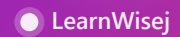


Architektur: EF Core in einer Wisej.NET-Anwendung

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Wo EF Core in einer zustandsbehafteten Wisej.NET-App hingehört: die Lebensdauern von Session, UI, Request und Unit of Work, eine DbContextFactory mit einem Context pro Operation, die Anbindung an die Microsoft-DI und die Regeln für Session-Sicherheit. Was das für Wisej.NET-Entwickler bedeutet, die mit echten Datenbanken arbeiten – und wie Sie dieses Modul lesen.



Fokus: Produktivbetrieb

Wo EF Core in einer zustandsbehafteten Wisej.NET-App hingehört: die Lebensdauern von Session, UI, Request und Unit of Work, eine DbContextFactory mit einem Context pro Operation, die Anbindung an die Microsoft-DI und die Regeln für Session-Sicherheit.

1. Das Problem im Produktivbetrieb

Eine Wisej.NET-Anwendung fühlt sich wie WinForms an, weil Formulare, Pages, Controls, `BindingSource` -Komponenten und Event-Handler alle auf dem Server leben – und genau diese Vertrautheit verleitet zu einer gefährlichen Abkürzung: einen `DbContext` als ein weiteres langlebiges Serverobjekt zu behandeln. EF Core läuft zwar ebenfalls serverseitig, ist aber kein Container für UI-Zustand, sondern eine kurze Unit of Work, deren Change Tracker weder threadsicher noch zum Teilen gedacht ist. Viele Browser-Sessions öffnen die Support Desk Data Console gleichzeitig; die erste Fähigkeit dieses Kurses besteht deshalb darin, die Lebensdauern der UI und die Lebensdauer der Datenbankzugriffe auseinanderzuhalten, bevor überhaupt ein Grid gebunden wird.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
Session-Zustand	Der serverseitige Zustand pro Browser-Session, den Wisej.NET am Leben hält: offene Pages, die ausgewählte Zeile, Filter, Validierungsmeldungen und ausstehende Änderungen, erreichbar über <code>Application.Session</code> und den aktiven Control-Baum.
Lebensdauer von UI-Objekten	Wie lange eine <code>Page</code> , eine <code>Form</code> oder eine <code>BindingSource</code> existiert – von der Konstruktion, bis der Benutzer sie schließt oder die Session endet; oft Stunden, weit länger als jeder Datenbankaufruf.
Request-/Thread-Lebensdauer	Ein Browser-Event oder eine <code>await</code> -Fortsetzung auf dem Server; der Thread, der einen Click-Handler begonnen hat, ist nicht unbedingt derjenige, der ihn beendet.
Unit of Work	Die Lebensdauer eines <code>DbContext</code> : erzeugen, Entitäten abfragen oder ändern, <code>SaveChangesAsync</code> aufrufen, freigeben; Sekunden, nicht Sessions.
DbContextFactory	Der von <code>AddDbContextFactory</code> registrierte Service <code>IDbContextFactory<T></code> , der einen frischen, korrekt konfigurierten Context liefert, sobald eine Operation einen braucht.
Anbindung an die Microsoft-DI	Die Registrierung des <code>IServiceProvider</code> von ASP.NET Core bei <code>Application.Services</code> , damit Wisej.NET-Pages dieselben Services auflösen können, die der Startup konfiguriert hat.
sessionübergreifendes Datenleck	Ein statisches oder gemeinsam genutztes veränderliches Objekt (ein Context, ein Change Tracker, eine gebundene Liste, eine ausgewählte Entität), das die Session eines Benutzers lesen kann, nachdem die Session eines anderen Benutzers hineingeschrieben hat.

3. Der rote Faden der Lektion

Vier Lebensdauern auf einem Server

Alles in einem Wisej.NET-Prozess gehört zu einer von vier Lebensdauern, und die meisten Data-Binding-Fehler entstehen, wenn ein Objekt in der falschen landet. Der Session-Zustand lebt so lange wie die Browser-Session: die Identität des Benutzers, der geöffnete Ticket-Browser, die aktuellen Suchfilter. Die Lebensdauer von UI-Objekten gehört zu einer bestimmten `Page` oder `Form` und den Komponenten, die sie besitzt, etwa der `BindingSource` hinter dem Ticket-Grid und der ausgewählten Zeile; ein modaler Editor existiert nur, solange der Dialog offen ist. Die Request-Lebensdauer ist ein Roundtrip: Der Browser meldet einen Klick, der Server führt den Handler aus, und jedes `await` darin kann auf einem anderen Thread-Pool-Thread fortgesetzt werden – ein Handler darf also keine Thread-Affinität voraussetzen. Die Unit-of-Work-Lebensdauer gehört EF Core: Ein `DbContext` wird erzeugt, fragt eine Handvoll Entitäten ab oder ändert sie, speichert und wird freigegeben.

