Abfragen mit LINQ to Entities

- LINQ-Abfragen werden dazu verwendet, Informationen aus SQL-Datenbanken zu beziehen. Bei diesen sogenannten LINQ-to-Entities - Abfragen spielen Erweiterungsmethoden, welche die statische Klasse Queryable im Namensraum System.Linq für alle Typen realisiert, welche die von IEnumerable abgeleitete Schnittstelle IQueryable implementieren, eine wesentliche Rolle.
- Nach erfolgreicher Migration kann man die DBContext-Klasse nutzen, um Daten per LINQ abzufragen und um Daten zu manipulieren (d.h. INSERT, UPDATE und DELETE-Befehle zu generieren)
- DbSet implementiert die Schnittstelle IQueryable, so dass die in der statischen Klasse Queryable definierten Erweiterungsmethoden genutzt werden können, um Entitäten aus den zum Kontext gehörenden Datenbanktabellen abzufragen. Die Quelle der Abfrage wird also über eine Eigenschaft der DbContext - Ableitung im EF Core - Modell angesprochen.
- Wir diskutieren jetzt Varianten der Formulierung von Abfragen:
  - Klassische Formulierung in LINQ
  - LINQ mit anonymen Funktionen (Lambda-Expression)

Die Schnittstellen für Collections

IEnumerable<T>, ICollection<T>, IList<T>, IQuerable <T>

FEATURES	IENUMERABLE	ICOLLECTION	ILIST
Loop Over	●	●	●
Count & Contains		●	●
Add / Remove		●	●
Use Indexer			●

Abfragen mit LINQ to Entities
mit einem Kriterium und einem Order By

- Variante „Abfrage-Kalkül“

```
var mitarbeiterProduktion_1= from m in
        dbctx.MitarbeiterListe
        where m.Abteilung.Bezeichnung=="Produktion")
        orderby m.Nachname
        select m;
```

```
Display(mitarbeiterProduktion_1.ToQueryString());
```

```
SELECT [m].[MitarbeiterId], [m].[AbteilungId],
[m].[Datum_Austritt], [m].[Datum_Eintritt], [m].[Durchwahl],
[m].[Email], [m].[Extern], [m].[Geburtsdatum],
[m].[Geschlecht],
[m].[Nachname], [m].[Vorname]
FROM [MitarbeiterListe] AS [m]
INNER JOIN [AbteilungsListe] AS [a] ON [m].[AbteilungId] =
[a].[AbteilungId]
WHERE [a].[Bezeichnung] = N'Produktion'
ORDER BY [m].[Nachname]
```

- Variante „LINQ-Abfrage mit Lambda-Ausdruck“

```
var mitarbeiterProduktion_2 = dbctx.MitarbeiterListe
    .Where(m => m.Abteilung.Bezeichnung == "Produktion")
    .OrderBy(m => m.Nachname);
```

```
Display(mitarbeiterProduktion_2.ToQueryString());
```

```
SELECT [m].[MitarbeiterId], [m].[AbteilungId],
[m].[Datum_Austritt], [m].[Datum_Eintritt], [m].[Durchwahl],
[m].[Email], [m].[Extern], [m].[Geburtsdatum],
[m].[Geschlecht],
[m].[Nachname], [m].[Vorname]
FROM [MitarbeiterListe] AS [m]
INNER JOIN [AbteilungsListe] AS [a] ON [m].[AbteilungId] =
[a].[AbteilungId]
WHERE [a].[Bezeichnung] = N'Produktion'
ORDER BY [m].[Nachname]
```

ToQueryString()

Abfragen mit LINQ to Entities mit einem Kriterium und einem Order By sowie einer speziellen Feldauswahl

- Variante „ Abfrage-Kalkül“

```
var mitarbeiterProduktion_1 = from m in dbctx.MitarbeiterListe
  where (m.Abteilung.Bezeichnung == "Produktion")
  orderby m.Nachname
  select new {
    Mitarbeitername = m.Nachname + ", " + m.Vorname,
    Abteilung = m.Abteilung.Bezeichnung };
```

```
Display(mitarbeiterProduktion_1.ToQueryString());
```

```
SELECT (COALESCE([m].[Nachname], N'') + N', ') +
COALESCE([m].[Vorname], N'') AS [Mitarbeitername],
[a].[Bezeichnung] AS [Abteilung]
FROM [MitarbeiterListe] AS [m]
INNER JOIN [AbteilungsListe] AS [a] ON [m].[AbteilungId] =
[a].[AbteilungId]
WHERE [a].[Bezeichnung] = N'Produktion'
ORDER BY [m].[Nachname]
```

