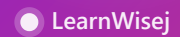


Modellierung, DbContext-Konfiguration und Migrationen

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Entitäten, die auf datengebundene Oberflächen zugeschnitten sind, Fluent API für Schlüssel, Längen, Indizes, Löscherhalten und RowVersion, Migrationen als geprüfter Quellcode mit Skripten und Bundles sowie die Design-Time-Factory. Was das für Wisej.NET-Entwickler bedeutet, die mit echten Datenbanken arbeiten – und wie Sie dieses Modul lesen.



Fokus: Produktivbetrieb

Entitäten, die auf datengebundene Oberflächen zugeschnitten sind, Fluent API für Schlüssel, Längen, Indizes, Löscherhalten und RowVersion, Migrationen als geprüfter Quellcode mit Skripten und Bundles sowie die Design-Time-Factory.

1. Das Problem im Produktivbetrieb

Die Support Desk Data Console zeigt Tickets in einem Grid mit Paging, bearbeitet sie in einem modalen Formular und lässt mehrere Agents gleichzeitig dieselbe Queue abarbeiten – und jede Oberfläche ist nur so gut wie das Modell darunter. Ein Filter ohne passenden Index wird bei fünfzigtausend Tickets zum Tabellenscan, zwei Agents, die dasselbe Ticket speichern, verlieren stillschweigend die Arbeit eines der beiden, wenn die Zeile kein Concurrency-Token trägt, und ein von Hand geflicktes Schema bricht das nächste Deployment. Das Modell ist der Vertrag zwischen UI und Datenbank; deshalb baut dieses Modul es explizit auf und versioniert es als Quellcode.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
bindungsfreundliche Entität	Eine Entität mit vorhersehbarem Schlüssel, stabilen Anzeigefeldern, expliziten Navigationseigenschaften und kleinen Lookup-Tabellen, sodass ein Grid, eine <code>ComboBox</code> und ein Editor alle an sie binden können.
View-Modell TicketListItem	Eine flache Projektion nur der Spalten, die das Grid zeigt, sodass das Grid nie den ganzen Ticket-Graphen erhält und unabhängig von der Unit of Work zum Bearbeiten bleibt.
Konfiguration per Fluent API	Mapping-Regeln in <code>OnModelCreating</code> (Pflicht-Strings, Längen, Indizes, Löschverhalten, Genauigkeit, Standardwerte), sodass Persistenzregeln direkt beim <code>DbContext</code> liegen.
RowVersion als Concurrency-Token	Ein von der Datenbank erzeugtes Byte-Array, gekennzeichnet mit <code>IsRowVersion</code> ; EF Core nimmt seinen Originalwert in die WHERE-Klausel des UPDATE auf, sodass ein veraltetes Speichern null Zeilen ändert und <code>DbUpdateConcurrencyException</code> auslöst.
Löschverhalten	Die <code>OnDelete</code> -Regel einer Beziehung (Restrict, SetNull, Cascade), die entscheidet, ob das Löschen eines Kunden abgelehnt wird, Tickets ihre Zuweisung verlieren oder auch Kommentare entfernt werden.
Migration	Eine C#-Klasse plus Modell-Snapshot, die eine geprüfte Schemaänderung beschreibt, in die Versionskontrolle committet und der Reihe nach angewendet wird, um jede Datenbank auf den Stand des Modells zu bringen.
Design-Time-Factory	Eine <code>IDesignTimeDbContextFactory</code> , die den <code>dotnet ef</code> -Tools sagt, wie sie den Context erzeugen, ohne den Startup der Wisej.NET-Anwendung auszuführen.

3. Der rote Faden der Lektion

Entitäten für Master-Detail-Oberflächen

Datengebundene Oberflächen funktionieren am besten, wenn das Modell vorhersehbare Schlüssel, stabile Anzeigefelder, explizite Beziehungen und kleine Lookup-Tabellen hat. In der Support Desk Data Console trägt die Entität `Ticket` die Eigenschaften `Customer`, `Agent`, `Category`, `Status`, `Priority`, `DueDate`, `Title`, `Description`, `RowVersion`, `CreatedAt` und `UpdatedAt`. `Customer`, `Agent` und `Category` sind kleine Lookup-Entitäten mit jeweils einer `Tickets`-Collection, und `TicketComment` ist die Detail-Collection unter einem Ticket.