Diese Lebensdauern passen schlecht ineinander. Eine Session überdauert Tausende Requests, ein Formular überdauert viele Datenbankaufrufe, und ein Context sollte keinen von ihnen überdauern. Ein Context, der in einer Session oder einem Formular gehalten wird, trackt jede Entität, die er je geladen hat, wächst, bis

die Session endet, und wird zu einer veralteten Zweitkopie der Datenbank, die der Benutzer für aktuell hält. Zwei asynchrone Operationen auf demselben Context führen zu einer Exception in EF Core, weil ein Context nicht threadsicher ist. Das sind die Fehlerbilder hinter der ersten Frage des Moduls: Warum ist ein statischer oder gemeinsam genutzter `DbContext` gefährlich?

EF Core registrieren und die beiden Container verbinden

Wisej.NET hat ein eigenes Service-System und kann zusätzlich mit dem `IServiceProvider` von Microsoft arbeiten. Für EF Core ist die Microsoft-DI der natürliche Ort: `AddDbContextFactory<SupportDeskContext>` (oder `AddPooledDbContextFactory`, wenn häufiges Auf- und Abbauen von Verbindungen ins Gewicht fällt) registriert eine Factory, die den Provider, den Connection-String aus der Konfiguration und die nur in der Entwicklung aktiven Diagnosefunktionen kennt. Services, die die Factory verwenden, etwa der `TicketQueryService`, den das Lab aus einer Page auflöst, kommen in denselben Container. Nach `builder.Build()` übergibt eine einzige Zeile den Provider von Microsoft an Wisej.NET, sodass UI-Container der obersten Ebene diese Services auflösen können – mit `Application.Services.GetService<T>()` oder per Property-Injection mit dem Attribut `[Inject]` auf einer `Page` oder `Form`.

```
var builder = WebApplication.CreateBuilder(args);
var cs = builder.Configuration.GetConnectionString("SupportDesk")
    ?? throw new InvalidOperationException("Missing connection string 'SupportDesk'.");

builder.Services.AddDbContextFactory<SupportDeskContext>(options =>
{
    options.UseSqlServer(cs, sql => sql.EnableRetryOnFailure());
    if (builder.Environment.IsDevelopment())
        options.EnableDetailedErrors(); // Logging sensibler Daten: nur lokale Entwicklung
});
builder.Services.AddTransient<TicketQueryService>();

var app = builder.Build();
Application.Services.AddService<IServiceProvider>(app.Services); // die Anbindung an
Wisej.NET
```

Sobald die Verbindung steht, gilt eine Regel: Registrieren Sie jeden Anwendungs-Service über die Microsoft-DI und vermischen Sie keine Container, sonst erhalten Sie zwei Objekte, wo Sie eines erwartet haben. Der Connection-String kommt aus der Konfiguration, nie aus einer Konstante auf einer Oberfläche, und `EnableSensitiveDataLogging` ist ein Schalter für die lokale Entwicklung, der nie in Produktiv-Logs landen darf. Die Design-Time-Factory im Datenprojekt existiert nur, damit die `dotnet ef`-Tools einen Context erzeugen können, ohne die Webanwendung zu starten.

Ein Context pro Operation, pro Editor-Formular oder hinter einem Service

Der sicherste Standard für Wisej.NET-Datenoberflächen ist die Factory plus ein Context pro Operation. Eine Grid-Suche, das Laden eines Lookups, ein Speicherbefehl und ein Löschbefehl erzeugen und verwerfen jeweils ihren eigenen Context. Das entspricht der Empfehlung von EF Core selbst für zustandsbehaftete serverseitige UI-Frameworks, in denen Komponenten langlebig sind und ein Context mit Request-Scope nicht zur Lebensdauer der UI passt. In der Support Desk Data Console hat das die Form einer Service-

Methode: `await using var db = await _factory.CreateDbContextAsync();` gefolgt von `await db.Tickets.CountAsync()` – genau die erste asynchrone Abfrage, die das Lab hinter einem Lade-Flag und einer Fehlermeldung ausführt, noch ohne Binding.