- Variante „LINQ-Abfrage mit Lambda-Ausdruck“

```
var mitarbeiterProduktion_2 = dbctx.MitarbeiterListe
  .Where(m => m.Abteilung.Bezeichnung == "Produktion")
  .OrderBy(m => m.Nachname)
  .Select (m => new {
       Mitarbeitername =m.Nachname + ", " + m.Vorname,
       Abteilung=m.Abteilung.Bezeichnung});
```

```
Display(mitarbeiterProduktion_2.ToQueryString());
```

```
SELECT (COALESCE([m].[Nachname], N'') + N', ') +
COALESCE([m].[Vorname], N'') AS [Mitarbeitername],
[a].[Bezeichnung] AS [Abteilung]
FROM [MitarbeiterListe] AS [m]
INNER JOIN [AbteilungsListe] AS [a] ON [m].[AbteilungId] =
[a].[AbteilungId]
WHERE [a].[Bezeichnung] = N'Produktion'
ORDER BY [m].[Nachname]
```

Wichtige Befehle

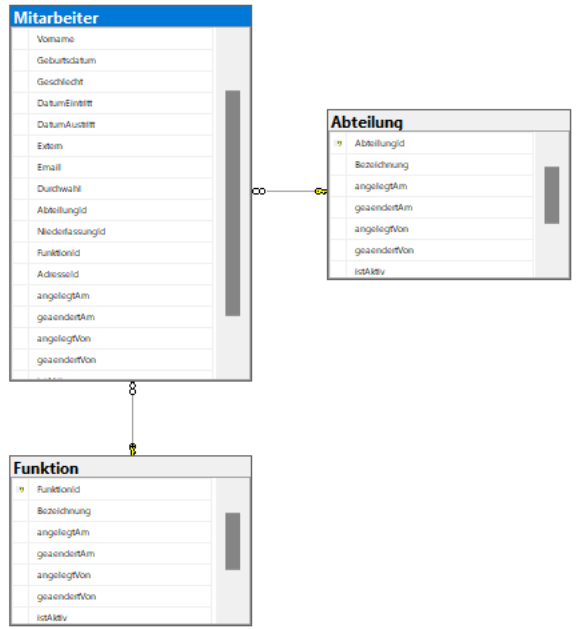
LINQ Methoden	
Select()	
Where()	
Find()	
GroupBy	
OrderBy	
ThenBy	
OrderByDescending	
Include	
ThenInclude	

LINQ Extension Methoden	
First()	
FirstOrDefault()	
Single()	
SingleOrDefault()	
ToList()	
Count()	
Min()	
Max()	
Last()	
LastOrDefault()	
Average()	

Abfragen zu den Mitarbeitern

```
//Einfache Abfrage:
Display("Mitarbeiter:");
var mitties = dbctx.MitarbeiterListe.ToList();
foreach (var m  in mitties)
{
   int? jahr = Convert.ToDateTime(m.Geburtsdatum).Year;
   Display($"{m.Nachname}, {m.Vorname}, {m.Geschlecht}," +
          $" Geburtsjahr: {jahr}");
}
```

```
//Abfrage mit JOIN:
Display("Mitarbeiter mit Funktion, Abteilung:");
var mittiesMitFktUndAbt= dbctx.MitarbeiterListe
   .Include(x => x.Funktion)
   .Include(x => x.Abteilung)
   .Where(x => x.Abteilung.Bezeichnung.Contains("Prod"))
   .OrderBy(x => x.Nachname)
   .ToList();
```



Komplexere Abfragen


```
Display("Mitarbeiter mit Funktion, Abteilung und Niederlassung und deren Adresse:");
var mittiesMitNiederlassung = dbctx.MitarbeiterListe
  .Include(x => x.Funktion)
  .Include(x => x.Abteilung)
  .Include(x => x.Niederlassung)
  .Include(x => x.Niederlassung.Adresse)
  .ToList();
foreach (var m in mittiesMitNiederlassung)
{
Display($"{m.Vorname + ' ' + m.Nachname} " +
    $"– {m.Funktion.Bezeichnung}, {m.Abteilung.Bezeichnung}" +
    $", Niederlassung: {m.Niederlassung.Bezeichnung} " +
    $" in {m.Niederlassung.Adresse.PLZ} {m.Niederlassung.Adresse.Ort} ");
}
```

Nutzung eines Recordtyps für eine Einschränkung der Felder (Projektion)

