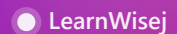


Validierung gebundener Daten und Modellfehler

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Das Modell validieren, nicht nur die Controls: BindingSource und EndEdit, DataAnnotations mit Validator.TryValidateObject ausführen, MemberNames wieder den Controls zuordnen, IDataErrorInfo mit einem datengebundenen ErrorProvider, BindToDataAndErrors und die sichere Reihenfolge beim Speichern. 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

Das Modell validieren, nicht nur die Controls: BindingSource und EndEdit, DataAnnotations mit Validator.TryValidateObject ausführen, MemberNames wieder den Controls zuordnen, IDataErrorInfo mit einem datengebundenen ErrorProvider, BindToDataAndErrors und die sichere Reihenfolge beim Speichern.

1. Das Problem im Produktivbetrieb

Alles, was ValidationClinic bisher prüft, steckt im Aufnahmeformular: hier ein `Validating`-Handler, dort eine Regel am `Validation`-Extender, ein Aufruf von `ContactValidator` in `btnSave_Click`. Ein Kontakt kommt aber nicht nur über dieses Formular herein. Er kann im Kontakt-Grid bearbeitet, aus einer Datei importiert oder von einem Service angelegt werden, und keiner dieser Wege führt die Handler des Formulars aus. Dieses Modul verlagert die grundlegenden Regeln auf das `ContactEditModel` selbst, sodass die Daten ihre eigene Definition von „gültig“ mitbringen und jede Maske ihnen dieselbe Frage stellen kann.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
BindingSource	Eine Komponente zwischen den Controls und Ihrem Objekt: Ihre <code>DataSource</code> ist das Modell, und jedes Control bindet sich über sie an eine Eigenschaft, statt von Hand kopiert zu werden.
EndEdit()	Überträgt den Wert, den der Benutzer noch bearbeitet, aus dem Control in das Modell, sodass die Validierung liest, was auf dem Bildschirm steht, und nicht den zuletzt übernommenen Wert.
DataAnnotations	Attribute wie <code>[Required]</code> , <code>[StringLength]</code> , <code>[EmailAddress]</code> und <code>[Range]</code> auf Modelleigenschaften, jeweils mit einer <code>ErrorMessage</code> , die für den Benutzer geschrieben ist.
Validator.TryValidateObject	Die .NET-Methode, die diese Attribute auf ein Objekt anwendet und eine Liste von <code>ValidationResult</code> -Objekten füllt; sie weiß nichts über Controls oder Wisej.NET.
MemberNames	Die Eigenschaftsnamen, zu denen ein <code>ValidationResult</code> gehört; sie sind die einzige Verbindung zwischen einer Fehlermeldung und dem Control, das sie anzeigen soll.
IDataErrorInfo	Ein Interface, über das ein Modell per Indexer die Frage „Was stimmt mit dieser Eigenschaft nicht?“ beantwortet, sodass die Binding-Schicht Fehler abfragen kann, statt dass Ihr Code sie hineinschiebt.
Datengebundener ErrorProvider	Ein <code>ErrorProvider</code> , dessen <code>DataSource</code> dieselbe Binding-Source ist wie die der Controls, sodass er die Fehler des Modells neben den gebundenen Controls anzeigt, ganz ohne <code>SetError</code> -Aufruf.

3. Der rote Faden der Lektion

Das Formular binden, dann das Modell validieren

Data Binding verbindet eine Control-Eigenschaft mit einer Modelleigenschaft. Legen Sie eine `BindingSource` an, setzen Sie ihre `DataSource` auf ein `ContactEditModel` und rufen Sie für jedes Feld `txtName.DataBindings.Add("Text", contactBindingSource, nameof(ContactEditModel.Name))` auf. Ab dann kopiert das Formular beim Klick auf Speichern keinen Text mehr in ein Objekt; das Modell sind die Daten des Formulars, und `contactBindingSource.Current` gibt es Ihnen zurück.

