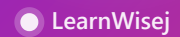


Nebenläufigkeit, Transaktionen, Deployment, Diagnose und Abschlussprojekt

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Optimistische Nebenläufigkeit mit RowVersion und DbUpdateConcurrencyException, ein Konfliktdialog mit Reload, Overwrite und Merge, Transaktionen und Ausführungsstrategien, umgebungsspezifische Konfiguration und Logging, eine Teststrategie und das Support-Desk-Abschlussprojekt. Was das für Wisej.NET-Entwickler bedeutet, die mit echten Datenbanken arbeiten – und wie Sie dieses Modul lesen.



Fokus: Produktivbetrieb

Optimistische Nebenläufigkeit mit RowVersion und DbUpdateConcurrencyException, ein Konfliktdialog mit Neuladen, Überschreiben und Zusammenführen, Transaktionen und Ausführungsstrategien, umgebungsspezifische Konfiguration und Logging, eine Teststrategie und das Support-Desk-Abschlussprojekt.

1. Das Problem im Produktivbetrieb

Zwei Bearbeiter öffnen das Ticket T-1042 in zwei Browser-Sessions; der eine schließt es, der andere erhöht seine Priorität, und welches Speichern als zweites ankommt, entscheidet, was sich die Datenbank merkt. Zeilen zu sperren ist in einer Webanwendung keine Option; die Support Desk Data Console muss also beide Benutzer arbeiten lassen, die Kollision beim Speichern erkennen, sie erklären und einen Menschen entscheiden lassen. Dieses Modul behandelt außerdem Transaktionen, Konfiguration, Migrationen, Logging, Tests und das Abschlussprojekt.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
optimistische Nebenläufigkeit	Benutzer bearbeiten frei; statt Zeilen zu sperren, erkennt die Anwendung beim Speichern, dass jemand anderes die Zeile seit dem Laden geändert hat.
RowVersion-Token	Ein von der Datenbank erzeugtes Concurrency-Token; EF Core setzt seinen Originalwert in die WHERE-Klausel des UPDATE, sodass eine geänderte Zeile zu null aktualisierten Zeilen führt.
OriginalValue	Die Version, die der Editor geladen hat, vor <code>SaveChangesAsync</code> auf der getrackten Entität wiederhergestellt, damit EF Core mit dem vergleicht, was der Benutzer gesehen hat.
DbUpdateConcurrencyException	Wird ausgelöst, wenn eine erwartete Zeile nicht betroffen war; ihre <code>Entries</code> enthalten die vorgeschlagenen Werte, und <code>GetDatabaseValuesAsync</code> holt, was jetzt in der Datenbank steht.
Konfliktdialog	Ein Wisej.NET-Formular, das Feld, Ihren Wert, Datenbankwert und Originalwert auflistet und Reload, Overwrite (sofern die Richtlinie es erlaubt) oder Merge anbietet.
Ausführungsstrategie	Der Wiederholungsmechanismus von EF Core für vorübergehende Fehler; eine explizite Transaktion muss darin laufen, damit eine Wiederholung die gesamte Unit of Work erneut ausführen kann.
idempotentes Migrationsskript	SQL aus <code>dotnet ef migrations script --idempotent</code> , das vor jedem Schritt die Migrationshistorie prüft und daher in jeder Umgebung sicher ist.

3. Der rote Faden der Lektion

Zwei Bearbeiter, ein Ticket

Optimistische Nebenläufigkeit lässt jeden Bearbeiter das Ticket T-1042 bearbeiten und prüft beim Speichern, ob die Zeile noch so aussieht wie beim Öffnen des Editors. Die Prüfung ist die `RowVersion` - Spalte, die in Modul 2 mit `IsRowVersion` konfiguriert wurde. Die Datenbank ändert sie bei jedem Update, und wenn die getrackte Entität die ursprüngliche Version trägt, schreibt EF Core `UPDATE ... WHERE Id = @id AND RowVersion = @original`. Hat eine andere Session zuerst gespeichert, passt die WHERE-Klausel auf nichts, null Zeilen sind betroffen, und `SaveChangesAsync` löst `DbUpdateConcurrencyException` aus.

