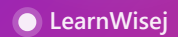


Echte Datenarchitektur: EF Core, Transaktionen, Repositorys und Commands

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Datenzugriff im Produktivbetrieb, Transaktionen, Kapselung durch Services, Trennung von Commands und Queries. Was das für ein Wisej.NET-Team auf Expertenniveau bedeutet – und wie Sie dieses Modul lesen.



Fokus: Produktivbetrieb

Datenzugriff im Produktivbetrieb, Transaktionen, Kapselung durch Services, Trennung von Commands und Queries.

1. Das Problem im Produktivbetrieb

Labs auf mittlerem Niveau können mit simulierten Services arbeiten. Anspruchsvolle Anwendungen müssen sich mit echten Datenspeichern verbinden, ohne Bildschirme in SQL-Clients zu verwandeln oder Persistenzbelange in die UI durchsickern zu lassen.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
EF-Core-Abgrenzung	Die Linie zwischen UI-Code und Entity Framework Core. Bildschirme rufen Services und Commands auf; sie halten nie einen DbContext und schreiben kein LINQ gegen Entitäten.
DbContext-Lebensdauer	Wie lange eine DbContext-Instanz lebt. Kurze Contexts pro Operation vermeiden veraltetes Tracking, wachsenden Speicherverbrauch und sessionübergreifende Leaks in einer langlebigen App.
Repository vs. Query-Service	Repositories laden und speichern Aggregate für Commands, während Query-Services schreibgeschützte, auf Bildschirme zugeschnittene Projektionen liefern. Die Trennung hält beide einfach.
Transaktionsgrenze	Die Einheit von Datenbankarbeit, die gemeinsam gelingen oder scheitern muss. Sie gehört in einen Service oder Command, nicht verteilt auf Event-Handler der UI.
Command-Objekt	Eine kleine Klasse, die die Eingaben für eine Geschäftsoperation transportiert und von einem Handler ausgeführt wird, sodass sich die Operation unabhängig validieren, protokollieren und testen lässt.
Unit of Work	Koordiniert alle Änderungen einer Operation und committet sie gemeinsam – ein einziger Ort für die Verwaltung von Transaktionen und Change Tracking.
Fehler-Mapping	Die Übersetzung von Datenbankausnahmen wie Constraint-Verletzungen und Nebenläufigkeitsfehlern in aussagekräftige Anwendungsfehler, die die UI gefahrlos anzeigen kann.

3. Der rote Faden der Lektion

Die UI sollte die Datenbank nicht kennen

Eine Wisej.NET-Form kann reichhaltig und zustandsbehaftet sein, ohne zu wissen, wie Datensätze gespeichert werden. Anspruchsvoller Code übersetzt die Absicht der UI in Commands und überlässt es den Anwendungsservices, das Ergebnis zu validieren, zu autorisieren, zu persistieren und zu auditieren.

Wählen Sie die einfachste Datengrenze, die Änderungen übersteht

Repositories, Query-Services und Command-Handler sind keine Pflichtzeremonie. Sie sind nützlich, wenn sie eine Datenbank isolieren, Tests vereinfachen oder verhindern, dass UI-Code Abfragen überall verteilt. Setzen Sie Grenzen ein, weil sie ein konkretes Problem lösen.

Eine Datengrenze in der richtigen Größe für die Anwendung

Eine Geschäftsanwendung braucht einen klar definierten Ort, an dem die Datenbank beginnt. Auf der einen Seite stehen Forms, Pages und die Anwendungsservices, die sie aufrufen. Auf der anderen Seite stehen Contexts, Entitäten und Abfragen von Entity Framework Core. Das Abgrenzungsdiagramm des Labs zieht diese Linie und listet auf, was sie überquert: hineingehende Commands, herauskommende Ergebnisse

und Lesemodelle – und sonst nichts. Keine Form hält einen DbContext, und keine Entität wird an ein Grid gebunden.

