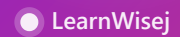


Asynchrones Laden, verknüpfte Daten, Filterung und Performance

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Die Entscheidungstabelle für verknüpfte Daten (Projektion, Include, gefiltertes Include, explizites Laden, kein Lazy Loading in Grids), Tracking im Vergleich zu No-Tracking, Paging und indizierte Filter, asynchrone Koordination mit Ladezuständen und Hintergrund-Tasks sowie Messen vor dem Optimieren. Was das für Wisej.NET-Entwickler bedeutet, die mit echten Datenbanken arbeiten – und wie Sie dieses Modul lesen.



Fokus: Produktivbetrieb

Die Entscheidungstabelle für verknüpfte Daten (Projektion, Include, gefiltertes Include, explizites Laden, kein Lazy Loading in Grids), Tracking im Vergleich zu No-Tracking, Paging und indizierte Filter, asynchrone Koordination mit Ladezuständen und Hintergrund-Tasks sowie Messen vor dem Optimieren.

1. Das Problem im Produktivbetrieb

Der Ticket-Browser des Support Desk hat in Modul 3 mit fünfzig Seed-Tickets funktioniert. Mit fünfzigtausend wird er es nicht mehr, es sei denn, die Abfrage erledigt weniger Arbeit an der richtigen Stelle. Das Versagen ist leise: ein Grid, das drei Sekunden braucht, weil Formatierungscode pro Zeile eine Navigationseigenschaft angefasst hat, eine Session, die Tausende getrackte Entitäten hält, die niemand bearbeiten wird, eine Suche, die eine Tabelle scannt. In diesem Modul geht es darum, bewusst weniger Arbeit zu erledigen, und zwar in SQL – und das mit einem Log zu belegen statt mit einem Bauchgefühl.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
Projektion	Ein <code>Select</code> , das genau die Felder liefert, die die Oberfläche braucht, etwa <code>TicketListItem</code> , sodass die Datenbank die verknüpften Namen in einer einzigen Anweisung joint.
Include und gefiltertes Include	Eager Loading verknüpfter Entitäten zusammen mit der Wurzel; ein gefiltertes <code>Include</code> grenzt eine Collection ein (die letzten fünf Kommentare), sodass ein Editor ein überschaubares Aggregat lädt.
explizites Laden	Eine Navigation später bei Bedarf laden, über <code>Collection</code> oder <code>Reference</code> des Entry und <code>LoadAsync</code> , wenn der Benutzer einen Detailbereich aufklappt.
Lazy Loading und N+1	Navigationseigenschaften, die beim ersten Zugriff transparent geladen werden; in einem Grid ist das eine versteckte Abfrage pro Zeile oder pro Navigationszugriff.
AsNoTracking und Identity Resolution	Ergebnisse, die für schreibgeschützte Oberflächen am Change Tracker vorbeigehen; <code>AsNoTrackingWithIdentityResolution</code> liefert trotzdem eine Instanz pro Entität, wenn dieselbe Zeile mehrfach vorkommt.
Paging auf Abfrageebene	<code>Where</code> , <code>OrderBy</code> , <code>Skip</code> , <code>Take</code> und <code>Select</code> , zusammengesetzt vor <code>ToListAsync</code> , mit einem separaten <code>CountAsync</code> für die Gesamtzahl.
EF-Core-Logging	Das erzeugte SQL jedes Befehls mit seiner Dauer, über <code>LogTo</code> oder <code>Microsoft.Extensions.Logging</code> .

3. Der rote Faden der Lektion

Woher das langsame Grid kommt

Beginnen Sie mit dem absichtlich langsamen Ticket-Browser, den das Lab vor der Korrektur misst. Sein `SearchTicketsAsync` lädt getrackte `Ticket`-Entitäten ohne Paging, und `CellFormatting` liest `ticket.Customer.Name`, `ticket.Agent?.DisplayName` und `ticket.Category.Name`. Mit aktiviertem Lazy Loading führt jeder erste Zugriff auf eine nicht geladene Navigation eine eigene Abfrage aus: Fünfzig Zeilen kosten eine Anweisung für die Tickets plus hundertfünfzig für die Namen. Nichts löst eine Exception aus; nur das Log zeigt es.

Die Regel, die das beseitigt: Anzeige-Code darf nie eine Abfrage auslösen. Zeilen, die über die `BindingSource` an ein Wisej.NET- `DataGridView` gebunden sind, sind `TicketListItem`-Records, die die angezeigten Namen bereits mitbringen, und `CellFormatting` formatiert nur. Dieselbe Regel gilt für Baum- und Listenansichten: Ein `TreeView` lädt die Kinder eines Knotens einmal, wenn er aufgeklappt wird, niemals durch das Ablaufen von Navigationseigenschaften pro Knoten.



