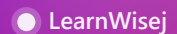


Bidirektionales Binding und CRUD-Editoren

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

DataBindings und BindingSource in Editoren, EndEdit, CancelEdit, AddNew und RemoveCurrent, direktes Binding an Entitäten im Vergleich zum Binding an ein Edit-Modell sowie Abläufe zum Speichern, Abbrechen, Löschen und Neuladen, die in SaveChangesAsync münden. Was das für Wisej.NET-Entwickler bedeutet, die mit echten Datenbanken arbeiten – und wie Sie dieses Modul lesen.



Fokus: Produktivbetrieb

DataBindings und BindingSource in Editoren, EndEdit, CancelEdit, AddNew und RemoveCurrent, direktes Binding an Entitäten im Vergleich zum Binding an ein Edit-Modell sowie Abläufe zum Speichern, Abbrechen, Löschen und Neuladen, die in SaveChangesAsync münden.

1. Das Problem im Produktivbetrieb

Die Bearbeiter im Support Desk wollen Oberflächen für Add Ticket und Edit Ticket, die sich wie Desktop-Formulare verhalten: ein modaler Dialog, die Buttons Save, Cancel und Delete und ein Grid, das die Änderung zeigt, sobald sich der Dialog schließt. Bidirektionales Binding macht diese Oberfläche schnell gebaut, aber Save ist eine bewusste Operation und kein Nebeneffekt des Tippens. Liest der Editor einen gebundenen Wert zu früh, speichert er eine Grid-Zeile ohne Tracking, als wäre sie getrackt, oder lässt er einen Doppelklick zweimal absenden, sieht das Formular weiterhin korrekt aus, während es stillschweigend Daten verliert oder dupliziert. Dieses Modul macht Speichern, Abbrechen, Löschen und Neuladen explizit, sodass sich der Editor wie ein Desktop-Formular anfühlt und trotzdem den Regeln für sicheren Datenzugriff im Web folgt.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
bidirektionales Binding	Eine Control-Eigenschaft wie <code>TextBox.Text</code> ist an eine Eigenschaft des aktuellen Eintrags gebunden, sodass das Control das Modell anzeigt und das Modell erhält, was der Benutzer eingegeben hat.
<code>DataSourceUpdateMode</code>	Das Argument von <code>DataBindings.Add</code> , das festlegt, wann ein Control seinen Wert in das Modell schreibt: bei der Validierung (wenn der Fokus das Control verlässt) oder bei jeder Eigenschaftsänderung.
EndEdit und CancelEdit	<code>BindingSource.EndEdit</code> übernimmt ausstehende Control-Änderungen in den aktuellen Eintrag; <code>CancelEdit</code> verwirft sie. Save beginnt immer mit EndEdit.
AddNew und RemoveCurrent	Listenoperationen der <code>BindingSource</code> , die einen neuen aktuellen Eintrag anlegen oder den ausgewählten aus der gebundenen Liste entfernen; keine von beiden berührt die Datenbank, bevor Sie speichern.
direktes Binding an Entitäten	Controls direkt an eine getrackte EF-Core-Entität binden, sodass der Change Tracker bereits weiß, was sich geändert hat. Produktiv für kleine, kurzlebige Formulare mit privatem Context.
Edit-Modell	Eine reine UI-Klasse wie <code>TicketEditModel</code> , die gebunden, validiert und dann auf die Entität gemappt wird – so erhalten Validierung, Formatierung und das Ausblenden sicherheitsrelevanter Felder eine eigene Grenze.
Speichern und Löschen mit frischem Context	Jedes Save und Delete erzeugt seinen eigenen <code>DbContext</code> aus der Factory, lädt die Entität über den Schlüssel, ändert sie und ruft <code>SaveChangesAsync</code> auf – so ist die Datenbank die maßgebliche Quelle, nicht die Grid-Zeile.