Es gibt eine Lücke, in die Einsteiger fast immer tappen. Binding aktualisiert das Modell nicht bei jedem Tastendruck. Tippt der Benutzer eine E-Mail-Adresse und klickt sofort auf Speichern, kann der neue Text noch in der TextBox stehen, während `Email` im Modell den alten Wert enthält. Wird das Modell in diesem Moment validiert, prüfen Sie Daten, die der Benutzer bereits ersetzt hat. Die Lösung ist ein einziger Aufruf: `contactBindingSource.EndEdit()` schließt die aktuelle Bearbeitung ab und überträgt die eingegebenen Werte in das Modell. Machen Sie ihn zur ersten Zeile jedes Speicherpfads, bevor irgendeine Regel das Modell liest.

DataAnnotations und Validator.TryValidateObject

DataAnnotations stellen einfache Regeln direkt neben die Daten, die sie schützen. Auf `ContactEditModel` erhält `Name` die Attribute `[Required]` und `[StringLength(80)]`, `Email` erhält `[Required]` und `[EmailAddress]`, und `Age` erhält `[Range(0, 120)]`. Jedes Attribut trägt eine `ErrorMessage` wie „Enter a valid email address.“ Behandeln Sie diese Meldungen als Produkttext, den der Benutzer liest, nicht als Notizen für Entwickler.

Attribute tun von sich aus nichts; irgendetwas muss sie ausführen. Legen Sie eine `List<ValidationResult>` und einen `ValidationContext` für das Modell an und rufen Sie dann `Validator.TryValidateObject(model, context, results, validateAllProperties: true)` auf. Die Methode gibt `false` zurück, wenn etwas fehlgeschlagen ist, und fügt pro verletzter Regel ein Ergebnis hinzu. Lassen Sie das letzte Argument nicht weg: Ohne `validateAllProperties: true` wird nur `[Required]` geprüft, und eine falsch aufgebaute E-Mail-Adresse oder ein Alter von 300 geht stillschweigend durch. Beachten Sie außerdem, dass `[EmailAddress]` und `[Range]` einen null-Wert als gültig behandeln; genau deshalb ist die Pflichtregel ein eigenes Attribut.

IDataErrorInfo und der datengebundene ErrorProvider

`IDataErrorInfo` kehrt die Beziehung um. Statt dass Ihr Code einen Validator ausführt und Meldungen auf Controls schiebt, beantwortet das Modell Fragen. Das Interface hat zwei Member: `Error`, eine Meldung zum Objekt als Ganzes (oft ein leerer String), und den Indexer `this[string columnName]`, der die Meldung für eine Eigenschaft zurückgibt oder einen leeren String, wenn diese Eigenschaft in Ordnung ist. Im zweiten Modell des Labs gibt der Indexer „Name is required.“ zurück, wenn `columnName` gleich `Name` und der Name leer ist.

Nützlich wird das Interface, wenn ein `ErrorProvider` mit denselben Daten verbunden ist. Setzen Sie `errorProvider.ContainerControl` auf das Formular und `errorProvider.DataSource` auf `contactBindingSource`. Der Provider fragt dann das aktuelle Element nach dem Fehler jeder gebundenen Eigenschaft und zeigt jede Meldung neben dem daran gebundenen Control an. Nirgends in Ihrem Code taucht ein `SetError`-Aufruf auf. .NET kennt außerdem `INotifyDataErrorInfo`, das mehrere Fehler pro Eigenschaft melden und asynchron ändern kann, aber für ein Einsteigerformular ist `IDataErrorInfo` der einfachere, gut unterstützte Ausgangspunkt.

BindToDataAndErrors zur Laufzeit

Bei einem datengebundenen Provider müssen zwei Einstellungen zusammenpassen: `DataSource`, das Objekt oder die Liste, und `DataMember`, der Pfad darin. Im Designer setzen Sie beide einmal und fassen sie nie wieder an. Zur Laufzeit, etwa wenn ValidationClinic den Editor vom annotierten Modell auf das `IDataErrorInfo`-Modell umschaltet, entsteht beim Setzen nacheinander ein Moment, in dem die neue Quelle mit dem alten Member gepaart ist, und der Provider versucht womöglich, sich an etwas zu binden, das nicht existiert.

