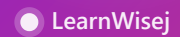


Validierung, ErrorProvider und Benutzer-Feedback

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Drei Validierungsebenen, DataAnnotations auf Edit-Modellen, ein TicketValidator-Service, der Feld- und Zusammenfassungsmeldungen liefert, ErrorProvider und zusammenfassendes Feedback in Wisej.NET sowie DbUpdateException als letzte Sicherheitsebene. Was das für Wisej.NET-Entwickler bedeutet, die mit echten Datenbanken arbeiten – und wie Sie dieses Modul lesen.



Fokus: Produktivbetrieb

Drei Validierungsebenen, DataAnnotations auf Edit-Modellen, ein TicketValidator-Service, der Feld- und Zusammenfassungsmeldungen liefert, ErrorProvider und zusammenfassendes Feedback in Wisej.NET sowie DbUpdateException als letzte Sicherheitsebene.

1. Das Problem im Produktivbetrieb

Der Ticket-Editor aus Modul 4 speichert, was auch immer der Benutzer eingegeben hat. Ein leerer Titel, ein Ticket ohne Kunde oder ein geschlossenes Ticket, das nächste Woche fällig ist – all das läuft durch `SaveChangesAsync` und wird, wenn überhaupt, von der Datenbank gestoppt, die mit einer Provider-Exception antwortet, die kein Support-Mitarbeiter lesen kann. Benutzer brauchen sofortige, verständliche Meldungen, bevor die Datenbank einen Datensatz ablehnt, und die Anwendung braucht eine letzte Verteidigungslinie, die auch dann greift, wenn die UI durch einen Import, eine zweite Oberfläche oder einen Bug umgangen wird.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
drei Validierungsebenen	Die UI-Validierung gibt sofortiges Feedback, die Domänenvalidierung setzt Regeln durch, die überall gelten müssen, und Datenbank-Constraints garantieren die abschließende Konsistenz; jede Ebene fängt ab, was die darüberliegende übersehen hat.
DataAnnotations auf dem Edit-Modell	Attribute wie <code>[Required]</code> und <code>[StringLength]</code> auf <code>TicketEditModel</code> beschreiben Eingaberegeln an einer deklarativen Stelle, getrennt von der EF-Core-Entität.
explizite Validierung	<code>Validator.TryValidateObject</code> führt die Annotationen bei Bedarf aus; EF Core erledigt das nicht vor <code>SaveChangesAsync</code> für Sie, also muss das Formular es aufrufen.
TicketValidator und ValidationMessage	Ein Service, der <code>ValidationMessage(FieldName, Message)</code> -Einträge liefert, sodass der Regelsatz testbar ist und das Formular nur noch Meldungen den Controls zuordnet.
ErrorProvider	Der Wisej.NET-Extender, der über <code>SetError</code> ein Icon mit Tooltip neben dem ungültigen Control anzeigt und auch an die Datenfehler des aktuellen Eintrags gebunden werden kann.
Validierungszusammenfassung	Ein Label oder eine <code>AlertBox</code> , die feldübergreifende Fehler auflistet, die zu keinem einzelnen Control gehören, etwa ein geschlossenes Ticket mit einem Fälligkeitsdatum in der Zukunft.
DbUpdateException	Die Exception, die EF Core auslöst, wenn die Datenbank die Änderung ablehnt; sie ist die letzte Sicherheitsebene und muss in eine verständliche Meldung übersetzt werden, niemals roh angezeigt.

3. Der rote Faden der Lektion

Drei Ebenen, eine letzte Instanz

Eine produktive Anwendung mit Wisej.NET und EF Core validiert an drei Stellen. Die UI-Ebene fängt offensichtliche Fehler ab, solange der Benutzer noch auf der Oberfläche ist: ein fehlender Titel, ein Fälligkeitsdatum im falschen Bereich, ein Kunde, der nie ausgewählt wurde. Die Domänenebene enthält Geschäftsregeln, die gelten müssen, egal welche Oberfläche, welcher Import oder welcher Hintergrund-Job den Datensatz anlegt. Die Datenbankebene mit ihren NOT-NULL-Spalten, eindeutigen Indizes, Fremdschlüsseln und dem `RowVersion`-Token aus Modul 2 ist die letzte Instanz für Konsistenz.

Die Ebenen sind keine Alternativen. UI-Feedback ist ein Entgegenkommen, das den Normalfall schnell macht; die Datenbank bleibt der einzige Ort, an dem Eindeutigkeit oder referenzielle Integrität bei gleichzeitig arbeitenden Benutzern garantiert werden kann. Der Editor des Support Desk validiert deshalb vor dem Speichern und rechnet trotzdem damit, dass die Datenbank gelegentlich Nein sagt.