- Durch die Auswahl von Feldern in der Abfrage (Projektion) wird die Last bei der Übertragung verringert:

```
Display("Mitarbeiter mit Funktion, Abteilung und Niederlassung mit Projektion:");
Display("(Für die Projektion nutzen wir den Recordtyp)");
var mittiesMitNiederlassungAuswahl = dbctx.MitarbeiterListe
   .Include(x => x.Funktion)
   .Include(x => x.Abteilung)
   .Include(x => x.Niederlassung)
   .Include(x => x.Niederlassung.Adresse)
   .Select(x => new MitarbeiterAuswahl(
          x.MitarbeiterId, x.Nachname, x.Vorname,
          x.Funktion.Bezeichnung, x.Abteilung.Bezeichnung,
          x.Niederlassung.Bezeichnung,
          x.Niederlassung.Adresse.PLZ + " " + x.Niederlassung.Adresse.Ort,
          x.Niederlassung.Telefon + "-" + x.Durchwahl))
    .AsNoTracking()
    .ToList();
```

```
private record MitarbeiterAuswahl(int MitarbeiterId, string Nachname, string Vorname,
            string Funktion, string Abteilung, string Niederlassung, string PLZundOrt, string Telefon);
```

Mitarbeiter mit Funktion, Abteilung, Niederlassung und Projekt

- Auch komplexe Abfragen lassen sich so gestalten:


```
Display("Mitarbeiter mit Funktion, Abteilung, Niederlassung und Projekt:");
var mittiesMitFktUndAbtUndProjektMitNiederlassung = dbctx.ProjektMitarbeiterListe
  .Include(pm => pm.Projekt)
  .Include(pm => pm.Mitarbeiter)
  .Select(x => new MitarbeiterImProjektMitNiederlassung(x.MitarbeiterId, x.Mitarbeiter.Nachname,
          x.Mitarbeiter.Vorname,
          x.Funktion.Bezeichnung, x.Mitarbeiter.Abteilung.Bezeichnung,
          x.Mitarbeiter.Niederlassung.Bezeichnung, x.Mitarbeiter.Niederlassung.Adresse.PLZ + "-" +
          x.Mitarbeiter.Niederlassung.Adresse.Ort,
          x.Projekt.Bezeichnung, x.EinsatzBeginn))
   .ToList();
```

```
private record MitarbeiterImProjektMitNiederlassung(int MitarbeiterId, string Nachname, string Vorname,
          string Funktion, string Abteilung,string Niederlassung, string PLZundOrt,
          string? Projekt, DateTime? Einsatzbeginn);
```

Projekte, denen keine Mitarbeiter zugeordnet sind und Mitarbeiter, die keinem Projekt zugeordnet sind

- Eine komplexe SQL-Abfrage mit FULL OUTER JOIN zeigt bei einer n:m – Beziehung auch nicht zugeordnete Elemente auf beiden Seiten.
- Mit EFC können wir 2 Abfragen generieren, die uns die nicht zugeordneten Elemente ausgeben:
- Projekte, denen keine Mitarbeiter zugeordnet sind:

```
var projekteOhneMitarbeiter = from p in dbctx.ProjektListe
                  where (!p.ProjektmitarbeiterListe.Any(pm => pm.ProjektId == p.ProjektId))
                  orderby p.Projektbezeichnung
                  select new { Projekt=p.Projektbezeichnung };
```

- Mitarbeiter, die keinen Projekten zugeordnet sind:

```
var mitarbeiterOhneProjekt = dbctx.MitarbeiterListe
    .Where(m => !m.ProjektmitarbeiterListe.Any(pm => pm.MitarbeiterId == m.MitarbeiterId))
    .Select(m => new
    {
        Name = m.Nachname + ", " + m.Vorname
    })
    .OrderBy(m => m.Name);
```

Any und All

- Die Methoden Any() und All() überprüfen eine Reihe von Datensätzen, um festzustellen, ob irgendwelche Datensätze die Kriterien erfüllen (Any()) oder ob alle Datensätze die Kriterien erfüllen (All()). Genau wie die Aggregationsmethoden kann die Any()-Methode (aber nicht die All()-Methode) am Ende einer LINQ-Abfrage mit der Where()-Methode hinzugefügt werden, oder der Filterausdruck kann in der Methode selbst enthalten sein.Any()- und All()-Methoden werden serverseitig ausgeführt, und von der Abfrage wird ein Boolean zurückgegeben.

Gruppierungsabfragen

Gruppierung nach Projekt, Summe fakt.Stunden und Anzahl Mitarbeiter

```
Display("Gruppierung nach Projekt, Summe fakt.Stunden und Anzahl Mitarbeiter:");
var gruppNachProjekt = dbctx.ProjektMitarbeiterListe
    .GroupBy(pm => pm.Projekt.Bezeichnung)
    .Select(pm => new
    {
          Projektname = pm.Select(pm => pm.Projekt.Bezeichnung).First(),
          Summe = pm.Sum(x => x.FakturierbareStunden),
          Anzahl = pm.Count()
    })
    .ToList();

foreach (var p in gruppNachProjekt)
{
    Display($"Projekt: {p.Projektname}, Summe: {p.Summe}, Anzahl der MItarbeiter: {p.Anzahl}");
}
```