`BindToDataAndErrors(source, member)` setzt beides in einem Aufruf, sodass der Provider immer nur ein zusammenpassendes Paar sieht. Wenn das gesamte Objekt die Daten sind, übergeben Sie `string.Empty` als Member: `errorProvider.BindToDataAndErrors(contactBindingSource, string.Empty)`. Verwenden Sie im Designer die beiden Eigenschaften und die Methode immer dann, wenn sich die Quelle bei geöffnetem Formular ändert.

Modellfehler Controls und der Zusammenfassung zuordnen

`Validator.TryValidateObject` weiß absichtlich nichts über Wisej.NET, daher muss das Formular die Ergebnisse übersetzen. Bauen Sie im Konstruktor eine Feldzuordnung auf, ein `Dictionary<string, Control>` von `nameof(ContactEditModel.Email)` auf `txtEmail` und so weiter, und durchlaufen Sie dann die Ergebnisse: Schlagen Sie für jeden Namen in `result.MemberNames` das Control nach und rufen Sie `errorProvider.SetError(control, result.ErrorMessage)` auf. Fügen Sie zum Schluss alle Meldungen in `validationSummaryLabel` zusammen und zeigen Sie es an, denn ein Icon allein wird leicht übersehen. Ein Ergebnis, dessen Member nicht in der Zuordnung steht, landet trotzdem in der Zusammenfassung, es geht also nie ein Fehler verloren. Die Zuordnung hat dieselbe Form wie `ApplyModelValidation` aus Modul 4, deshalb können sich beide eine Zusammenfassung teilen.

Sind alle Ebenen vorhanden, folgt der Speichern-Handler einer festen Reihenfolge: die Bearbeitung abschließen, die untergeordneten Controls validieren, das Modell validieren, feldübergreifende Regeln ausführen, nur speichern, wenn nichts fehlgeschlagen ist, und Systemausnahmen separat abfangen, damit ein Datenbankausfall nie wie ein Tippfehler aussieht.

```
private void btnSave_Click(object sender, EventArgs e)
{
    contactBindingSource.EndEdit();           // 1. eingegebene Werte ins Modell
    if (!ValidateChildren()) return;           // 2. Feldvalidierung
    var model = (ContactEditModel)contactBindingSource.Current;
    if (!ApplyAnnotationErrors(model)) return; // 3. DataAnnotations
    if (!ApplyModelValidation(model)) return;  // 4. ContactValidator, feldübergreifend
    try { _repository.Save(model); ShowSaved(); } // 5. erst jetzt
    catch (Exception ex) { ShowSystemError(ex); } // 6. keine Validierungsmeldung
}
```



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:

- ✓ Controls an ein Modell binden und das Modell vor dem Speichern validieren.
- ✓ DataAnnotations mit `Validator.TryValidateObject` verwenden.
- ✓ `IDataErrorInfo` und das Verhalten eines datengebundenen `ErrorProviders` verstehen.
- ✓ `BindToDataAndErrors` verwenden, wenn Sie die Datenquelle eines `Providers` zur Laufzeit ändern.

- ✓ Modellfehler Wisej.NET-Controls und Zusammenfassungen zuordnen.

Lab- und Prüfungsleitfaden

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



Praxis-Lab

Binden Sie das Aufnahmeformular von ValidationClinic über eine `contactBindingSource` an ein `ContactEditModel` und versehen Sie dessen Eigenschaften `Name`, `Email` und `Age` mit den Annotationen `Required`, `StringLength`, `EmailAddress` und `Range`. Rufen Sie in `btnSave_Click` zuerst `contactBindingSource.EndEdit()` auf, führen Sie `Validator.TryValidateObject` mit `validateAllProperties` auf `true` aus und ordnen Sie den Member-Namen jedes `ValidationResult` über eine Feldzuordnung und die Validierungszusammenfassung seinem Control zu. Implementieren Sie dann `IDataErrorInfo` auf einem zweiten Modell, richten Sie den `ErrorProvider` mit `BindToDataAndErrors` auf die Binding-Source aus und zeigen Sie, dass dieselben Fehler über Data Binding erscheinen, ohne handgeschriebenen `SetError`-Aufruf.