Das Edit-Modell annotieren und dann explizit validieren

Modul 4 hat `TicketEditModel` als das Objekt eingeführt, an das der Editor bindet. Dieses Modul versieht es mit `System.ComponentModel.DataAnnotations`: `[Required, StringLength(180)]` auf `Title`, `[StringLength(4000)]` auf `Description`, `[Required]` auf `Status`, `Priority`, `CustomerId` und `CategoryId`. Die Fremdschlüssel sind absichtlich nullable `int?`: Eine `ComboBox` ohne Auswahl liefert null, und genau null lehnt `[Required]` ab, während ein nicht-nullable `int` stillschweigend als null durchginge.

Annotationen beschreiben Regeln; irgendetwas muss sie ausführen. `Validator.TryValidateObject` mit `validateAllProperties: true` wertet jedes Attribut aus und füllt eine Liste von `ValidationResult`-Einträgen. Das Lab kapselt diesen Aufruf in einem `TicketValidator`-Service, der die Ergebnisse zu `ValidationMessage`-Records zusammenführt und die Regeln anhängt, die Annotationen nicht ausdrücken können. Die feldübergreifende Regel ist das klassische Beispiel: Ein geschlossenes Ticket darf kein Fälligkeitsdatum in der Zukunft haben.

```
public IReadOnlyList<ValidationMessage> Validate(TicketEditModel model)
{
    var results = new List<ValidationResult>();
    var context = new ValidationContext(model);
    Validator.TryValidateObject(model, context, results, validateAllProperties: true);

    var messages = results
        .SelectMany(r => r.MemberNames.DefaultIfEmpty(null),
            (r, member) => new ValidationMessage(member, r.ErrorMessage ?? "Invalid
value."))
        .ToList();

    if (model.Status == "Closed" && model.DueDate?.Date > DateTime.Today)
        messages.Add(new ValidationMessage(nameof(model.DueDate), "Closed tickets cannot
have a future due date.));
    return messages;
}
```

Weil der Validator ein einfacher Service ohne Referenz auf `Wisej.Web` ist, laufen die fünf Negativ-Testfälle, die das Lab verlangt, im Tests-Projekt ohne Formular, Session oder Datenbank.

Warum Mapping-Attribute keine Validierungspipeline sind

EF Core versteht ebenfalls `[Required]` und `[MaxLength]`, aber auf einer Entität sind das Mapping-Anweisungen: Sie legen fest, ob eine Spalte nullable ist und wie breit sie ist. EF Core führt vor `SaveChangesAsync` keinen vollständigen DataAnnotations-Validierungsdurchlauf aus. Ein Titel mit 300 Zeichen auf einer Entität, die auf 180 gemappt ist, wird in .NET nicht abgefangen; er wird an die Datenbank gesendet, die ihn je nach Provider abschneidet oder einen Fehler auslöst. Halten Sie die Persistenzstruktur in der Fluent API auf `SupportDeskContext`, die Eingaberegeln auf dem Edit-Modell – und rufen Sie den Validator selbst auf.

Der praktische Test ist einfach: Braucht eine Regel eine Meldung, die ein Mensch lesen soll, gehört sie auf das Edit-Modell und in `TicketValidator`. Ist sie eine Tatsache über die Spalte, gehört sie in die Modellkonfiguration und die Migration.

ErrorProvider für Felder, eine Zusammenfassung für feldübergreifende Regeln

Der `ErrorProvider` von Wisej.NET hängt an jedes Control ein rotes Icon mit Tooltip. Der Editor hält eine Instanz, für das Formular erzeugt, und folgt in `SaveAsync` einer festen Abfolge: `Clear()` aufrufen, damit die Icons des vorigen Durchlaufs verschwinden, `EndEdit` auf der `BindingSource` aufrufen, damit das Modell enthält, was der Benutzer eingegeben hat, `TicketValidator.Validate` ausführen und dann die Meldungen durchgehen. Eine Meldung mit dem `FieldName` `Title`, `CustomerId`, `CategoryId` oder `DueDate` wird zu `errorProvider.SetError(control, message)` auf der passenden `TextBox`, `ComboBox` oder dem passenden `DateTimePicker`. Der Provider kann auch an den aktuellen Eintrag einer Datenquelle gebunden werden, sodass ein Modell, das seine eigenen Fehler meldet, Controls automatisch markiert, und der Validation-Extender deckt einfache Regeln pro Control ohne Code ab.