Der Haken: Der Editor hält zwischen Laden und Speichern keine getrackte Entität am Leben. Er mappt ein Lesen ohne Tracking in `TicketEditModel`, und der `DbContext`, der speichert, ist ein frischer. Die Version, die der Benutzer gesehen hat, reist daher als verborgener Zustand im Edit-Modell mit: eine `RowVersion` - Eigenschaft, die nie an ein Control gebunden wird. Bevor ein bestehendes Ticket gespeichert wird, lädt das Formular die Entität neu und stellt das Token mit `db.Entry(ticket).Property(t => t.RowVersion).OriginalValue = model.RowVersion` wieder her. Ohne diese Zeile vergleicht der frische Context mit der Version, die er gerade geladen hat – und die ist immer aktuell, sodass nie ein Konflikt erkannt wird.

Kommt die Exception, übersetzen Sie sie in die Sprache des Fachbereichs. `ex.Entries` listet die fehlgeschlagenen Einträge auf; für jeden liefert `GetDatabaseValuesAsync`, was jetzt in der Datenbank steht, oder `null`, wenn ein anderer Benutzer die Zeile gelöscht hat, und der Vergleich von `CurrentValues` mit den Datenbankwerten ergibt die Konfliktliste:

```
catch (DbUpdateConcurrencyException ex)
{
    var conflicts = new List<ConflictField>();
    foreach (var entry in ex.Entries)
    {
        var dbValues = await entry.GetDatabaseValuesAsync();
        if (dbValues is null) { conflicts.Add(new("Ticket", "your edit", "deleted"));
        continue; }
        foreach (var p in entry.Metadata.GetProperties())
            if (!Equals(entry.CurrentValues[p], dbValues[p]))
                conflicts.Add(new(p.Name, entry.CurrentValues[p], dbValues[p]));
    }
    await ShowConflictDialogAsync(conflicts, ex);    // Reload · Overwrite · Merge
}
```

Ein Konfliktdialog mit Reload, Overwrite und Merge

Der Dialog erklärt, was passiert ist: Jemand hat dieses Ticket geändert, während Sie es bearbeitet haben. Er zeigt pro Feld Ihren Wert, den aktuellen Datenbankwert und, wo hilfreich, den Originalwert. **Reload** verwirft die lokalen Änderungen: `ReloadAsync` für jeden Eintrag oder ein erneutes `LoadEditorAsync` ersetzt das Modell durch die Datenbankwerte und die neue `RowVersion`. **Overwrite** speichert die Werte des Benutzers trotzdem, indem es `OriginalValue` auf die Datenbankversion setzt und `SaveChangesAsync` erneut aufruft; es wird nur angeboten, wenn die Richtlinie es erlaubt, zum Beispiel einem Supervisor. **Merge** übernimmt einige Werte von jeder Seite und speichert dann auf Basis der aktuellen Version.

Unabhängig von der Wahl: Aktualisieren Sie anschließend das Grid, damit der sichtbare Zustand der Datenbank entspricht; der Ticket-Browser ohne Tracking aktualisiert sich nicht selbst. Der catch-Block steht in `SaveAsync` neben dem `DbUpdateException`-Handler, und der Lade-Schutz hält `_saving` auf true, solange der Dialog offen ist.



Typischer Fehler: die Änderung eines anderen Benutzers stillschweigend überschreiben

Wer `DbUpdateConcurrencyException` abfängt und mit der `RowVersion` aus der Datenbank erneut speichert, öffnet das Ticket wieder, das ein Kollege gerade geschlossen hat. Overwrite ist eine Richtlinienentscheidung im Dialog, niemals eine automatische Wiederholung.

Transaktionen und Ausführungsstrategien

`SaveChangesAsync` ist für die Änderungen in diesem Aufruf bereits transaktional: Wenn der Editor ein `Ticket` und seinen ersten `TicketComment` in einem Context hinzufügt und einmal speichert, werden beide Zeilen geschrieben oder keine. Explizite Transaktionen sind nur nötig, wenn mehrere Operationen gemeinsam gelingen oder scheitern müssen, etwa das Schließen eines Tickets und das Archivieren seiner Kommentare in zwei Aufrufen; dann umschließt `db.Database.BeginTransactionAsync` die Einheit.