Ausgangspunkt

ValidationClinic nach Modul 4: Das Aufnahmeformular hat Feldregeln, einen `ContactValidator`-Service, `ApplyModelValidation` und eine `MinimumAgeValidationRule`, aber seine Controls sind an nichts gebunden, daher liest `btnSave_Click` die Werte immer noch direkt aus `txtName`, `txtEmail` und `txtAge`.

Vorgeschlagenes Vorgehen

- ✓ Legen Sie `ContactEditModel` mit `Name`, `Email` und einem nullable `Age` an, fügen Sie die vier Annotationen mit verständlichen `ErrorMessage`-Texten hinzu und binden Sie die drei TextBoxen mit `DataBindings.Add("Text", contactBindingSource, ...)` daran.
- ✓ Weisen Sie die Bearbeitungslücke nach, bevor Sie irgendeine Validierung schreiben: Tippen Sie eine E-Mail-Adresse ein, klicken Sie auf Speichern und loggen Sie `((ContactEditModel)contactBindingSource.Current).Email` einmal mit und einmal ohne vorangestellten Aufruf von `contactBindingSource.EndEdit()`.
- ✓ Schreiben Sie `ValidateModel` rund um `Validator.TryValidateObject(model, context, results, validateAllProperties: true)`, bauen Sie dann im Konstruktor `_fieldMap` auf und schreiben Sie eine Methode `ApplyAnnotationErrors`, die jeden Control-Fehler setzt und `validationSummaryLabel` füllt.
- ✓ Schreiben Sie `btnSave_Click` in der sicheren Reihenfolge neu: `EndEdit`, `ValidateChildren`, Annotationen, der `ContactValidator` aus Modul 4, Speichern und ein separates `catch`, das einen verständlichen Systemfehler anzeigt.
- ✓ Fügen Sie ein zweites Modell hinzu, das `IDataErrorInfo` implementiert, schalten Sie die Binding-Source zur Laufzeit darauf um, rufen Sie `errorProvider.BindToDataAndErrors(contactBindingSource, string.Empty)` auf und leeren Sie den Namen, um zu beobachten, wie der Fehler ohne jedes `SetError` erscheint.

Geforderte Ergebnisse

- ✓ `ContactEditModel` mit DataAnnotations, über eine BindingSource an das Formular gebunden

- ✓ Speichern-Handler, der EndEdit aufruft, bevor Validator.TryValidateObject läuft
- ✓ Feldzuordnung, die Member-Namen aus ValidationResult in Control-Fehler und Zeilen der Zusammenfassung umsetzt
- ✓ IDataErrorInfo-Modell, dessen Fehler über einen datengebundenen ErrorProvider erscheinen
- ✓ Aufruf von BindToDataAndErrors, wenn sich die Datenquelle des Providers zur Laufzeit ändert

Wiederholungsfragen

- ✓ Eine Testerin tippt eine neue E-Mail-Adresse ein und drückt zum Speichern die Eingabetaste, geprüft und gespeichert wird aber die alte, gültige Adresse. Welche Zeile fehlt, und warum sieht das Formular auf dem Bildschirm richtig aus, während das Modell falsch ist?
- ✓ Sie entfernen `validateAllProperties: true`, und jedes Speichern „funktioniert“ weiterhin. Welche Ihrer vier Annotationen laufen nicht mehr, und welche fehlerhaften Daten können jetzt das Repository erreichen?
- ✓ Der Import-Service legt Kontakte an, ohne das Formular zu öffnen. Welche Regeln dieses Moduls schützen diesen Weg weiterhin, und welche laufen nur, weil ein Benutzer auf Speichern geklickt hat?



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.