`CustomerId` und `CategoryId` sind Pflicht-Integer, weil ein Ticket ohne sie auf keiner Oberfläche Sinn ergibt; `AgentId` ist ein nullable `int?`, weil ein neues Ticket existiert, bevor jemand zugewiesen ist – so kann die Agent-`ComboBox` „Unassigned“ ohne Platzhalter-Zeile anzeigen. Ein Grid braucht nicht jede Eigenschaft; der Kurs definiert daher zusätzlich `TicketListItem`-View-Modelle (Nummer, Titel, Kunde, Status, Fälligkeitsdatum, Agent, letzte Änderung) für das schreibgeschützte Grid, während ein Editor direkt an ein getracktes `Ticket` bindet oder an ein Edit-Modell, das auf die Entität zurückgemappt wird.

Fluent API zuerst für Produktivmodelle

Die Konventionen von EF Core erkennen Schlüssel und Fremdschlüssel anhand von Namen, aber eine Produktivanwendung sollte alles explizit konfigurieren, was Schema und Verhalten beeinflusst: Pflicht-Strings, Maximallängen, die Indizes, die die Suchfilter nutzen, Löschverhalten, Dezimalgenauigkeit, Standardwerte und Concurrency-Token. Data Annotations sind für einfache Projekte in Ordnung; die Fluent API hält Persistenzregeln nahe am `DbContext` und vermeidet, dass UI-Validierung und Datenbank-Mapping vermischt werden. Das Herzstück von `SupportDeskContext.OnModelCreating` :

```
modelBuilder.Entity<Ticket>(b =>
{
    b.Property(x => x.Title).HasMaxLength(180).IsRequired();
    b.Property(x => x.Status).HasMaxLength(30).IsRequired();
    b.Property(x => x.RowVersion).IsRowVersion();
    b.HasIndex(x => new { x.Status, x.DueDate });
    b.HasIndex(x => x.CustomerId);
    b.HasIndex(x => x.UpdatedAt);
    b.HasOne(x => x.Customer).WithMany(c => c.Tickets)
        .HasForeignKey(x => x.CustomerId).OnDelete(DeleteBehavior.Restrict);
    b.HasOne(x => x.Agent).WithMany(a => a.Tickets)
        .HasForeignKey(x => x.AgentId).OnDelete(DeleteBehavior.SetNull);
});
```

Jede Zeile ist eine Entscheidung, von der der Ticket-Browser abhängt. Der zusammengesetzte Index auf `Status` und `DueDate` bedient die Queue, die nach Status filtert und nach Fälligkeitsdatum sortiert, `CustomerId` bedient den Kundenfilter und `UpdatedAt` die Sortierung „zuletzt geändert“. `Restrict` bedeutet, dass ein Kunde oder eine Kategorie mit Tickets nicht versehentlich gelöscht werden kann, `SetNull` bedeutet, dass die Tickets eines ausscheidenden Agents einfach ihre Zuweisung verlieren, und Kommentare verwenden `Cascade`. Eine geldähnliche Spalte bekäme hier `HasPrecision(18, 2)`, und ein Standardstatus „Open“ wird hier gesetzt statt im Formularcode.

Das Concurrency-Token ist in einer Wisej.NET-Anwendung mit mehreren Benutzern am wichtigsten. Ein mit `IsRowVersion` konfiguriertes `byte[] RowVersion` wird zu einer SQL-Server-Spalte vom Typ `rowversion`, die die Datenbank bei jedem Schreibvorgang ändert, und EF Core nimmt den Originalwert in die Update-Bedingung auf: `UPDATE ... WHERE Id = @p0 AND RowVersion = @p1`. Hat ein anderer Benutzer die Zeile nach dem Laden geändert, betrifft das Update null Zeilen, und `SaveChangesAsync` löst `DbUpdateConcurrencyException` aus; die UI kann dann die aktuellen Datenbankwerte anzeigen und Neuladen, Überschreiben oder Zusammenführen anbieten. SQLite kennt keinen Typ `rowversion`: Dort ist die Eigenschaft zwar weiterhin ein Concurrency-Token, aber ein Trigger oder der Code muss sie hochzählen.



Typischer Fehler: Annotationen für alles

Jede Entität mit `[Required]`, `[MaxLength]` und `[Timestamp]` versehen, obwohl die Fluent API klarer wäre – sodass die Entität zugleich Persistenz-Mapping und Validierungsvertrag der UI ist.



Typischer Fehler: kein Concurrency-Token auf Oberflächen mit mehreren Benutzern

`RowVersion` weglassen, weil „das Grid ja schreibgeschützt ist“. Der Editor ist es nicht, und ohne Token überschreibt der zweite Agent, der speichert, stillschweigend den ersten.