Die Form der Grenze hängt von der Größe der Anwendung ab. Eine kleine Anwendung kann Abfragen und Commands in einigen wenigen Anwendungsservices unterbringen, die EF Core direkt verwenden. Eine größere trennt Repositories, die Aggregate für Commands laden und speichern, von Query-Services, die schreibgeschützte Projektionen für Bildschirme liefern. Die Regel lautet: Fügen Sie eine Schicht hinzu, wenn sie etwas isoliert, das sich ändert, einen Test vereinfacht oder verhindert, dass sich Abfragen im Code ausbreiten – und nicht früher.

Wie auch immer die Grenze aussieht, sie wird einmal dokumentiert und im Review durchgesetzt. Ein Pull Request, der einer Form eine LINQ-Abfrage hinzufügt, wird sofort abgelehnt.

DbContext-Lebensdauer, Transaktionen und Commands

Ein DbContext ist eine Unit of Work mit Change Tracker, und seine Lebensdauer sollte zu der Operation passen, der er dient. In einer langlebigen Wisej.NET-Session lebt ein in einem Feld der Form gespeicherter Context stundenlang, sammelt getrackte Entitäten an, liefert veraltete Daten und kann nicht gefahrlos mit einem Hintergrund-Task geteilt werden. Die Entscheidung zur Lebensdauer im Lab fällt deshalb kurz aus: ein Context pro Command oder pro Query, erzeugt beim Start der Operation und entsorgt an ihrem Ende.

Eine Transaktionsgrenze umschließt die Schreibvorgänge, die gemeinsam gelingen oder scheitern müssen. Das Genehmigen eines Arbeitsauftrags aktualisiert den Datensatz, schreibt einen Audit-Eintrag und stellt eine Benachrichtigung in die Warteschlange; die ersten beiden gehören in eine Transaktion, die dritte wird erst nach dem Commit eingereiht. Diese Grenze liegt im Command-Handler, wo die gesamte Operation sichtbar ist – nicht in der Form und nicht im Repository.

Ein Command-Objekt wie `ApproveWorkOrderCommand` transportiert die Eingaben für diese Operation und wird von einem Handler ausgeführt, der ein `CommandResult` zurückgibt. Da der Command ein einfaches Objekt ist, lässt sich der Handler mit einer In-Memory-Datenbank und ganz ohne Form testen – genau das ist die erste Wiederholungsfrage des Labs.

Schlanke Handler, echte Services

Der Click-Handler des Approve-Buttons im Lab besteht aus einer Zeile: Er baut aus dem aktuellen Context und dem ausgewählten Arbeitsauftrag einen Command, übergibt ihn dem Service und zeigt das Ergebnis an. Das ist die Zielform für jeden Event-Handler der UI. Der Handler übersetzt eine Geste in eine Absicht und zeigt ein Ergebnis an. Er enthält keine Validierung über das hinaus, was der Bildschirm zum Aktivieren eines Buttons braucht, keine Abfrage, keine Transaktion und kein Wissen darüber, was eine Datenbank ist.

Alles andere wandert in Services und Handler, die sich reviewen lassen, ohne den Designer zu öffnen. Ein Validator prüft den Command gegen Geschäftsregeln. Ein Repository lädt das Aggregat. Der Handler führt den Zustandsübergang aus und speichert. Ein Audit-Writer hält fest, was passiert ist. Jedes Teil lässt sich isoliert testen, von einem Importjob oder einer WebMethod wiederverwenden und an einer Stelle absichern.

Der Test für einen Handler lautet: Könnte ihn jemand löschen und neu schreiben, der die Datenbank nie gesehen hat? Falls nicht, ist Persistenz in die UI durchgesickert.

Datenbankfehler in hilfreiche Meldungen übersetzen

Die Datenbank meldet Fehler in ihrem eigenen Vokabular: eine Verletzung eines Unique-Constraints, ein Fremdschlüsselfehler, eine Nebenläufigkeitsausnahme, ein Timeout, ein Verbindungsabbruch. Nichts davon sollte den Benutzer als Text erreichen, und nichts davon sollte als roher Stacktrace im Audit-Log landen. Die Tabelle für das Fehler-Mapping im Lab listet jeden Datenbankfehler auf, den die Anwendung erzeugen kann, und ordnet ihm drei Dinge zu: einen benannten Ergebniscode, eine Benutzermeldung und einen Audit-Eintrag.