Typischer Fehler: Lazy Loading aus Zeilenformatierung oder Anzeige-Code

Eine Navigationseigenschaft, die in `CellFormatting` oder beim Zeichnen eines Baumknotens gelesen wird, ist eine getarnte Abfrage, die einmal pro Zeile läuft. Liefern Sie die Namen stattdessen in der Projektion mit.

Die Entscheidungstabelle für verknüpfte Daten

EF Core bietet vier Wege, verknüpfte Daten auf eine Oberfläche zu bringen; die Wahl hängt davon ab, was die Oberfläche damit tun wird.

- Verwenden Sie **Projektion**, wenn die UI ein flaches Lesemodell braucht: den Ticket-Browser, Dashboards, Exporte.
- Verwenden Sie **Include**, wenn ein Editor ein vollständiges Aggregat braucht und dessen Größe überschaubar ist: ein Ticket mit Kunde, Kategorie und Agent.
- Verwenden Sie ein **gefiltertes Include**, wenn nur eine kleine Teilmenge einer Collection benötigt wird: die letzten fünf öffentlichen Kommentare.
- Verwenden Sie **explizites Laden**, wenn der Benutzer einen Detailbereich bei Bedarf aufklappt: den Kommentarverlauf hinter einem eingeklappten Panel.
- Verlassen Sie sich in Grids, Listen oder Bäumen nie auf **Lazy Loading**: Es erzeugt stillschweigend eine Abfrage pro Zeile oder pro Navigationszugriff.

Im Support Desk projiziert der Browser auf `TicketListItem`; `LoadEditorAsync` im Editor verwendet `Include` für die drei Referenzen und ein gefiltertes `Include` für die Kommentare, die er als Vorschau zeigt; das Kommentar-Panel lädt den vollständigen Verlauf mit `db.Entry(ticket).Collection(t => t.Comments).LoadAsync()` im eigenen Context des Editors.

Tracking im Vergleich zu No-Tracking

Tracking ist richtig, wenn derselbe Context Änderungen an denselben Entitätsinstanzen speichern wird – so wie der Editor, wenn `SaveChangesAsync` die Differenz schreibt. No-Tracking ist richtig für schreibgeschützte Grids, Dashboards, Lookup-Listen, Exporte und Autocomplete-Ergebnisse.

`AsNoTracking` überspringt den Change Tracker: weniger Speicher pro Session, schnellere Materialisierung und kein versehentliches Update von einer Oberfläche, die nur liest.

`AsNoTrackingWithIdentityResolution` liegt zwischen beiden. Eine Abfrage ohne Tracking, die Entitäten liefert, kann für jedes Ticket desselben Kunden ein eigenes `Customer`-Objekt zurückgeben; Identity Resolution liefert eine Instanz pro Schlüssel. Verwenden Sie es, wenn eine schreibgeschützte Oberfläche einen Objektgraphen aufbaut; verzichten Sie darauf, wenn Sie auf flache Zeilen projizieren.



Typischer Fehler: Entitäten ohne Tracking verwenden und später eine automatische Erkennung von Änderungen erwarten

Wer ein `Ticket` aus einer `AsNoTracking`-Abfrage oder ein `TicketListItem` aus dem Grid bearbeitet und dann `SaveChangesAsync` aufruft, ändert nichts. Der Editor lädt in seinem eigenen Context eine frische, getrackte Entität über den Schlüssel und mappt die freigegebenen Werte darauf.

Paging und indizierte Filter auf Abfrageebene

Ein Wisej.NET-Grid sollte nicht 100.000 Zeilen laden, nur weil es sie anzeigen kann. Setzen Sie Filter, Sortierung, `Skip`, `Take` und `Select` auf Abfrageebene zusammen, damit die Datenbank die Arbeit erledigt, und liefern Sie eine Gesamtzahl, wenn die UI Seitenzahlen anzeigt – aus einem separaten `CountAsync` auf der gefilterten Abfrage:

```
var total = await query.CountAsync(token);

var page = await query
    .AsNoTracking()
    .OrderByDescending(t => t.UpdatedAt)
    .Skip(criteria.PageIndex * 50)
    .Take(50)
    .Select(t => new TicketListItem(
        t.Id, t.Number, t.Title, t.Customer.Name,
        t.Agent == null ? null : t.Agent.DisplayName,
        t.Category.Name, t.Status, t.Priority, t.DueDate, t.UpdatedAt))
    .ToListAsync(token);
```

Die Indizes aus Modul 2 auf `Status`, `DueDate`, `CustomerId` und `UpdatedAt` passen zu diesen Filtern und dieser Sortierung. Halten Sie die Textsuche indexfreundlich: Ein `StartsWith` auf der Ticketnummer wird zu einem Präfix-`LIKE`, das ein Index bedienen kann, während ein `Contains` auf dem Titel zu einem führenden Platzhalter wird, der scannt. Zeigt das Log einen häufigen Filter ohne Index, fügen Sie einen hinzu und messen Sie erneut.

