Moment meldet jede Regel, die der Validation-Extender ausführt, einschließlich Ihrer `MinimumAgeValidationRule` aus Modul 4, in das Panel, ohne zusätzlichen Code in der Regel selbst.

Feld-, Grid- und Modellfehler an einem Ort

Fehler kommen jetzt aus drei Richtungen. Feldfehler stammen aus `Validating` -Handlern und Extender-Regeln. Modellfehler stammen aus den Member-Namen von `ValidationResult` und aus den Meldungen von `ContactValidator` , die Sie mit der Feldzuordnung aus Modul 5 wieder den Controls zuordnen. Grid-Fehler sitzen als `ErrorText` an Zellen und Zeilen und werden in `CellValidating` , `ValidateRow` und `DataError` gesetzt. Der Benutzer sollte nicht wissen müssen, welche dieser Quellen eine Meldung erzeugt hat.

Erreichen lässt sich das, indem Sie Formular- und Modellfehler über eine einzige Hilfsmethode `SetError(Control, string)` im Formular leiten, die `errorProvider.SetError` für das Icon und

`summaryProvider.SetError` für die Liste aufruft. Grid-Fehler bleiben im Grid, wo der Benutzer die rote Zelle sieht, und die Grid-Stufe der Pipeline fügt eine Zusammenfassungszeile wie „Row 3: Closed contacts require a closed date.“ hinzu, damit das Problem auch oben sichtbar ist. Das Entfernen funktioniert genauso in umgekehrter Richtung: Eine leere Meldung entfernt Icon und Zusammenfassungszeile gemeinsam.

Zehn Regeln für Validierung, mit der Menschen arbeiten können

Gute Validierungs-UX ist zum größten Teil Disziplin, nicht Code. Die zehn Regeln unten, zusammengefasst in sechs Zeilen, sind die Checkliste, an der das Abschlussprojekt gemessen wird:

- Setzen Sie jede Meldung neben ihr Feld und wiederholen Sie sie in einer Zusammenfassung für das gesamte Formular.
- Schreiben Sie in verständlicher Sprache, die sagt, was als Nächstes zu tun ist, und geben Sie nie dem Benutzer die Schuld.
- Zeigen Sie nie Stacktraces, SQL-Text oder Konvertierungs-Exceptions an.
- Speichern Sie nie ungültige Daten, und sperren Sie den Benutzer nie ein: Blockieren Sie Abbrechen oder Schließen nicht ohne echten Grund.
- Halten Sie langsame oder entfernte Prüfungen aus der Validierung pro Tastendruck heraus; führen Sie sie beim Speichern aus.
- Führen Sie dieselben Modellregeln aus dem Formular, dem Grid und jedem Service aus, und testen Sie Pflicht-, ungültige, gültige und Grenzwerte.

Ein Zusammenfassungs-Panel ist auch für die Barrierefreiheit wichtig: Ein Fehler, der nur erscheint, wenn jemand mit der Maus über ein Icon fährt, ist für Tastaturbenutzer unsichtbar und für alle anderen leicht zu übersehen. Eine Meldung, die ein Screenreader als Text erreichen kann, ist Teil des Designs, keine Dekoration.

Das ValidationClinic-Abschlussprojekt

Das Abschlussprojekt ist keine große Anwendung. Es ist der Nachweis, dass Sie wissen, wohin jede Prüfung gehört. ValidationClinic muss Folgendes haben: Name und E-Mail-Adresse als Pflichtfelder, einen `ErrorProvider` mit sichtbarer Zusammenfassung, den Validation-Extender mit mindestens drei integrierten Regeln, eine eigene `ValidationRule`-Klasse, einen Validator-Service, den Speichern verwendet, Modellvalidierung mit DataAnnotations oder `IDataErrorInfo`, Zellvalidierung für E-Mail und Alter sowie einen `DataError`-Handler im Grid, die Zusammenfassung über den eigenen `IErrorProvider` und ein kurzes README, das das Design erklärt.

