# Was ist EF-Core und welche Features hat es?

* Object-relational mapper (O/RM)
* Verbindet Relationale-Datenbanken (SQL, …) mit .NET-Objekten
* Konfiguration erfolgt über **class Context : DbContext**
* Eliminiert inline Queries durch Methodenaufrufe
  + Übersichtlicher & sicherer
* Entkoppelt Backend von der verwendeten Datenbank
* LinQ (Funktional) like Queries mit Hilfe von Expression-Trees
  + Compiler generiert den Code zur Konstruktion beim Kompilieren
  + Extension-Methoden für **IQueryable<T>** wie für **IEnumerable<T>**
  + Verwenden **Expression<Func<T, R>>** anstatt **Func<T, R>** als Parameter
* Durch Migrationen sind Änderungen an der Datenbank leicht anwendbar

Relational

Abstraktion für Relationale Datenbanken

NuGet: *Microsoft.EntityFrameWorkCore.Relational*

Provider

Konkrete Implementation von Relational für bestimmte Datenbank

NuGet für SQL Server/Azure: *Microsoft.EntityFrameworkCore.SqlServer*

NuGet für PostgreSQL: *Npgsql.EntityFrameworkCore.PostgreSQL*

Design

Erstellen von Migrationen, benötigt einen **IHost** als entry-point für das Migration-CLI

NuGet: *Microsoft.EntityFrameworkCore.Design*

CLI-Tool: *dotnet tool install --global dotnet-ef*

# EF-Core Setup

1. Füge NuGet Pakete von oben hinzu
2. Erstelle Context-Klasse die von **DbContext** erbt
3. Hinzufügen von DB-Context zu den Services
4. Verwende services.**AddDbContext<TContext>** bzw. **AddDbContextPool<TContext>**
   * ContextPool wiederverwendet Instanzen des DbContext
5. Konfiguration der **DbContextOptions**
   * Lambda beim Konfigurieren
   * OderÜberschreibe **OnConfiguring** in der Context-Klasse
6. Konfiguration des ContextModels
   * **DbContext** ist die Basisklasse
   * Definiere die EntityTabellen über public Properties DbSet<T>
   * Überschreibe **OnModelCreating** um das ContextModel zu Konfigurieren

# Entity-Konfiguration

* Lege Klasse für Entity an
* Lege die Verschiedenen Properties an
* Lege die Properties für Abhängigkeiten an
  + List<B> wenn B von A abhängig ist
  + B wenn A von B abhängig ist
    - Ebenfalls Property anlegen für den Foreign-Key z.B. BId
* Gehe in Context-Klasse **OnModelCreating**
* **builder.Entity<TEntity>(entity => {…})**
* Verwende das Lambda, um die Entity des Typs TEntity zu konfigurieren.

### Primary-Key

* Über Attribut **[Key]**
* Über Aufruf **entity.HasKey(x => x.Key)**

### Foreign-Key

* 1 – n – **entity.HasMany(…).WithOne(…).HasForeignKey(…)**
* n – 1 – **entity.HasOne(…).WithMany(…).HasForeignKey(…)**
* 1 – 1 – **entity.HasOne(…).WithOne(…).HasForeignKey<T>(…)**
  + T Definiert hier die Entity welche den Foreign Key enthält
* n – n – **entity.HasMany(…).WithMany(…).UsingEntity<T>()**
  + Verwendet einen JOIN table mit der Entity T und macht somit aus n – n zwei 1 – n Relationen

### Property

* Konfigurieren einer Property der Entity
  + **entity.Property(x => x. …)**
* Maximale Länge – für Strings, Arrays
  + **[MaxLength(…)]** Attribut auf der Property
  + **HasMaxLength(…)** als Folgeaufruf auf **entity.Property**
* Nullability – darf der Wert in der Spalte **null** sein
  + **public** **required …**
  + **public string?** … wenn in Nullable Context oder ein Value-Type
  + **[Required]** Attribut
  + **IsRequired()** als Folgeaufruf auf **entity.Property**
* ValueGenerated – Folgeaufruf – Wird ein Wert generiert bei Add/Update
* HasValueConversion – Legt einen weg fest, wie der Wert in der Datenbank repräsentiert wird. Nützlich für z.B. JSON, Enums. Benötigt ValueConverter und ValueComparer. Value Comparer wird benötigt für Change-Detection
  + ValueConverter hat 2 Paramter
    - FromProviderExpr: Konstruktion des .Net-Typ aus dem Typ der DB
    - ToProvioderExpr: Konstruktion des Db-Typs aus dem .Net-Typ
  + ValueComparer hat 3 Parameter
    - EqualsExpr: Gleichheitsausdruck
    - HashCodeExpr: Ausdruck zum HashCode für schnellere Vergleiche
    - SnapshotExpr: DeepCopy für den Change-Tracker
  + JSON-Blobs
    - In SQL ist das nvarchar(max), in PostgreSQL jsonb
    - Seit .NET7 Unterstützt EF-Core Provider-Agnostik JSON-Columns
    - Konfigurieren mit entity.OwnsOne/Many und .ToJson
    - SQL-Server und PostgreSQL unterstützen JSON-Queries
    - JSON ist oft aber langsamer und schwieriger zu patchen bei Updates
    - Wir verwenden oft einen ValueConverter & ValueComparer für JSON Columns
  + Enums
    - Ebenfalls nur konkreter anwendungsfall für ValueConverter & Comparer