Verbindungsresilienz bringt eine Besonderheit mit sich. Eine mit `EnableRetryOnFailure` aktivierte Ausführungsstrategie wiederholt vorübergehende Fehler automatisch, aber nur für Operationen, die ihr gehören. Eine außerhalb der Strategie begonnene Transaktion kann nicht wiederholt werden; deshalb verweigert EF Core das Starten einer solchen, solange Wiederholungen aktiviert sind. Holen Sie die Strategie über `db.Database.CreateExecutionStrategy()` und übergeben Sie die gesamte Einheit, einschließlich Begin und Commit, an `ExecuteAsync`; eine Wiederholung kann sie zweimal ausführen, sie muss also gefahrlos wiederholbar sein.

Migrationen, Konfiguration und Logging

Migrationen sind Quellcode und werden wie jeder andere geprüft. Erzeugen Sie für den Produktivbetrieb mit `dotnet ef migrations script --idempotent` ein Skript, prüfen Sie das SQL und führen Sie es als Release-Schritt aus, oder liefern Sie – sofern freigegeben – ein `dotnet ef migrations bundle` aus. Die Design-Time-Factory hält die Tools unabhängig vom Wisej.NET-Startup, und das Web-Projekt darf in einer Umgebung mit mehreren Instanzen beim Start nicht `Migrate()` aufrufen.



Typischer Fehler: Migrationen automatisch von jeder Produktivinstanz aus anwenden

Zwei Server, die im selben Moment starten, konkurrieren darum, dieselben Tabellen zu ändern, und es gibt kein geprüftes Skript, von dem aus zurückgerollt werden kann. Schemaänderungen sind ein Release-Schritt, keine Funktion der Anwendung.

Connection-Strings und Secrets kommen aus umgebungsspezifischer Konfiguration, Secret Stores oder Deployment-Einstellungen, niemals aus dem Quellcode: Der Entwicklungs-String liegt in `appsettings.Development.json` oder in den User Secrets, und den Rest injiziert der Host. Dieselbe Disziplin schützt die Logs. `EnableSensitiveDataLogging` schreibt Parameterwerte in die SQL-Ausgabe und wird daher an die Bedingung geknüpft, dass die Umgebung Development ist. Setzen Sie im Produktivbetrieb die Kategorie `Microsoft.EntityFrameworkCore.Database.Command` auf Warning und protokollieren Sie genug, um einen Fehler zu diagnostizieren, ohne personenbezogene Daten oder Datenbank-Zugangsdaten zu schreiben.



Typischer Fehler: Logging sensibler Daten im Produktivbetrieb aktivieren

Ticket-Titel, Kundennamen und Zugangsdaten landen in Log-Dateien, die viele Leute lesen können. Knüpfen Sie die Einstellung an die Prüfung auf die Development-Umgebung, nicht an ein Flag, das jemand auf einem Live-Server umlegen könnte.

Tests, KI-Grounding und das Abschlussprojekt

Testen Sie die Abfrage-Services getrennt von der Wisej.NET-UI: `SearchTicketsAsync` und `TicketValidator` hängen von keinem Formular ab. Verwenden Sie SQLite für Integrationstests, die relationales Verhalten brauchen; den InMemory-Provider von EF Core nur für einfache Tests, bei denen Constraints und SQL-Übersetzung keine Rolle spielen. Testen Sie die Validatoren, das Mapping, den Nebenläufigkeitspfad mit zwei Contexts und das Migrationsskript gegen eine leere Datenbank.

Der Kurs enthält ein Grounding-Paket für KI. Ein kursspezifischer Assistent antwortet zuerst aus diesem Material, nennt den Modulabschnitt oder den Eintrag der Source Map, auf den er sich stützt, und sagt es

offen, wenn er sich bei providerspezifischem Verhalten unsicher ist, statt etwas zu erfinden. Er empfiehlt kurzlebige Contexts, **BindingSource** als Anbindung an die UI, explizite Validierung und asynchrone Lade-Schutzmechanismen, sofern es keinen triftigen Grund für eine andere Wahl gibt; Ihre Notiz zum Abschlussprojekt ist sein letzter Grounding-Baustein.

Das Abschlussprojekt ist die Support Desk Data Console selbst: der projizierte, seitenweise ladende Ticket-Browser, gebunden über eine **BindingSource** ; Lookup-ComboBoxen; der modale Editor mit Edit-Modell und **ErrorProvider** -Validierung; asynchrones Laden, Speichern und Löschen hinter Lade-Schutzmechanismen; RowVersion-Nebenläufigkeit mit Konfliktdialog; ein Plan für Migrationsskript oder Bundle mit Konfigurationshinweisen; und die Grounding-Notiz. Das Bewertungsraster gewichtet Architektur, Binding und korrekten Einsatz von EF Core mit je 20 %, Validierungs-UX und Performance mit je 15 % sowie Deployment und Diagnose mit 10 %.