Anschließend führen Sie es nach einem kurzen, festen Ablauf vor: ein ungültiges Feld verlassen, auf Speichern klicken und zeigen, wie `ValidateChildren` es stoppt, einen Modellfehler zeigen, der erst nach `EndEdit` erscheint, eine fehlerhafte E-Mail-Adresse in eine Grid-Zelle tippen, auf das Zusammenfassungs-Panel zeigen, das alles auflistet, dann jedes Problem korrigieren und erfolgreich speichern. Schreiben Sie zuerst die Testtabelle im README, mit Pflicht-, ungültigen, gültigen und Grenzwerten für jede Regel, dann wird die Vorführung einfach dazu, Ihre eigenen Tests laut durchzugehen.



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:

- ✓ eine vollständige Speicher-Pipeline für ein einfaches Wisej.NET-Formular aufbauen.
- ✓ einen eigenen IErrorProvider für ein Zusammenfassungs-Panel erstellen.
- ✓ Feldfehler, Grid-Fehler und Modellfehler kombinieren.
- ✓ eine Validierungs-UX gestalten, die klar, barrierefrei und testbar ist.
- ✓ die Abschlussanwendung des Kurses fertigstellen.

Lab- und Prüfungsleitfaden

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



Praxis-Lab

Schließen Sie das ValidationClinic-Abschlussprojekt ab. Schreiben Sie `btnSave_Click` als feste Pipeline: Zusammenfassung zurücksetzen, `bindingSource.EndEdit()` aufrufen, abbrechen, wenn `ValidateChildren()` fehlschlägt, abbrechen, wenn die `DataAnnotations`-Prüfung fehlschlägt, abbrechen, wenn `ContactValidator` Geschäftsfehler zurückgibt, abbrechen, wenn das Kontakt-Grid noch Zeilenfehler hat, und erst dann innerhalb eines `try/catch` speichern, das einen doppelten Namen in einen Feldfehler und jede andere Exception in eine verständliche Meldung umsetzt. Legen Sie `SummaryErrorProvider` als Implementierung von `IErrorProvider` an, weisen Sie ihn `validation.ErrorProvider` zu und leiten Sie Fehler über eine Hilfsmethode `SetError`, die sowohl in den `Icon-ErrorProvider` als auch in den Zusammenfassungs-Provider schreibt. Testen Sie Pflicht-, ungültige, gültige und Grenzwerte und schreiben Sie ein kurzes README, das das Validierungsdesign erklärt.

Ausgangspunkt

ValidationClinic so, wie Sie es nach Modul 6 verlassen haben: Das Aufnahmeformular hat Feld-Handler, Extender-Regeln, `MinimumAgeValidationRule`, einen `ContactValidator`-Service und ein `ContactEditModel`, das über eine `BindingSource` gebunden ist, und das Kontakt-Grid validiert seine Zellen für E-Mail, Alter und Abschlussdatum. Jede Ebene funktioniert für sich, aber Speichern führt sie noch nicht alle der Reihe nach aus, und Fehler erscheinen an unterschiedlichen Stellen, je nachdem, welche Ebene sie gefunden hat.

Vorgeschlagenes Vorgehen

- ✓ Schreiben Sie zuerst `SummaryErrorProvider`: eine Klasse, die `IErrorProvider` implementiert, im Konstruktor das Zusammenfassungs-`Label` entgegennimmt, Control-Meldungs-Paare speichert, einen Eintrag bei leerer Meldung entfernt und bei jedem `SetError` Text und Sichtbarkeit des Labels neu setzt. Implementieren Sie `GetError` so, dass es für ein Control ohne Fehler einen leeren String zurückgibt.
- ✓ Erzeugen Sie den Provider rund um `validationSummaryLabel`, weisen Sie `validation.ErrorProvider = summaryProvider` zu und ergänzen Sie auf Formularebene eine Hilfsmethode `SetError(Control, string)`, die `errorProvider.SetError` und `summaryProvider.SetError` mit derselben Meldung aufruft. Ersetzen Sie direkte `SetError`-Aufrufe in Ihrem Zuordnungscode für Modell und Geschäftsregeln durch die Hilfsmethode.
- ✓ Schreiben Sie `btnSave_Click` als feste Pipeline neu: Zusammenfassung zurücksetzen, `bindingSource.EndEdit()`, `ValidateChildren()`, die Prüfung mit `Validator.TryValidateObject`, `ContactValidator.Validate` und dann eine Grid-Prüfung, die die Platzhalterzeile für neue Einträge überspringt und jede Zeile meldet, die noch `ErrorText` trägt. Jede Stufe kehrt zurück, sobald sie fehlschlägt.
- ✓ Setzen Sie nur das Speichern in `try / catch`: Fangen Sie zuerst `DuplicateNameException` ab und machen Sie daraus über Ihre Hilfsmethode einen Fehler an `txtName`, fangen Sie dann