Asynchrone Koordination mit Ladezuständen und Hintergrund-Tasks

EF Core bietet für jede I/O-Operation async-Methoden. Warten Sie jede ab, bevor Sie die nächste auf demselben Context starten, denn ein `DbContext` unterstützt keine parallelen Operationen; die beiden `await`-Aufrufe oben laufen bewusst nacheinander. In Wisej.NET gibt ein asynchroner Event-Handler die Kontrolle an den Browser zurück, während die Abfrage läuft – deshalb ist der Lade-Schutz wichtig: Zeigen Sie mit `ShowLoader` des Grids oder einem Status-`Label` einen Ladezustand, deaktivieren Sie die Buttons Search, Next und Previous, die konkurrierende Arbeit starten würden, und aktivieren Sie sie in einem `finally`-Block wieder.

Lang laufende Import-, Export- und Reporting-Jobs gehören nicht in einen Event-Handler. Starten Sie sie mit `Application.StartTask`, erzeugen Sie den `DbContext` innerhalb des Tasks aus der Factory, statt den Context oder die Entitäten der Page zu übernehmen, und übertragen Sie den Fortschritt nur über `Application.Update`, den kontextbewussten Update-Mechanismus von Wisej.NET. Ein roher Thread, der ein Control anfasst, hat keinen Session-Kontext, und das übergebene Cancellation-Token ist der Weg, über den ein Job stoppt, wenn die Session endet.

Messen vor dem Optimieren

Raten Sie nicht. Aktivieren Sie das EF-Core-Logging in der Entwicklung mit `LogTo` oder `Microsoft.Extensions.Logging`, lesen Sie SQL und Dauer jedes Befehls und schalten Sie `EnableSensitiveDataLogging` nur auf einem Entwicklerrechner ein, da es Parameterwerte ausgibt. Prüfen

Sie bei einer langsamen Anweisung den Ausführungsplan der Datenbank, in dem ein fehlender Index oder ein Platzhalter-Scan sichtbar wird. Das Vorher-nachher-Log im Lab macht diese Disziplin greifbar.

Erst wenn ein Profiling den Engpass zeigt, rechtfertigen die spezialisierten Werkzeuge ihre Komplexität. Kompilierte Abfragen helfen, wenn der Kompilierungsaufwand bei einer häufig ausgeführten Abfrage messbar ist. Context-Pooling mit `AddPooledDbContextFactory` hilft, wenn die Erzeugung des Contexts dominiert, und erfordert Sorgfalt, wenn der Context-Zustand pro Benutzer oder Mandant variiert, da ein gepoolter Context zurückgesetzt und wiederverwendet wird. Split Queries helfen, wenn mehrere Collection- `Include` s die Zeilen vervielfachen. Rohes SQL ist das letzte Mittel. Jedes davon ist eine Antwort auf eine Zahl im Log, nie eine Voreinstellung.



Typischer Fehler: optimieren ohne zu messen

Kompilierte Abfragen, Pooling oder rohes SQL auf einer Oberfläche ohne Profiling tauschen Lesbarkeit gegen eine Verbesserung, die es vielleicht gar nicht gibt. Messen, eine Sache ändern, erneut messen.



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:

- ✓ N+1-Verhalten durch Lazy Loading in Grids sowie Baum- und Listenansichten vermeiden.
- ✓ Include, gefiltertes Include, explizites Laden und Projektion gezielt einsetzen.
- ✓ AsNoTracking, AsNoTrackingWithIdentityResolution, Paging, Indizes und gemessene Performance-Verbesserungen nutzen.
- ✓ asynchrone EF-Core-Operationen mit Ladeanzeigen und Hintergrund-Tasks von Wisej.NET koordinieren.