### Indices

* Die zentrale Beschleunigungsstruktur für Queries
* Erstellen mit **entity.HasIndex(x => new {…})**
* Kann mehrere Spalten enthalten, diese sind anzugeben in der new {}
* Reihenfolge der Spalten ist wichtig! Von Most-Selective nach Least-Selective
* Provider kann nur bedingt Indices verwenden, aufpassen beim Bauen der Queries
* Index kann eindeutig sein mit .**IsUnique()**
  + Ist keine der spalten nullable ergibt sich ein Alternate-Key der eine Zeile eindeutig identifiziert.
  + Unique Indices filtern **null** raus, wichtig für Queries. Eventuell 2 Indices anlegen, einen für Unqiue Constraint & einen für Performance
* Indices können Spalten einbeziehen
  + Ist z.B. nützlich, wenn der Index ein Key Lookup für z.B. JOINs ist
  + Besonders gut wenn Zeilen besonders breit mit vielen Daten, z.B. großen JsonBlobs
  + Nicht zu viele Spalten auswählen
  + Reihenfolge ist hier egal
  + Extension-Methode ist nur verfügbar, wenn der Kontext seinen Provider weiß
  + Funktioniert in SQL-Server und PostgreSQL

# Migrationen

Änderungen am Model repräsentieren echte Änderungen an der Datenbank. Da es aber für Produktivsysteme nicht möglich ist die Datenbank einfach neu aufzusetzen ist es notwendig den letzten Stand der Datenbank zu behalten, um die Änderungen in der Reihenfolge auszuführen, in der sie durchgeführt wurden.

Beim Konfigurieren des Datenbank Providers werden folgende Dinge angegeben:

* **MigrationsHistoryTable(…)** – In dieser Tabelle Merkt sich die Datenbank welche Migrationen bisher ausgeführt wurden.
* **MigrationsAssembly(…) –** Hier werden im Ordner **Migrations** die generierten Migrations-Dateien gespeichert

Der Ordner Migrations enthält 3 Typen von Dateien:

1. Migration Dateien: Diese beginnen mit einem Timestamp im Namen um die Reihenfolge festzulegen. Diese enthalten die Operationen die nötig sind, um die Migration auszuführen.

Diese sollten stets reviewt werden und können ggf. angepasst werden.

1. .Designer Dateien: Diese beinhalten den zustand den die Datenbank haben sollte direkt nachdem die Migration ausgeführt wurde.
2. ContextModelSnapshot: Ist die wichtigste von allen, diese beinhaltet den aktuellen Stand der Datenbank, nachdem alle Migrationen ausgeführt wurden. Diese sollte **niemals** manuell bearbeitet oder gar gelöscht werden.

Anlegen von Migrationen:

1. Ändern der Entities bzw einzelner Konfiguration in ContextModel.
2. Commit aller Änderungen vor dem erstellen der Migration
3. Benutzung der CLI von EF-Core zum hinzufügen oder ein Skript das bereit gestellt wurde.
   * Terminal: cd <Pfad zum Projekt das EF-Core.Design hat>
   * Terminal: *dotnet ef migrations add <MigrationName> -c <ContextName>*
4. **IMMER** Review der Migration Datei **sorgfältig** durchlesen. Bei fragen auch andere Leute zu Hilfe ziehen.
5. Am besten dann auch noch die GIT-DIFF auf den ContextModelSnapshot durchschauen
6. Commit der Migration, dieser sollte nur 3 Dateien enthalten.