`Exception` ab, loggen Sie sie mit `System.Diagnostics.Debug.WriteLine` und zeigen Sie einen verständlichen Text mit `MessageBox.Show` an. Rufen Sie bei Erfolg `AlertBox.Show("Saved.", MessageBoxIcon.Information)` auf.

- ✓ Schreiben Sie das README: einen kurzen Absatz pro Ebene (Feld, Modell, Speichern), eine Tabelle mit Testfällen mit Pflicht-, ungültigen, gültigen und Grenzwerten für jede Regel und die Checkliste des Abschlussprojekts. Üben Sie dann den Vorführungsablauf, bis Sie ihn in fünf Minuten durchspielen können.

Das Abschlussprojekt wird anhand von zehn geforderten Funktionen bewertet: Kundenname und E-Mail-Adresse als Pflichtfelder; ein `ErrorProvider` mit sichtbarer Zusammenfassung; der Validation-Extender mit mindestens drei integrierten Regeln; eine eigene `ValidationRule`-Klasse; ein Validator-Service, den Speichern verwendet; Modellvalidierung mit DataAnnotations oder `IDataErrorInfo`; Zellvalidierung für E-Mail und Alter im Grid; ein `DataError`-Handler im Grid; die Zusammenfassung über den eigenen `IErrorProvider`; und ein kurzes README. Die Vorführung folgt einer festen Reihenfolge: ein ungültiges Feld zeigen, zeigen, wie Speichern durch `ValidateChildren` gestoppt wird, einen Modellfehler zeigen, der nach `EndEdit` erscheint, die Zellvalidierung im Grid zeigen, den Zusammenfassungs-Provider zeigen, der alles auflistet, dann die Daten korrigieren und erfolgreich speichern. Prüfen Sie Ihre Masken vor der Abgabe anhand der zehn UX-Regeln aus der Lektion: Meldungen neben dem Feld und in einer Zusammenfassung, verständliche Sprache, die sagt, was zu tun ist, keine Schuldzuweisung, keine Stacktraces, kein Speichern ungültiger Daten, Abbrechen und Schließen nie ohne Grund blockiert, keine entfernten Prüfungen bei jedem Tastendruck, überall dieselben Modellregeln und alle vier Arten von Testwerten.

Geforderte Ergebnisse

- ✓ Speicher-Pipeline, die `EndEdit`, `ValidateChildren` sowie Modell-, Geschäfts- und Grid-Prüfungen vor dem Speichern ausführt
- ✓ Klasse `SummaryErrorProvider`, die `SetError` und `GetError` von `IErrorProvider` implementiert
- ✓ Hilfsmethode `SetError`, die sowohl in den Icon-`ErrorProvider` als auch in den Zusammenfassungs-Provider schreibt
- ✓ try/catch um das Speichern, das einen Fehler wegen eines doppelten Namens von Systemfehlern trennt
- ✓ README mit dem Validierungsdesign und Testfällen für Pflicht-, ungültige, gültige und Grenzwerte

Wiederholungsfragen

- ✓ Ein Tester tippt eine neue E-Mail-Adresse in das Formular und klickt sofort auf Speichern; gespeichert wird die alte Adresse. Welche Stufe der Pipeline fehlt oder steht an der falschen Stelle, und warum ist ihre Position wichtig?
- ✓ Ihr `SummaryErrorProvider` zeigt „Email is required.“ zweimal an, nachdem der Benutzer das Feld korrigiert und erneut ungültig gemacht hat. Welcher Teil von `SetError` sollte das verhindern, und wovon hängt er ab?

- ✓ Das Repository wirft beim Abbruch der Netzwerkverbindung eine Exception mit einer Datenbankmeldung. Gehen Sie durch, was der Benutzer sieht und was der Entwickler hinterher finden kann, und erklären Sie, warum sich das vom Fall des doppelten Namens unterscheidet.



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.