Feldübergreifende Regeln haben keinen einzelnen Ort; deshalb wird jede Meldung zusätzlich in einem Zusammenfassungs-Label zusammengeführt oder, wenn das Formular voll ist, in einer `AlertBox` angezeigt. Save bleibt deaktiviert, solange das Modell ungültig ist, und wird in `finally` wieder aktiviert, und die Methode kehrt zurück, bevor überhaupt ein `DbContext` erzeugt wird. Schlägt die Validierung fehl, wird EF Core nicht aufgerufen.

DbUpdateException als letzte Sicherheitsebene

Selbst ein vollständig validiertes Modell kann abgelehnt werden. Das Beispiel im Lab ist eine doppelte Ticketnummer: Zwei Agents legen in derselben Sekunde Tickets an, der eindeutige Index auf `Number` schlägt beim zweiten Speichern an, und EF Core löst `DbUpdateException` aus. Dieselbe Exception deckt Fremdschlüsselverletzungen ab, wenn eine Kategorie gelöscht wurde, während der Editor offen war, sowie Fehler bei Pflichtspalten, von denen die UI nie wusste.

Der Handler erledigt drei Dinge. Er protokolliert die vollständige Exception mit dem inneren Provider-Fehler über den Logger der Anwendung auf dem Server. Er zeigt dem Benutzer einen einfachen Satz, etwa „Das Ticket konnte nicht gespeichert werden, weil die Datenbank die Änderung abgelehnt hat. Bitte prüfen Sie die Pflichtfelder und die ausgewählten Lookup-Werte.“ Und wenn ein Lookup veraltet sein könnte, lädt er die Lookup-Listen neu, damit der Benutzer einen gültigen Wert wählen kann. Er zeigt nie die SQL-Anweisung, den Namen der Constraint oder irgendetwas, das einen Connection-String enthalten könnte. `DbUpdateConcurrencyException` leitet von `DbUpdateException` ab, fangen Sie sie daher zuerst ab; Modul 7 gibt ihr einen eigenen Dialog.



Typische Fehler

Annehmen, dass EF Core vor `SaveChanges` automatisch die gesamte DataAnnotations-Validierung ausführt; es nutzt sie nur für das Mapping. Benutzern rohes SQL oder Provider-Fehler zeigen, was Schemadetails und manchmal Verbindungsinformationen preisgibt. Wiederverwendbare Geschäftsregeln nur in Button-Handler schreiben, wo sie nicht testbar sind und in der nächsten Oberfläche, die ein Ticket speichert, stillschweigend fehlen.



Grundregel für Data Binding

Halten Sie die beiden Lebensdauern auseinander. Wisej.NET-Formulare, BindingSources und die ausgewählte Zeile leben für die Session; ein DbContext lebt für eine Operation. Erzeugen Sie ihn aus der Factory, fragen Sie ab oder speichern Sie, geben Sie ihn frei – und lassen Sie ihn nie zu Session-Zustand werden.



Lernziele

Am Ende dieses Moduls können Sie:

- ✓ System.ComponentModel.DataAnnotations oder eigene Validatoren auf Edit-Modellen einsetzen.
- ✓ Validierungsfehler mit ErrorProvider, Validation-Extender, Labels und Zusammenfassungsmeldungen anzeigen.
- ✓ verstehen, dass die Mapping-Attribute von EF Core kein vollständiges automatisches UI-Validierungssystem sind.
- ✓ Datenbank-Constraints und DbUpdateException als letzte Sicherheitsebene behandeln.

Lab- und Prüfungsleitfaden

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

Praxis-Lab

Ergänzen Sie den Ticket-Editor des Support Desk um Validierung: DataAnnotations auf TicketEditModel, einen TicketValidator, der Fehler auf Feldebene und zusammenfassende Fehler liefert, einen ErrorProvider, der vor jedem Durchlauf geleert und für Title, Customer, Category und DueDate gesetzt wird, ein Zusammenfassungs-Label für feldübergreifende Regeln, ein deaktiviertes Save, solange das Modell ungültig ist, einen DbUpdateException-Handler, der bei einer doppelten Ticketnummer eine verständliche Meldung anzeigt, und mindestens fünf Negativ-Testfälle.