Lab- und Prüfungsleitfaden

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



Praxis-Lab

Optimieren Sie den Ticket-Browser des Support Desk: Aktivieren Sie das EF-Core-Logging in der Entwicklung, protokollieren Sie SQL und Laufzeit der nicht optimierten Suche, ersetzen Sie das Laden von Entitäten durch Projektion, ergänzen Sie AsNoTracking bei Grid-Abfragen, führen Sie Paging mit einer Seitengröße von 50 ein, ersetzen Sie Lazy-Navigation-Zugriffe durch Projektion oder Include und dokumentieren Sie die gemessenen Vorher-nachher-Ergebnisse in Ihren Lab-Notizen.

Ausgangspunkt

Öffnen Sie die Projektmappe Support Desk Data Console aus Modul 5 und die Ticket-Browser-Page aus Modul 3. Beginnen Sie dieses Lab mit einem absichtlich naiven Zweig von `TicketQueryService.SearchTicketsAsync`: einem, der getrackte `Ticket`-Entitäten ohne Paging lädt und `CellFormatting` des Grids die Namen von Kunde, Agent und Kategorie über Navigationseigenschaften lesen lässt – so gibt es ein echtes N+1, das Sie messen können, bevor Sie es beseitigen.

Vorgeschlagenes Vorgehen

- ✓ Aktivieren Sie das EF-Core-Logging in der Entwicklung mit `LogTo` oder `Microsoft.Extensions.Logging` auf `SupportDeskContext`, lassen Sie `EnableSensitiveDataLogging` hinter einer Prüfung auf die Entwicklungsumgebung und führen Sie die naive Suche einmal aus; zählen Sie die Anweisungen, notieren Sie die Millisekunden und fügen Sie das SQL in Ihre Lab-Notizen ein.
- ✓ Schreiben Sie die Grid-Abfrage als Projektion ohne Tracking neu: `AsNoTracking`, die zusammengesetzten `Where`-Filter, `OrderByDescending(t => t.UpdatedAt)`, dann `Select` in `TicketListItem` mit den von der Datenbank gejointen Namen von Kunde, Agent und Kategorie, abgeschlossen mit `ToListAsync`.
- ✓ Führen Sie Paging mit einer Seitengröße von 50 ein: ein `CountAsync` auf der gefilterten Abfrage für die Gesamtzahl, dann `Skip` und `Take` vor der Projektion; stellen Sie sicher, dass jeder Filter und die Sortierung die Indizes auf Status, DueDate, CustomerId und UpdatedAt nutzen, und belassen Sie die Suche nach Ticketnummer als Präfixsuche.
- ✓ Entfernen Sie jeden Navigationszugriff aus dem Anzeige-Code: `CellFormatting` formatiert nur die Felder, die bereits in der Zeile stehen, und alle verknüpften Daten, die der Editor braucht, werden mit `Include`, einem gefilterten `Include` oder explizitem Laden im eigenen Context des Editors geladen.
- ✓ Halten Sie den Async-Schutz ehrlich: Warten Sie jede Datenbankoperation ab, bevor die nächste auf demselben Context startet, zeigen Sie den Ladezustand an und deaktivieren Sie Search, Next Page und Previous Page, während eine Seite lädt, aktivieren Sie sie in `finally` wieder, führen Sie dann die optimierte Suche aus und tragen Sie die Nachher-Werte neben den Vorher-Werten ein.

Geforderte Ergebnisse

- ✓ EF-Core-Logging in der Entwicklung aktiviert, mit protokolliertem SQL und protokollierter Laufzeit der nicht optimierten Abfrage
- ✓ Grid-Abfrage als Projektion ohne Tracking neu geschrieben
- ✓ Paging mit einer Seitengröße von 50 und indizierten Filtern
- ✓ Kein N+1-Verhalten durch Lazy Loading mehr im Anzeige-Code
- ✓ Lab-Notizen, die die gemessene Verbesserung vorher und nachher dokumentieren

Wiederholungsfragen

- ✓ Ihr Log zeigte 151 Anweisungen für eine Oberfläche mit 50 Tickets. Welche Codezeile hat die 150 zusätzlichen verursacht, und warum hat Sie nichts in der Anwendung gewarnt?
- ✓ Der Editor braucht das Ticket, seinen Kunden und seine Kategorie sowie die letzten fünf Kommentare; das Kommentar-Panel öffnet den vollständigen Verlauf bei Bedarf. Welche Ladestrategie bedient jeden dieser drei Bedarfe, und warum ist Projektion für den Editor die falsche Antwort?
- ✓ Ein Teamkollege schlägt für den Ticket-Browser kompilierte Abfragen und Context-Pooling vor, bevor er sich das Log angesehen hat. Was würden Sie zuerst messen, und was müssten die Zahlen zeigen, damit sich eine der beiden Änderungen lohnt?



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.