Bei folgenden Dingen aufpassen:

* Umbenennungen insbesondere, wenn sich der Typ auch geändert hat
* Hinzufügen von nicht-nullable Columns
* Neue Navigation-Properties sollten **IMMER EXPLIZIT** im OnModelCreating konfiguiert werden, sonst erfindet EF-Core Shadow-Properties für ForeignKeys.
* Neue Spalten mit Komplexen Typen, also nicht string, int etc.
* Alternate-Keys

Es gibt fälle wo es notwendig ist SQL zu schreiben damit eine Migration richtig funktioniert. In diesen Fällen muss die Migration-Datei bearbeitet werden. Aber nicht der Designer oder der ContextModelSnapshot.

Wenn ihr beim **Mergekonflikte** mit Migrationen habt, geht sorgfältig die Änderungen durch und wenn ihr nicht wisst, welche ihr übernehmen, solltet frag einen Senior. Ansonsten, falls die Migration noch nicht ausgeführt wurde: Löschen und neu anlegen (GIT-Revert auf den Commit mit der Migration).

# Wie Funktionieren Queries?

1. Auswahl der Query Root über context.Set<T> (quasi das FROM)
2. Verkettung von Extension Methoden für IQueryable<T>
   1. Mit jedem weiteren Methodenaufruf wird die Query-Expression erweitert
3. Ausführung der Query über den QueryProvider (das wäre z.B. SQL, Postgres)
   1. Optimierung des Expression Trees (z.B. streichen von Auswertungen)
   2. Query-Cache Lookup, Query erstellen ist teuer -> Cache
   3. Kompilierung der Query in Provider-Command (z.B. SQL)
      1. Hier passieren die meisten Übersetzungs-Fehler, meist sind das nicht unterstütze Verkettungen von Aufrufen.
   4. Senden an die DB und initialisieren des Result-Readers
   5. Lesen des Results und eventuelle Nachbearbeitung, insbesondere Includes
   6. Aufbau der Runtime-Objekte

# Wie werden manche manche Methoden übersetzt?

Die meisten sind klar, die interessanten sind hier:

* First = Take 1
* Single = Take 2, Throw wenn 2 gefunden
* Any,All = (NOT) Exists
* (Then) Include = LEFT JOIN
* Join = INNER-JOIN
* GroupBy = GROUP BY
* GroupJoin = OUTER APPLY
* AsQuerable = Kein Effekt, man kann hiermit Expression<Func<…>> als paramter für z.B. Where übergeben
* AsEnumerable = Ab hier wird die Query serverseitig ausgewertet.
  + ACHTUNG dadurch ist Query nicht länger ASYNC und somit Blocking!
* ToList,ToArray = Hints für Typinferenz in Projektionen für AnonymeObjekte

# Welche Queries sind nicht übersetzbar?

Allgemein sind die äußeren Aufruf-Ketten auf das IQueryable immer übersetzbar, es sind meist die Inner-Queries die die Probleme verursachen, insbesondere jene die man bei einer Map für Automapper generiert, da es sich bei jenen praktisch immer um Inner-Queries handelt.

Zum Debugging gibt es ein paar Tricks:

* Breakpoints vor dem Ausführen
* Breakpoint im **NavigationExpandingExpressionVisitor**
  + Methode: **VisitMethodCall**
  + Der Switch über **Method.Name** kann Aufschluss geben
  + Schrittweise durchsteppen
* Manchmal sind diese Breakpoints nicht verfügbar
  + Dann Breakpoint in Konstruktor der **InvalidOperationException(string?)**
  + Von hieraus kann man sehen welcher der Aufruf Pfad ist
  + Auswahl der obersten **VisitMethodCall** im Call-Stack, dann suchen nach Method.Name
* Der Debug String von EF-Core sollte teils auch ausreichen, dieser ist **NUR** die nicht übersetzbare INNER-Expression. Im Fall von Automapper also Map.ForMember
* Stückweise nur einzelnen Projektionen durchgehen bzw nur teile der Query auswerten
* Hilfe holen
* Beten

Es gibt so einige Kandidaten, die gerne nicht übersetzbar sind. Nachfolgend eine Liste.

Innere Queries:

* Aggregate overloads
* GroupJoin overloads
* Zip
* SequenceEqual overloads
* ElementAt
* ElementAtOrDefault
* SkipWhile
* TakeWhile
* DefaultIfEmpty with argument
* Index based lambda overloads of Where, SkipWhile, TakeWhile, Select, SelectMany
* IEqualityComparer overloads of Distinct, Contains, Join, Except, Intersect, Union, OrderBy, ThenBy, OrderByDescending, ThenByDescending, GroupBy
* Projektion von Properties oder Objekten, die nicht dem Typ der Spalte entsprechen (z.B. **JSON-Object** => **string**), oder Logik, die auf ihnen basiert (z.B. JOINs über liste von Werten in JSON-Array)