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:

- ✓ Konflikte bei optimistischer Nebenläufigkeit in einem benutzerfreundlichen Wisej.NET-Dialog behandeln.
- ✓ Transaktionen und Ausführungsstrategien angemessen einsetzen.
- ✓ Abläufe für Migration und Deployment sowie umgebungsspezifische Konfiguration vorbereiten.
- ✓ Logging, Diagnose, eine Teststrategie und Grounding-Material für KI-Assistenten ergänzen.

Lab- und Prüfungsleitfaden

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



Praxis-Lab

Machen Sie die Support Desk Data Console sicher für mehrere Benutzer: Ergänzen Sie RowVersion im Edit-Modell und in seinem verborgenen Zustand, setzen Sie OriginalValue, bevor Sie eine bestehende Entität speichern, reproduzieren Sie einen Konflikt, indem Sie dasselbe Ticket in zwei Browser-Sessions bearbeiten, fangen Sie DbUpdateConcurrencyException ab, erstellen Sie eine Konfliktliste mit vorgeschlagenen Werten und Datenbankwerten, ergänzen Sie gemäß Richtlinie die Pfade Reload und Overwrite, erzeugen Sie ein idempotentes SQL-Migrationsskript mit Deployment-Hinweisen und umgebungsspezifischen Connection-Strings, lassen Sie das Logging sensibler Daten außerhalb der Entwicklung ausgeschaltet und schließen Sie dann das Review des Abschlussprojekts ab.

Ausgangspunkt

Öffnen Sie die Support Desk Data Console, die Sie durch die Module 1 bis 6 weitergeführt haben:

`SupportDeskContext` konfiguriert `RowVersion` bereits mit `IsRowVersion`, der seitenweise ladende Ticket-Browser ist eine Projektion ohne Tracking, und `TicketEditorForm` speichert ein `TicketEditModel` über `TicketValidator`, einen `ErrorProvider` und einen `DbUpdateException`-Handler. Was fehlt, ist der Nebenläufigkeitspfad: Das Edit-Modell trägt keine Version, `SaveAsync` hat kein catch für `DbUpdateConcurrencyException`, und es gibt kein Migrationsskript für den Produktivbetrieb.

Vorgeschlagenes Vorgehen

- ✓ Fügen Sie `TicketEditModel` eine Eigenschaft `byte[] RowVersion` hinzu, füllen Sie sie in `LoadEditorAsync`, halten Sie sie als verborgenen Zustand, an den kein Control gebunden ist, und stellen Sie sie in `SaveAsync` mit `db.Entry(ticket).Property(t => t.RowVersion).OriginalValue = model.RowVersion` wieder her, bevor Sie die übrigen Felder mappen.
- ✓ Reproduzieren Sie den Konflikt, bevor Sie ihn behandeln: Öffnen Sie die Anwendung in zwei Browser-Sessions (ein zweites Browserprofil oder ein privates Fenster, damit es eine andere Wisej.NET-Session ist), öffnen Sie in beiden das Ticket T-1042, speichern Sie zuerst in der ersten, dann in der zweiten, und prüfen Sie im Debugger, dass `SaveChangesAsync` `DbUpdateConcurrencyException` auslöst.
- ✓ Fangen Sie die Exception in `SaveAsync` ab, erstellen Sie die `ConflictField`-Liste aus `ex.Entries` mit `GetDatabaseValuesAsync` (ein null-Ergebnis gilt als „von einem anderen Benutzer gelöscht“) und zeigen Sie sie in einem Formular `ConflictDialog`, dessen `DataGridView` Feld, Ihren Wert, Datenbankwert und Originalwert auflistet.
- ✓ Implementieren Sie Reload mit `ReloadAsync` oder einem frischen `LoadEditorAsync`, implementieren Sie Overwrite, indem Sie `OriginalValue` auf die Datenbankversion setzen und erneut speichern, und zeigen Sie Overwrite nur, wenn die Rolle des aktuellen Benutzers es erlaubt; aktualisieren Sie nach beiden Pfaden das übergeordnete Grid, damit es der Datenbank entspricht.