Migrationen als geprüfter Quellcode

Migrationen sind Quellcode. Fügen Sie während der Entwicklung bei jeder Modelländerung eine hinzu, mit `dotnet ef migrations add InitialCreate --project SupportDesk.Data --startup-project SupportDesk.Web`, prüfen Sie die generierten Operationen `Up` und `Down` und wenden Sie sie dann mit `dotnet ef database update` an. Das Review deckt eine Umbenennung auf, die als Löschen und Neuanlegen einer Spalte gescaffoldet wurde und Daten verliert, ein `Cascade`, wo Sie `Restrict` gemeint haben, einen fehlenden Index oder eine `rowversion`-Spalte, die nie erschienen ist. Migration und Modell-Snapshot werden gemeinsam committet.

Im Produktivbetrieb ist es anders. Vermeiden Sie in Umgebungen mit mehreren Instanzen ungeprüfte automatische Schemaänderungen beim Start der Anwendung: Zwei gleichzeitig startende Instanzen konkurrieren darum, dieselbe Migration anzuwenden. Bevorzugen Sie ein geprüftes SQL-Skript aus `dotnet ef migrations script --idempotent`, wenn eine Datenbankfreigabe und eine Rollback-Planung erforderlich sind, oder ein Migrations-Bundle aus `dotnet ef migrations bundle` – eine eigenständige ausführbare Datei, die eine Release-Pipeline ausführt, bevor der Traffic umgeschaltet wird.

- Connection-Strings ausgelagert, nie im Quellcode.
- Jede Migration geprüft; Skript oder Bundle erzeugt und mit dem Release aufbewahrt.
- Das Risiko von Migrationen beim Start mehrerer Instanzen schriftlich behandelt.

Die `dotnet ef`-Tools müssen `SupportDeskContext` zur Design-Zeit erzeugen können, und eine Design-Time-Factory ist der expliziteste Weg, das zu ermöglichen. `SupportDeskDesignTimeFactory` implementiert `IDesignTimeDbContextFactory<SupportDeskContext>`; ihr `CreateDbContext` baut die Optionen mit dem Provider der Anwendung und einer sicheren lokalen Verbindung auf (`SupportDeskDev` auf LocalDB, Trusted Connection, kein Passwort) und gibt einen neuen Context zurück. Das ist wichtig, weil der Wisej.NET-Startup Setup-Code für das UI-Framework enthält, der beim Scaffolding von Migrationen nicht laufen sollte; mit der Factory berühren die Tools ihn nie.



Typischer Fehler: Migrationen als generierte Dateien, die nie geprüft werden müssen

Den Ordner Migrations wie Build-Ausgabe behandeln. Eine Migration ist eine Schemaänderung, die jemand gegen die Produktivumgebung ausführen wird; prüfen Sie sie wie jeden anderen Commit.

Schema-Constraints sind keine UI-Validierung

`HasMaxLength(180).IsRequired()` auf `Title` erzeugt eine Spalte `NOT NULL nvarchar(180)`, und eine Verletzung erzeugt nach einem Roundtrip eine `DbUpdateException` mit einer Provider-Meldung, die kein Benutzer lesen sollte. Das ist die letzte Sicherheitsebene, nicht die erste. Das verständliche „Enter a title“ gehört zum `ErrorProvider` des Editors und zum `TicketValidator`, den Modul 5 hinzufügt – ebenso wie „geschlossene Tickets dürfen kein Fälligkeitsdatum in der Zukunft haben“, was keine Spalten-Constraint ausdrücken kann. Halten Sie die Ebenen bewusst getrennt: Die Fluent API deklariert, was die Datenbank

garantieren muss (Nullability, Länge, den eindeutigen Index auf der Ticketnummer, Fremdschlüssel, das Concurrency-Token); `DataAnnotations` und Validatoren auf dem Edit-Modell deklarieren, was dem Benutzer mitgeteilt werden muss.

Seed-Daten folgen derselben Disziplin. Feste Lookup-Zeilen wie Kategorien können über `HasData` laufen, damit sie mit der Migration mitreisen; die fünfzig Tickets erzeugt besser eine nur in der Entwicklung genutzte Seeding-Methode, die einen Context aus der Factory erzeugt, prüft, ob bereits Tickets existieren, die Zeilen hinzufügt, `SaveChangesAsync` aufruft und den Context freigibt. Hinter einem Button „Seed development data“ mit Lade-Schutz und `try / catch` ist sie der Ort, an dem die Zählabfrage aus Modul 1 zum ersten Mal mehr als null liefert.