Und als Innere Query in Kombination mit einem GroupBy in der Äußern Query:

* Average/Max/Min/Sum
* Distinct
* Contains
* First/Single/Last(OrDefault)
* Join, LeftJoin, GroupJoin
* SelectMany
* Concat/Except/Intersect/Union
* Cast/OfType
* Include/ThenInclude/NotQuiteInclude
* GroupBy
* Reverse
* DefaultIfEmpty

# Change Tracking

EF-Cores Persistenz Konzept basiert auf Change Tracking, heißt der DbContext speichert sich eine Kopie des Orginalzustands einer Entity beim holen von der Datenbank (Snapshot) und kann so beim **SaveChanges** rausfinden welche Entitys sich echt geändert haben und aktualisiert dann die entsprechenden Zeilen in Datenbank.

Beim Arbeiten mit EF-Core interagieren wir auf folgende Weise mit dem ChangeTracking:

* query.ToList () (o.ä.), Funktionen die Queries auf der Datenbank ausführen und **nicht** mit **AsNoTracking** konfiguriert wurden fügen diese zur Change-Detection hinzu.
* **Add, Update, Remove** auf Context oder **DbSet** fügt die Entities in dem Change-Tracker in dem entsprechenden zustand hinzu.
  + Gilt auch für **Async** / bzw **Range** varianten
* **SaveChanges** führt DetectChanges aus und detektiert Änderungen an Entities und speichert diese dann in der Datenbank
  + Die automatische Change-Detection bei SaveChanges lässt sich auch deaktivieren
* **ChangeTracker.Clear()** – Hört auf alle Entities zu Tracken.

# Row-Versions & Concurrency-Tokens

Bei großen Systemen kann es vorkommen das es Race-Conditions zu Updates auf Entities in der Datenbank gibt. Hierfür gibt es Row-Versionen und Concurrency-Token. Diese werden von EF-Core genutzt, um diese zu detektieren und eine Exception zu schmeißen. Diese funktionieren wie folgt:

1. Beim erstellen des Models kann mit **entity.Property(x => x.Version).IsRowVersion()** bzw. **IsConcurencyToken()** Konfiguriert werden.
2. Wird nun eine Entity aktualisiert, überprüft EF-Core die Row-Version/das ConcurrencyToken auf Gleichheit mit dem neuen Zustand der Entity. Ist dieser nicht gleicht wird kein update ausgeführt und eine Exception wird generiert.
3. Dann muss die Entity erneuet angefordert werden und das update auf dem neuen zustand mit der richtigen Version ausgeführt werden.
4. Stichwort: **Optimistic-Concurrency,** man wiederholt das ein paar mal und hofft das sich der Konflikt von selbst löst.

# Transactions

Bei manchen Updates ist wichtig das diese von außen als eine Atomare Operation sichtbar ist, also entweder alle Änderungen auf einmal oder keine. SQL-Server und andere Provider unterstützen diese. EF-Core erlaubt ebenfalls Zugriff auf diese mit:

* **await using var transaction = await Context.BeginTransactionAsync()** 
  + Erstellt eine neue Datenbanktransaktion auf dem Context. Alle Änderungen, die nach diesem punkt passieren werden nur auf dieser Transaktion ausgeführt, bis diese Commited wird.
  + Es kann nur eine Transaktion pro Context Instanz gleichzeitig geben.
* **await transaction.CommitAsync()**
  + Persistiert die Transaktion auf der Datenbank und macht die Änderungen nach außen sichtbar.
  + Ab hier werden alle Änderungen wieder direkt ausgeführt.
  + Es kann nun eine neue Transaktion erstellt werden.
* **await transaction.RollbackAsync()**
  + Macht alle Änderungen rückgängig.
  + Geht die Transaction durch das using out-of-scope, wird automatisch das Rollback ausgeführt, wenn die Transaction Disposed wird. Ein try-catch block um den Inhalt der Transaction ist also nicht notwendig, solange diese am Ende out-of-scope geht.

# Bulk-Operationen

Manchmal ist es vorteilhaft bzw. notwendig Operationen ohne Change-Tracking auszuführen. Für diesen Zweck gibt es Bulk-Operationen. Diese erlauben es viele Änderungen an der Datenbank durchzuführen mit wenig/ohne einen Lesevorgang. Folgende Operationen erweisen sich oft als nützlich.

**Achtung:** Bulk-Operation berücksichtigen keine Row-Versions und Concurrency-Tokens beim Ausführen, schreiben diese aber teils. Diese können also in Kombination mit Change-Tracking Szenarien produzieren in denen ein **SaveChanges** **IMMER** fehlschlägt.