- ✓ Führen Sie `dotnet ef migrations script --idempotent` mit Ausgabe nach `artifacts/sql` aus, schreiben Sie die Deployment-Hinweise mit umgebungsspezifischen Connection-Strings aus der Konfiguration oder einem Secret Store, sichern Sie `EnableSensitiveDataLogging` mit der Prüfung auf die Development-Umgebung ab, konfigurieren Sie die Logging-Kategorien, gehen Sie dann das Bewertungsraster des Abschlussprojekts durch und füllen Sie die Checklisten aus.

Geforderte Ergebnisse

- ✓ RowVersion im Edit-Modell mitgeführt und vor dem Speichern als OriginalValue gesetzt
- ✓ Reproduzierbarer Nebenläufigkeitskonflikt, als DbUpdateConcurrencyException abgefangen
- ✓ Konfliktdialog, der vorgeschlagene Werte und Datenbankwerte auflistet, mit den Pfaden Reload und Overwrite
- ✓ Idempotentes SQL-Migrationsskript mit Deployment-Hinweisen und umgebungsspezifischen Connection-Strings
- ✓ Logging sensibler Daten außerhalb der Entwicklung deaktiviert und Review des Abschlussprojekts abgeschlossen

Wiederholungsfragen

- ✓ Was würde der Test mit zwei Sessions zeigen, wenn Sie die Zeile mit `OriginalValue` weglassen, und warum sähe das wie ein Erfolg aus, während die Änderung des ersten Bearbeiters stillschweigend verloren geht?
- ✓ Welche Auflösungspfade würden Sie einem normalen Agent anbieten und welche einem Supervisor, und was muss nach jedem Pfad mit dem Grid und der `RowVersion` im Edit-Modell geschehen?
- ✓ Nach einem Deployment starten drei Produktivinstanzen. Welche davon wendet in Ihrem Entwurf die Migration an und wann, und was würde schiefgehen, wenn `Migrate()` im Startup jeder Instanz liefe?

Abgabe des Abschlussprojekts

Das Lab dieses Moduls ist zugleich die Abgabe des Abschlussprojekts. Die Support Desk Data Console muss zeigen, dass Sie Entitäten mit bidirektionalem Binding, Validierung, asynchronem Laden und Mehrbenutzersicherheit an die UI binden können, und sie muss diese Funktionen enthalten:

- ✓ Ticket-Browser mit Filtern, Paging und projizierten Listeneinträgen ohne Tracking, über eine `BindingSource` an ein `DataGridView` gebunden.
- ✓ Lookup-ComboBoxen für Customer, Agent, Category, Status und Priority.
- ✓ Modales Formular zum Anlegen und Bearbeiten mit Edit-Modell und `BindingSource`, mit expliziter Validierung über `ErrorProvider` und zusammenfassendem Feedback.
- ✓ Asynchrone Lade-, Speicher- und Löschoperationen mit Lade-Schutz.
- ✓ Optimistische Nebenläufigkeit per RowVersion mit Konfliktdialog.
- ✓ Migrationsskript oder Bundle-Plan, Hinweise zur umgebungsspezifischen Konfiguration und eine kurze Grounding-Notiz für den KI-Assistenten, die die umgesetzten Muster erklärt.

Reichen Sie das Quell-Repository oder ein ZIP ein; Screenshots des Grids, des Editors, eines Validierungsfehlers und des Nebenläufigkeitskonflikts; die Migrationsdateien und das erzeugte SQL-Skript für den Produktivbetrieb; eine einseitige Architekturnotiz; und die ausgefüllten Checklisten-Dateien.

Bewertungsbereich	Gewicht	So sieht „exzellent“ aus
Architektur	20 %	Klare Service-Kapselung, Einsatz der Factory, kein gemeinsam genutzter veränderlicher Zustand.
Binding	20 %	BindingSource, DTO oder Edit-Modell, EndEdit und Aktualisierung des Grids – alles korrekt.
Korrektter Einsatz von EF Core	20 %	Async, Tracking im Vergleich zu No-Tracking, Migrationen und Nebenläufigkeit behandelt.
Validierungs-UX	15 %	Feld- und Zusammenfassungsfehler klar und wiederverwendbar.
Performance	15 %	Serverseitige Filterung, Paging und Projektion mit Messnotizen.
Deployment und Diagnose	10 %	Sichere Konfiguration, Logging und ein Migrationsplan.



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.