Ausgangspunkt

Öffnen Sie die Projektmappe `SupportDesk` aus Modul 4: Der Ticket-Browser, die modalen Formulare Add Ticket und Edit Ticket, `TicketEditModel` und ein `TicketEditorForm` mit einer `BindingSource` und einem `ErrorProvider` existieren bereits, und `SaveAsync` enthält noch den Validierungs-Platzhalter, der jedes Modell bis zu `SaveChangesAsync` durchlässt. Das Services-Projekt hat noch keinen Validator, und das Tests-Projekt ist leer.

Vorgeschlagenes Vorgehen

- ✓ Versehen Sie `TicketEditModel` im Services-Projekt mit `[Required, StringLength(180)]` auf `Title`, `[StringLength(4000)]` auf `Description` und `[Required]` auf `Status`, `Priority`, `CustomerId` und `CategoryId`; lassen Sie die Fremdschlüssel nullable, damit ein nicht ausgewählter Lookup als null erkannt wird.
- ✓ Fügen Sie einen Record `ValidationMessage(FieldName, Message)` und einen Service `TicketValidator` hinzu, dessen Methode `Validate` `Validator.TryValidateObject` mit `validateAllProperties: true` ausführt, die Ergebnisse nach Member-Namen auflöst und die feldübergreifende Regel anhängt, dass ein geschlossenes Ticket kein Fälligkeitsdatum in der Zukunft haben darf – mit einem null- `FieldName` oder `DueDate` als Feld; registrieren Sie ihn in der Microsoft-DI neben der Context-Factory.
- ✓ Rufen Sie in `SaveAsync` `errorProvider.Clear()` und `editBindingSource.EndEdit()` auf und dann den Validator; schreiben Sie eine Methode `ShowValidation`, die `Title`, `CustomerId`, `CategoryId` und `DueDate` per `SetError` auf die passende `TextBox`, `ComboBox` und den passenden `DateTimePicker` abbildet, alle Meldungen im Zusammenfassungs-Label zusammenführt und zurückkehrt, bevor ein `DbContext` erzeugt wird.
- ✓ Lassen Sie Save deaktiviert, solange das Modell ungültig ist und solange ein Speichervorgang läuft, und stellen Sie es in `finally` wieder her; ergänzen Sie dann unterhalb des Concurrency-catch ein `catch (DbUpdateException)`, das die vollständige Exception serverseitig protokolliert, über `AlertBox` einen verständlichen Satz anzeigt und die Lookup-Listen neu lädt, wenn ein Fremdschlüssel veraltet sein könnte.
- ✓ Schreiben Sie im Tests-Projekt mindestens fünf Negativ-Fälle nur gegen `TicketValidator` (leerer Titel, Titel mit mehr als 180 Zeichen, fehlender Kunde, fehlende Kategorie, geschlossenes

Ticket mit Fälligkeitsdatum in der Zukunft), reproduzieren Sie dann die doppelte Ticketnummer gegen den eindeutigen Index und prüfen Sie, dass die Meldung kein SQL enthält.

Geforderte Ergebnisse

- ✓ DataAnnotations auf TicketEditModel und ein TicketValidator, der Fehler auf Feldebene und zusammenfassende Fehler liefert
- ✓ ErrorProvider vor jedem Durchlauf geleert und für Title, Customer, Category und DueDate gesetzt
- ✓ Zusammenfassungs-Label für feldübergreifende Regeln und Save deaktiviert, solange das Modell ungültig ist
- ✓ DbUpdateException mit einer verständlichen Meldung für eine doppelte Ticketnummer behandelt
- ✓ Mindestens fünf Negativ-Testfälle

Wiederholungsfragen

- ✓ Wenn Sie alle DataAnnotations von `TicketEditModel` entfernen, sie aber auf der Entität `Ticket` belassen: Welche Ihrer fünf Negativ-Tests würden dann noch vor `SaveChangesAsync` fehlschlagen, und warum?
- ✓ Die Regel für geschlossene Tickets meldet `DueDate` als Feld. Was würde sich für den Benutzer ändern, wenn sie gar kein Feld meldete, und wann wäre das die bessere Wahl?
- ✓ Zwei Agents legen in derselben Sekunde ein Ticket an, und das zweite Speichern löst `DbUpdateException` aus. Was sieht der zweite Agent, was protokolliert der Server, und was davon würden Sie einem Kunden zeigen?



Grundregel für Data Binding

Halten Sie die beiden Lebensdauern auseinander. Wisej.NET-Formulare, BindingSources und die ausgewählte Zeile leben für die Session; ein DbContext lebt für eine Operation. Erzeugen Sie ihn aus der Factory, fragen Sie ab oder speichern Sie, geben Sie ihn frei – und lassen Sie ihn nie zu Session-Zustand werden.