Ein länger lebender Context ist an genau einer Stelle akzeptabel: in einem modalen Editor, wenn der Benutzer ein Ticket öffnet, getrackte Entitäten bearbeitet, durch einige kleine verknüpfte Collections wie die Kommentare navigiert und als eine Einheit speichert oder abbricht. Selbst dann ist er privat für diese Editor-Instanz, so geschützt, dass nie zwei Operationen gleichzeitig darauf laufen, und wird beim Freigeben des Formulars freigegeben – niemals in einem gemeinsam genutzten Service, einem statischen Feld oder einem anwendungsweiten Cache. Die dritte Option ist die Grenze über ein Repository oder einen Service: Pages sprechen mit `TicketQueryService` und `TicketCommandService`, die die Factory besitzen, sodass die UI nie einen Context zu sehen bekommt. Diese Grenze wächst im Verlauf der weiteren Labs, und deshalb löst das Lab aus der Page einen Abfrage-Service auf und keinen Context.

Session-Sicherheit: was geteilt werden darf und was nie

Jede Browser-Session hat ihre eigenen UI-Objekte, ausgewählte Zeile, Validierungsmeldungen, Filter und ausstehende Änderungen; das sind Belange der Session oder des Formulars. EF-Core-Entitäten, `DbContext`-Instanzen und Change Tracker dürfen Session-Grenzen nicht überschreiten. Ein statischer Cache darf unveränderliche Referenzdaten enthalten, etwa die einmal geladenen Ticket-Kategorien, oder kompilierte Abfrage-Delegates, weil niemand sie verändert. Er darf niemals einen veränderlichen `DbContext`, eine `BindingSource` oder eine ausgewählte Entität enthalten, denn ein statisches Member wird von jeder Session auf dem Server geteilt, und wer zuletzt schreibt, gewinnt – stillschweigend.

Dieselbe Regel gilt für Hintergrundverarbeitung: Ein mit `Application.StartTask` gestarteter Task erzeugt innerhalb der Operation seinen eigenen Context und aktualisiert die UI über die sessionbewussten Mechanismen von Wisej.NET, statt sich einen Context zu leihen, den eine Page erzeugt hat. Innerhalb eines Handlers wird jeder EF-Core-Aufruf abgewartet, bevor derselbe Context erneut verwendet wird, Buttons sind deaktiviert, solange ein Lade- oder Speichervorgang läuft, und eine verständliche Statusmeldung ersetzt die rohe Exception; das Lade-Flag im Lab ist die kleinste Form dieser Disziplin.



Typische Fehler

Den `DbContext` als global geteilten Wisej.NET-Service registrieren, sodass jede Session Entitäten im selben Objekt trackt. Entitäten oder eine `BindingSource` über ein statisches Feld oder einen Cache zwischen Sessions teilen. Parallele EF-Core-Operationen auf demselben Context starten, etwa zwei Abfragen, ohne die erste abzuwarten. Einen Context in einer Page am Leben halten, weil das schneller wirkt – bis sein Change Tracker jedes jemals geöffnete Ticket enthält. Nach dem Einrichten der Anbindung der Services zwischen der Registry von Wisej.NET und der Microsoft-DI vermischen.



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:

- ✓ den Unterschied zwischen dem Session-Zustand von Wisej.NET, der Lebensdauer von UI-Objekten, der Request-/Thread-Lebensdauer und der Unit-of-Work-Lebensdauer von EF Core erklären.
- ✓ EF Core bei der Dependency Injection von Microsoft registrieren und den Service-Provider für Wisej.NET bereitstellen.
- ✓ zwischen einem DbContext pro Operation, einem DbContext pro Editor-Formular und einer Repository-/Service-Kapselung wählen.
- ✓ gemeinsam genutzte statische DbContext-Instanzen und versehentliche sessionübergreifende Datenlecks vermeiden.