3. Der rote Faden der Lektion

Den Editor verdrahten: `DataBindings` und `BindingSource`

Ein typischer Editor beginnt damit, einer `BindingSource` ein Edit-Modell oder eine Entität zuzuweisen und jedes Control an eine Eigenschaft des aktuellen Eintrags zu binden. Im `TicketEditorForm` bindet das Titelfeld `Text` an `Title`, die Kunden-`ComboBox` bindet `SelectedValue` an `CustomerId`, die Datumsauswahl für das Fälligkeitsdatum bindet `Value` an `DueDate`, und eine `CheckBox` bindet `Checked` an ein boolesches Flag. Jeder Aufruf von `DataBindings.Add` benennt die Control-Eigenschaft, die `BindingSource` und den Member-Namen (verwenden Sie `nameof`, damit eine Umbenennung den Build bricht und nicht die Oberfläche), aktiviert die Formatierung und wählt einen `DataSourceUpdateMode`. `OnValidation` passt zu Texten und Datumswerten, weil der Wert übernommen wird, wenn der Fokus das Control verlässt; `OnPropertyChanged` passt zu Lookups, bei denen die Auswahl sofort im Modell ankommen soll.

Bei Lookups kommt es auf die Reihenfolge an. Setzen Sie `DisplayMember` und `ValueMember` und laden Sie die Listen für Kunden, Agents und Kategorien, bevor das Edit-Modell zur `DataSource` wird; andernfalls hat der gebundene Fremdschlüssel nichts, was er auswählen kann. `LoadEditorAsync` baut das Modell auf:

ein leeres `TicketEditModel` mit Standardwerten für ein neues Ticket oder ein Lesen ohne Tracking, das in das Modell gemappt wird, einschließlich der `RowVersion`, die die Nebenläufigkeitsprüfung später braucht.



Typischer Fehler

`TextBox.Text` oder das Modell vor `EndEdit` lesen, während die `BindingSource` ausstehende Änderungen hat. Das zuletzt bearbeitete Feld ist dann noch nicht übernommen, und `Save` schreibt den vorherigen Wert.

EndEdit, CancelEdit, AddNew und RemoveCurrent

Klickt der Benutzer auf `Save`, ist die Abfolge fest: `ValidateChildren`, damit jeder `Validating`-Handler läuft, `BindingSource.EndEdit`, damit der letzte ausstehende Control-Wert im aktuellen Eintrag landet, Domänenvalidierung auf dem Modell, bei Verwendung eines Edit-Modells das Kopieren der freigegebenen Werte auf die EF-Core-Entität und dann `SaveChangesAsync`. Lesen Sie das Modell über `BindingSource.Current`, nicht über die Controls.

Die übrigen `BindingSource`-Member verwalten den aktuellen Eintrag, nicht die Datenbank. `CancelEdit` verwirft ausstehende Änderungen am aktuellen Eintrag, sofern der Eintrag die Semantik editierbarer Objekte unterstützt; ein einfaches Modell wird schlicht neu geladen oder das Formular freigegeben. `AddNew` erzeugt einen neuen Eintrag vom Elementtyp der Liste und macht ihn zum aktuellen, sodass ein Add-Ticket-Ablauf, der sich eine Liste mit einem Grid teilt, dort beginnen kann. `RemoveCurrent` nimmt den ausgewählten Eintrag aus der gebundenen Liste, sodass sich das Grid sofort aktualisiert. Keiner davon hat mit EF Core gesprochen; eine entfernte Zeile taucht bei der nächsten Suche wieder auf, sofern kein frischer Context sie löscht.

Abläufe zum Speichern, Abbrechen, Löschen und Neuladen

Die Speichermethode sollte doppelte Klicks verhindern, Binding-Bearbeitungen abschließen, validieren, einen `DbContext` erzeugen, die Entität laden oder eine neue anlegen, Felder mappen, `SaveChangesAsync` aufrufen und den Editor schließen oder aktualisieren. Schützen Sie die Methode mit einem Speicher-Flag und einem deaktivierten `Save`-Button, stellen Sie beides in `finally` wieder her und halten Sie den Erfolgspfad kurz:

```
editBindingSource.EndEdit();
var model = (TicketEditModel)editBindingSource.Current!;
var errors = Validator.Validate(model);
if (errors.Count > 0) { ShowValidation(errors); return; }

await using var db = await DbFactory.CreateDbContextAsync();
var ticket = model.Id == 0
    ? db.Tickets.Add(new Ticket { Number = await NextNumberAsync(db) }).Entity
    : await db.Tickets.SingleAsync(t => t.Id == model.Id);
ticket.Title = model.Title.Trim();
ticket.CustomerId = model.CustomerId!.Value;
ticket.DueDate = model.DueDate;
await db.SaveChangesAsync();
DialogResult = DialogResult.OK; Close();
```