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:

- ✓ Entitäten und Beziehungen für Master-Detail-Oberflächen modellieren.
- ✓ die Fluent API für Schlüssel, Indizes, Pflichtfelder, Länge, Dezimalgenauigkeit, Löscherhalten und Row-Version einsetzen.
- ✓ Migrationen erstellen und verstehen, wann Skripte, Bundles und Design-Time-Factories zum Einsatz kommen.
- ✓ Constraints des Datenbankschemas von UI- und Domänenvalidierung trennen.

Lab- und Prüfungsleitfaden

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



Praxis-Lab

Erstellen Sie die Entitäten Customer, Agent, Category, Ticket und TicketComment für die Support Desk Data Console, konfigurieren Sie Beziehungen und Löscherhalten mit der Fluent API, konfigurieren Sie RowVersion mit IsRowVersion, fügen Sie Indizes für Status, DueDate, CustomerId und UpdatedAt hinzu, ergänzen Sie eine Design-Time-Factory, erstellen Sie die Migration InitialCreate und befüllen Sie eine lokale SQLite- oder SQL-Server-LocalDB-Datenbank mit fünf Kunden, drei Agents, Kategorien und mindestens fünfzig Tickets.

Ausgangspunkt

Öffnen Sie die Projektmappe Support Desk Data Console aus Modul 1: `SupportDesk.Web` (das Wisej.NET-Projekt), `SupportDesk.Data` mit einem leeren `SupportDeskContext`, registriert über `AddDbContextFactory`, sowie `SupportDesk.Services` und `SupportDesk.Tests`. Die Page aus Modul 1 führt ihre asynchrone Zählabfrage noch gegen eine Datenbank ohne Tabellen aus; das Erste, was Sie nach diesem Lab sehen, ist also, wie aus dieser Zählung eine echte Zahl wird.

Vorgeschlagenes Vorgehen

- ✓ Schreiben Sie zuerst die fünf Entitätsklassen in `SupportDesk.Data`, mit Integer-Schlüsseln, Navigationseigenschaften in beide Richtungen, einer nullable `AgentId` und einem `byte[] RowVersion` auf `Ticket`; halten Sie sie frei von Mapping-Attributen.
- ✓ Konfigurieren Sie alles, was das Schema betrifft, in `OnModelCreating`: Pflicht-Strings und Längen, `IsRowVersion`, die vier geplanten Indizes und `onDelete` pro Beziehung (Restrict für Customer und Category, SetNull für Agent, Cascade für Kommentare).
- ✓ Fügen Sie `SupportDeskDesignTimeFactory` mit demselben Provider wie die Anwendung und einer lokalen Verbindung ohne Passwort hinzu und führen Sie dann im Stammverzeichnis der Projektmappe `dotnet ef migrations add InitialCreate` aus.
- ✓ Lesen Sie die generierte Migration, bevor Sie sie anwenden: Prüfen Sie die `rowversion`-Spalte, die Indexnamen, die `ON DELETE`-Klauseln und den Snapshot, und führen Sie dann `dotnet ef database update` aus.
- ✓ Schreiben Sie eine nur in der Entwicklung genutzte Seeding-Methode, die einen Context aus der Factory erzeugt, prüft, ob bereits Tickets existieren, Kunden, Agents, Kategorien und Tickets hinzufügt, speichert und freigibt, und rufen Sie sie aus einem Seed-Button hinter einem Lade-Schutz und einem `try / catch` auf.

Geforderte Ergebnisse

- ✓ Entitäten Customer, Agent, Category, Ticket und TicketComment mit Beziehungen und Löscherhalten
- ✓ RowVersion mit IsRowVersion konfiguriert

- ✓ Indizes für Status, DueDate, CustomerId und UpdatedAt
- ✓ Design-Time-Factory und die Migration InitialCreate
- ✓ Seed-Daten: fünf Kunden, drei Agents, Kategorien und mindestens fünfzig Tickets

Wiederholungsfragen

- ✓ Sie löschen einen Kunden, der noch offene Tickets hat. Was passiert mit dem von Ihnen konfigurierten Löschverhalten, und was wäre mit Cascade passiert?
- ✓ Zwei Agents laden das Ticket SD-1042 und klicken im Abstand von einer Sekunde auf Save. Welches Speichern löst eine Exception aus, was steht in der WHERE-Klausel der UPDATE-Anweisung, und was sieht der Benutzer?
- ✓ Warum liefert `IsRequired()` auf `Title` dem Benutzer keine Meldung „Enter a title“, und woher kommt diese Meldung später im Kurs?



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.