Eine doppelte Arbeitsauftragsnummer wird zu einem Ergebnis, das besagt, dass die Nummer bereits vergeben ist, neben dem Feld angezeigt und als Validierungsfehler protokolliert. Eine Nebenläufigkeitsausnahme wird zum Konfliktablauf aus Modul 3. Ein Timeout wird zu einer Meldung, dass die Operation nicht abgeschlossen werden konnte, mit einer Korrelations-ID für den Support, und zu einem Log-Eintrag mit Abfragenname und Dauer. Der Benutzer bekommt etwas, mit dem er handeln kann; der Support bekommt etwas, wonach er suchen kann.

Das Mapping geschieht im Command-Handler, an der Grenze, damit jeder Aufrufer dieselben Ergebniscodes erhält – ob Form, Stapelverarbeitung oder Job.



Expertenregel für den Produktivbetrieb

Hören Sie nicht auf, sobald der Bildschirm korrekt aussieht. Erklären Sie, wem der Zustand gehört, welche Sicherheitsfolgen es gibt, welche Fehlerpfade existieren und wie sich das System im Produktivbetrieb verhält, bevor Sie das Modul als abgeschlossen betrachten.



Lernziele

Am Ende dieses Moduls können Sie:

- ✓ eine Datenzugriffsgrenze entwerfen, die zu Wisej.NET-Geschäftsanwendungen passt.
- ✓ DbContext-Lebensdauer, Transaktionen und Command-Objekte sicher einsetzen.
- ✓ Persistenzlogik aus den Event-Handlern der UI heraushalten.
- ✓ Datenbankfehler in hilfreiche Meldungen für Benutzer und Audit übersetzen.

Lab- und Prüfungsleitfaden

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



Praxis-Lab

Ersetzen Sie die simulierte Persistenz des WorkOrderService durch eine Implementierung auf Basis von EF Core oder durch eine praxisnahe Repository-Abstraktion. Implementieren Sie die Operationen Create, Update, Approve, Search und Audit mit Transaktionsgrenzen und gemappten Fehlern.

Vorgeschlagenes Vorgehen

- ✓ Lesen Sie das Problem im Produktivbetrieb und benennen Sie das Architekturrisiko, bevor Sie die UI anfassen.
- ✓ Legen Sie zuerst die betroffene Grenze für Service, Command, Modell oder Komponente an oder aktualisieren Sie sie.
- ✓ Erstellen Sie den Wisej.NET-Bildschirm oder die Komponente nach den etablierten Projektstandards.
- ✓ Demonstrieren Sie auch einen Fehlerpfad, nicht nur den Happy Path.
- ✓ Halten Sie die Änderungen im Lernprotokoll fest und aktualisieren Sie die zugehörige Checkliste.

Geforderte Ergebnisse

- ✓ Diagramm der Datenzugriffsgrenze
- ✓ Entscheidung zur DbContext-Lebensdauer
- ✓ Command- und Result-Klassen
- ✓ Transaktionsbeispiel
- ✓ Tabelle für das Fehler-Mapping

Wiederholungsfragen

- ✓ Lässt sich ein Speichervorgang testen, ohne eine Form zu erzeugen?
- ✓ Was liegt innerhalb der Transaktion?
- ✓ Wie werden Nebenläufigkeitsfehler und Verletzungen eindeutiger Schlüssel sichtbar gemacht?



Expertenregel für den Produktivbetrieb

Hören Sie nicht auf, sobald der Bildschirm korrekt aussieht. Erklären Sie, wem der Zustand gehört, welche Sicherheitsfolgen es gibt, welche Fehlerpfade existieren und wie sich das System im Produktivbetrieb verhält, bevor Sie das Modul als abgeschlossen betrachten.