Der Fehlerpfad ist explizit. Fangen Sie zuerst `DbUpdateConcurrencyException` ab: Ein anderer Bearbeiter hat dasselbe Ticket gespeichert, und der Benutzer braucht einen Konfliktdialog, keinen allgemeinen Fehler. Fangen Sie `DbUpdateException` separat ab: Die Datenbank hat die Änderung abgelehnt (ein fehlender Lookup, eine doppelte Nummer); protokollieren Sie die Details auf dem Server und zeigen Sie eine verständliche `AlertBox`. Cancel ist der einfachste Pfad: Der Context wurde am Ende von `LoadEditorAsync` freigegeben, ein Schließen mit `DialogResult.Cancel` speichert also nichts. Das Neuladen ist Aufgabe des übergeordneten Formulars: Wenn `ShowDialog` `DialogResult.OK` zurückgibt, führt der Ticket-Browser seine Suche erneut aus.

Delete sollte sich nicht nur auf die ausgewählte Grid-Zeile verlassen. Lassen Sie die Aktion bestätigen, erzeugen Sie einen frischen Context, laden Sie die Entität über den Schlüssel, prüfen Sie Geschäftsregeln wie „Tickets mit Kommentaren dürfen nicht gelöscht werden“, entfernen Sie sie und rufen Sie `SaveChangesAsync` auf. Liefert das Laden null, hat jemand anderes sie bereits gelöscht: Zeigen Sie eine verständliche Meldung und aktualisieren Sie die Liste. Wo Nachvollziehbarkeit wichtig ist, bevorzugen Sie ein Soft Delete, das die Zeile behält.



Typischer Fehler

Listeneinträge ohne Tracking speichern, als wären sie getrackte Entitäten. Ein `TicketListItem` aus dem Grid wurde mit `AsNoTracking` projiziert; wer ihn ändert und auf einem neuen Context `SaveChangesAsync` aufruft, schreibt nichts. Laden Sie stattdessen die Entität über den Schlüssel im Speicher-Context.



Typischer Fehler

Save nicht gegen Doppelklicks schützen. Zwei überlappende `SaveAsync` -Aufrufe legen zwei Tickets an oder konkurrieren um dieselbe Zeile; das Speicher-Flag plus ein deaktivierter Button, wiederhergestellt in `finally`, gehört zur Korrektheit, nicht zum Feinschliff.

Direktes Binding an Entitäten oder ein Edit-Modell?

Direktes Binding an Entitäten ist für einfache Oberflächen produktiv, weil das Change Tracking von EF Core die ursprünglichen und aktuellen Werte bereits kennt; Save ist dann kaum mehr als `SaveChangesAsync`. Verwenden Sie es, wenn das Formular ein Aggregat bearbeitet, der Context privat für das Formular ist, das Formular schnell freigegeben wird und der Benutzer durch Freigeben oder Neuladen abbrechen kann. Binden Sie niemals direkt an einen Grid-Eintrag ohne Tracking in der Erwartung, dass EF Core die Änderungen speichert: Niemand trackt ihn.

Ein Edit-Modell zieht eine bewusste Abgrenzung zwischen UI und Persistenz. Es ist die bessere Wahl für Validierung, Formatierung, berechnete Anzeigewerte, das Ausblenden sicherheitsrelevanter Felder und Teil-Updates. Der Ablauf: die getrackte Entität laden, sie auf `TicketEditModel` mappen, das Modell binden, es validieren und dann nur die freigegebenen Werte auf die getrackte Entität zurückkopieren. Das Modell kann `CreatedAt` und `Number` ganz weglassen, sodass ein Benutzer kein Feld bearbeiten kann, das die Oberfläche nie anbieten sollte, und ein Validator, der ein einfaches Modell entgegennimmt, braucht für Tests keine Datenbank. Entscheiden Sie nach Risikoprofil: Ein privates, kurzlebiges Formular für ein

einzelnes Aggregat darf an die Entität binden; alles mit Regeln, Rollen oder Teil-Updates bekommt ein Modell.