Lab- und Prüfungsleitfaden

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



Praxis-Lab

Legen Sie die Projektmappe Support Desk Data Console mit den Projekten Web, Data, Services und Tests an, fügen Sie dem Datenprojekt die EF-Core-Provider-Pakete hinzu, erstellen Sie SupportDeskContext mit einer Design-Time-Factory, registrieren Sie AddDbContextFactory im ASP.NET-Core-Startup und stellen Sie Wisej.NET den IServiceProvider von Microsoft bereit. Lösen Sie dann aus einer Wisej.NET-Page einen Abfrage-Service auf und führen Sie eine erste asynchrone Zählabfrage hinter einem Lade-Flag und einer Fehlermeldung aus – noch ohne Binding.

Ausgangspunkt

Öffnen Sie Visual Studio und legen Sie eine neue Projektmappe namens `SupportDesk` mit einer Wisej.NET-Webanwendung (`SupportDesk.Web`) sowie den leeren Klassenbibliotheken `SupportDesk.Data` und `SupportDesk.Services` und einem Testprojekt `SupportDesk.Tests` an; eine lokale SQL-Server-LocalDB- oder SQLite-Datenbank ist die einzige weitere Voraussetzung. Das Web-Template liefert Ihnen eine `Program.cs` und eine leere Haupt- `Page` ; es gibt noch keine Entitäten, keinen Context und keine Services, und das vollständige Modell folgt in Modul 2.

Vorgeschlagenes Vorgehen

- ✓ Fügen Sie `SupportDesk.Data` das EF-Core-Provider-Paket (SQL Server oder SQLite) und das Design-Paket hinzu und erstellen Sie dann `SupportDeskContext` mit einer minimalen `Ticket` -Entität und `DbSet<Ticket> Tickets` , damit die Zählabfrage etwas zu zählen hat.
- ✓ Fügen Sie `SupportDeskDesignTimeFactory` hinzu, die `IDesignTimeDbContextFactory<SupportDeskContext>` implementiert, damit die `dotnet ef` -Tools den Context erzeugen können, ohne die Webanwendung zu starten.
- ✓ Lesen Sie in `Program.cs` den Connection-String aus der Konfiguration, rufen Sie `AddDbContextFactory<SupportDeskContext>` mit nur in der Entwicklung aktiven Diagnosefunktionen auf, registrieren Sie `TicketQueryService` und rufen Sie nach `Build()` `Application.Services.AddService<IServiceProvider>(app.Services)` auf.
- ✓ Schreiben Sie `TicketQueryService` in `SupportDesk.Services` mit einer Konstruktor-Abhängigkeit `IDbContextFactory<SupportDeskContext>` und einer Methode `CountTicketsAsync` , die genau einen Context erzeugt, verwendet und freigibt.
- ✓ Lösen Sie in `TicketBrowserPage` den Service über eine `[Inject]` -Eigenschaft auf und verdrahten Sie dann `countButton_Click` als asynchronen Handler mit einem `_loading` -Flag, einem deaktivierten Button, einem Status-Label, einem try / catch, das eine verständliche Meldung anzeigt, und einem finally-Block, der den Button wiederherstellt.

Geforderte Ergebnisse

- ✓ Projektmappe mit den Projekten Web, Data, Services und Tests und den EF-Core-Provider-Paketen

- ✓ SupportDeskContext über AddDbContextFactory registriert, mit einer Design-Time-Factory
- ✓ IServiceProvider von Microsoft für Wisej.NET bereitgestellt und ein Abfrage-Service aus einer Page aufgelöst
- ✓ Erste asynchrone Zählabfrage mit Lade-Schutz und Fehlermeldung
- ✓ Nirgends in der Projektmappe ein statischer DbContext

Wiederholungsfragen

- ✓ Welche Lebensdauer haben Sie für den DbContext hinter `CountTicketsAsync` gewählt, und was würde schiefgehen, wenn die Page diesen Context zwischen den Klicks am Leben hielte?
- ✓ Zwei Benutzer öffnen gleichzeitig die Support-Desk-Konsole und klicken beide auf Count. Welche Objekte sind pro Session getrennt, welche werden geteilt, und warum ist diese Aufteilung sicher?
- ✓ Was sieht der Benutzer, wenn die Datenbank nicht erreichbar ist, was macht der Button, und wie wird das Lade-Flag wieder false?



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.