## **BulkInsert, BulkUpdate, BulkMerge**

Diese sind Extension-Methoden auf DbSet<T> des NuGet Pakets: **Z.EntityFramework.Extensions.EFCore.**

Diese erlauben es große Updates mit möglichst wenig Queries durchzuführen und basieren auf einem Table-Merge.

Diese funktionieren wie folgt:

1. Eine Liste von Entities wird übergeben.
2. Operation mit einem Lambda als 2. Parameter konfigurieren. Es gibt u.a. folgende Optionen
   1. **ColumnPrimaryKeyExpression** – gibt an mit welchem Key das Matching in Schritt 4 ausgeführt wird. Besonders nützlich wenn man einen Unique Constraint erfülllen will.

Syntax: x => new { x.PropertyA, x.PropertyB}

* 1. **MergeKeepIdentity** – Definiert ob der Primary mit dem der hochgeladen Entity überschrieben wird, am besten auf true-setzten, wenn nicht mit dem Primary gemachted wird.
  2. **ColumnInputExpression** – Definiert die Spalten die gesetzt werden. Diese **MÜSSEN** die ColumnPrimaryKeyExpression enthalten

Syntax => new { x.Key, x.PropertyA, x.PropertyB }

* 1. **IgnoreOnMergeUpdateExpression** – Definiert spalten die beim Update ignoriert werden sollen aber beim Insert eingefügt werden sollen, z.B. nützlich für ein Created Date.
  2. **IgnoreOnMergeInsertExpression**- So wie d. nur andersherum.

1. Die Entities werden mit einem INSERT in einen temporären Staging-Table übertragen
2. Ein Table-Merge wir ausgeführt mit den 2 Branches:
   1. Matched: Der Konfigurierte Key existiert bereits, dann UPDATE
      1. Falls BulkInsert => Exception
   2. Not-Matched: Der Konfigurierte Key existiert noch nicht, dann INSERT
      1. Falls BulkUpdate => Nichts passiert, **ACHTUNG** Silent Failure

## **ExecuteUpdate/UpdateFromQuery**

Diese beiden sind Extensions auf IQuerable und werten das Query datenbankseitig aus und Produzieren ein UPDATE mit den angegebenen SET werten.

* Execute Update erwartet eine Expression von s => s.SetProperty(…).SetProperty(…) Aufrufen.
* UpdateFromQuery erwartet eine NewExpression als entity => new TEntity{ … = … }

Seit EF.Core7 ist **UpdateFromQuery** **überflüssig**, die hausinterne variante von EF-Core sollte vorgezogen werden, da diese direkt in das Framework integriert ist.

## **ExecuteDelete/DeleteFromQuery**

Diese beiden sind Extensions auf IQuerable und werten das Query datenbankseitig aus und Produzieren ein DELETE. Die Verwendung ist lediglich ein Aufruf am ende der Query, wichtig ist hierbei nur das es sich bei dem Projizierten Typ handelt der ein DbSet<TEntity> hat. Hier sollte man ebenfalls ExecuteDelete vorziehen da es direkt von EF-Core ist.

## **InsertFromQuery**

Die Most-Underrated BulkExtension, diese erlaubt es neue Zeilen zu produzieren nur durch einen Aufruf von der Serverseite, ohne vorheriges lesen aus der Datenbank.

Zunächst baut man sich eine Query zusammen die einzufügenden Zeilen aus den vorhandenen Daten produziert, z.B. füge für jede Zeile in Tabelle A eine Zeile in Tabelle B ein.

## **WhereBulkContains**

Ist quasi wie ein Where(x => list.Contains(x.Property)) auf dem IQuerable, nur produziert dieses anstatt eines **IN (…)** Statements einen **INNER JOIN** mit Tmp-Table mit einem Index auf dem KEY. Also **O (log n)** anstatt **O(n)**. Nützlich wenn in der **IN** Liste Tausende von Werten drinstehen. **Achtung**: Es ist nur ein WhereBulkContains pro Query möglich, warum? Keine Ahnung.

# Zum Abschluss:

Nur zu wissen, wie die Entities aussehen reicht nicht um gut mit EF-Core zu entwickeln, macht euch mit dem Datenbank-schema vertraut, welche Indizes gibt es etc. Wo lohnt es sich Bulk zu verwenden, wo hat man Queries die verallgemeinert werden können. Wo habe ich einen Loop, der eine Query immer wieder ausführt, nur mit anderem Parameter? Viele Queries zeigen ihre schlechte Optimierung erst wenn die Datenbank ausreichend befüllt ist, deswegen sind für EF-Core Stresstests besonders wichtig.