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:

- ✓ Editor-Controls mit DataBindings und BindingSource an Modelleigenschaften binden.
- ✓ BindingSource.EndEdit, CancelEdit, AddNew, RemoveCurrent und die Verwaltung des aktuellen Eintrags korrekt einsetzen.
- ✓ Abläufe zum Anlegen, Bearbeiten, Löschen, Speichern, Abbrechen und Neuladen implementieren.
- ✓ anhand des Risikoprofils des Formulars zwischen direktem Binding an Entitäten und Binding an ein Edit-Modell wählen.

Lab- und Prüfungsleitfaden

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



Praxis-Lab

Bauen Sie die modalen Formulare Add Ticket und Edit Ticket für die Support Desk Data Console: ein TicketEditModel, ein TicketEditorForm mit einer BindingSource und einem ErrorProvider, TextBox-, ComboBox-, DateTimePicker- und CheckBox-Eigenschaften, die an das Modell gebunden sind, LoadEditorAsync für neue und bestehende Tickets, SaveAsync mit EndEdit, einem Platzhalter für die Validierung, Mapping und SaveChangesAsync, Delete mit Bestätigung und frisch geladener Entität sowie ein übergeordnetes Grid, das sich nach DialogResult.OK aktualisiert.

Ausgangspunkt

Öffnen Sie die Projektmappe Support Desk Data Console aus Modul 3. Die Ticket-Browser-Page sucht bereits `TicketListItem`-Zeilen und zeigt sie seitenweise über eine `BindingSource` in einem `DataGridView` an, die Lookup-ComboBoxen für Status und Kunde laden vor der ersten Suche, und `SupportDeskContext` ist über `AddDbContextFactory` registriert. Noch kann nichts ein Ticket anlegen, bearbeiten oder löschen.

Vorgeschlagenes Vorgehen

- ✓ Erstellen Sie zuerst `TicketEditModel` im Services-Projekt, nur mit den Eigenschaften, die die Oberfläche bearbeiten darf, plus `Id` und `RowVersion`, und entscheiden Sie, welche Felder der Entität draußen bleiben.
- ✓ Fügen Sie `TicketEditorForm` mit einer injizierten `IDbContextFactory<SupportDeskContext>`, einer `BindingSource`, einem `ErrorProvider` und den Buttons Save, Cancel und Delete hinzu; laden Sie die Lookups, bevor das Modell zugewiesen wird.
- ✓ Binden Sie jedes Control mit `DataBindings.Add` und einem expliziten `DataSourceUpdateMode` und schreiben Sie dann `LoadEditorAsync` für den Fall eines neuen und den eines bestehenden Tickets.
- ✓ Schreiben Sie `SaveAsync` als die Pipeline aus der Lektüre (Schutz, `EndEdit`, validieren, frischer Context, laden oder anlegen, mappen, `SaveChangesAsync`, schließen) und `DeleteAsync` mit Bestätigung und frischem Laden über den Schlüssel.
- ✓ Verbinden Sie die Buttons Add und Edit des Ticket-Browsers mit `ShowDialog` und führen Sie die Suche nur erneut aus, wenn das Ergebnis `DialogResult.OK` ist; testen Sie dann jeden Pfad mit einer zweiten offenen Browser-Session.

Geforderte Ergebnisse

- ✓ TicketEditModel und TicketEditorForm mit einer BindingSource und einem ErrorProvider
- ✓ TextBox-, ComboBox-, DateTimePicker- und CheckBox-Eigenschaften an das Edit-Modell gebunden

- ✓ LoadEditorAsync für neue und bestehende Tickets und SaveAsync mit EndEdit, Mapping und SaveChangesAsync
- ✓ Delete mit Bestätigung, frisch geladener Entität und sauberer Behandlung bereits gelöschter Datensätze
- ✓ Übergeordnetes Grid nach DialogResult.OK aktualisiert, und Cancel speichert nichts

Wiederholungsfragen

- ✓ Welche Lebensdauer haben Sie für den DbContext im Editor gewählt, und was müsste für das Formular gelten, damit ein Context mit Formular-Lebensdauer akzeptabel wäre?
- ✓ Welche Werte sind an die UI gebunden, welche leben nur im Edit-Modell, welche bleiben nur in der Datenbank – und warum ist **Number** nicht bearbeitbar?
- ✓ Welche Fehlerfälle haben Sie explizit behandelt (Doppelklick, bereits gelöscht Ticket, Ablehnung durch die Datenbank, gleichzeitige Bearbeitung), und was sieht der Bearbeiter jeweils?



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.