Nutzung von Funktionen in LINQ, hier: Mitarbeiter mit Altersberechnung per C#-Funktion und Berechnung der Dienstjahre mit EF.Functions.DateDiffYear(…)

```
Display(":");
int aktJahr = DateTime.Now.Year;
var mittiesMitEFFunktion = db.MitarbeiterListe
.Include(x => x.Funktion)
.Include(x => x.Abteilung)
.Select(x => new MitarbeiterAuswahlMitAlter(
          x.MitarbeiterId, x.Nachname, x.Vorname,x.Geschlecht,
          x.Funktion.Bezeichnung, x.Abteilung.Bezeichnung,
          BerechneAlter(x.Geburtsdatum), //Alter: Beispiel für die Nutzung einer C#-Funktion
          (int)EF.Functions.DateDiffYear(x.DatumEintritt, DateTime.Now))) //Dienstjahre: Nutzung einer EF-Funktion
.ToList();

foreach (var m in mittiesMitEFFunktion)
{
    Display($"{m.Nachname}, {m.Vorname}, {m.Geschlecht}, Alter: {m.Alter}, Dienstjahre: {m.Dienstjahre}");
}
```

```
private static int BerechneAlter(DateTime? geburtsdatum)
{
  int alter = 0;
  if (geburtsdatum == null)
     return alter;
   alter = DateTime.Now.Year - Convert.ToDateTime(geburtsdatum).Year;
   if (Convert.ToDateTime(geburtsdatum).AddYears(alter) > DateTime.Now)
     alter -= 1;
   return alter;
}
```

```
private record MitarbeiterAuswahlMitAlter(int MitarbeiterId, string Nachname, string Vorname,
            EGeschlecht Geschlecht, string Funktion, string Abteilung, int Alter, int Dienstjahre);
```

Nutzung von nutzerdefinierten Datenbank-Funktionen

- Beispiele: Die Datenbank enthält die beiden Funktionen ufnBerechneAlter und ufnBerechneDienstjahre. Diese sollen genutzt werden, um in LINQ-Abfragen die entsprechenden Mitarbeiter-Eigenschaften zu berechnen
- In die DBContext-Klasse werden die beiden statischen Funktionen eingeführt, die die Annotation DbFunction erhalten

```
[DbFunction("ufnBerechneAlter", "dbo")]
public static int Alter(DateTime? geburtsdatum)
                  => throw new InvalidOperationException($" {nameof(Alter)} hat zu einem Fehler geführt");

[DbFunction("ufnBerechneDienstjahre", "dbo")]
public static int Dienstjahre(DateTime eintrittsdatum, DateTime? austrittsdatum)
                  => throw new InvalidOperationException($"{nameof(Dienstjahre)} hat zu einem Fehler geführt");
```

- Im LINQ-Aufruf kann nun auf die statischen Methoden der Klasse zugegriffen werden:

```
var mitarbeiterMitDBFunktion =
  db.MitarbeiterListe
    .Include(x => x.Funktion)
    .Include(x => x.Abteilung)
    .Select(x => new MitarbeiterAuswahlMitAlter(
           x.MitarbeiterId, x.Nachname, x.Vorname, x.Geschlecht,
           x.Funktion.Bezeichnung,
           x.Abteilung.Bezeichnung,
           ProjektDBContext.Alter(x.Geburtsdatum),
           ProjektDBContext.Dienstjahre(x.DatumEintritt, (x.DatumAustritt==null) ? new DateTime(1900,1,1) : x.DatumAustritt )))
    .ToList();
```

Zugriff auf die statischen Methoden der DBContext-Klasse!

Nutzung von Computed Columns

- Viele Datenbanken enthalten die Möglichkeit berechnete Spalten in Tabellen einzurichten.
- Beispiele:
  - Die Tabelle Mitarbeiter soll eine solche Spalte erhalten, in der mittels einer DB-Funktion ufnAnredeImAdressfenster die Anrede im Adressfenster berechnet werden soll
  - Eine zweite Spalte soll den vollständigen Namen berechnen
- Hierzu werden die Entity-Klasse Mitarbeiter und die DBContext-Klasse erweitert:

```
[DatabaseGenerated(DatabaseGeneratedOption.Computed)]
[StringLength(100)]
public string VollstaendigerName { get; private set; }

[DatabaseGenerated(DatabaseGeneratedOption.Computed)]
[StringLength(100)]
public string AnredeImAdressFenster { get; private set; }
```

```
modelBuilder.Entity<Mitarbeiter>()
  .Property(u => u.VollstaendigerName)
  .HasMaxLength(100)
  .HasComputedColumnSql("CONCAT([VorName], ' ' ,[NachName])");

modelBuilder.Entity<Mitarbeiter>()
  .Property(u => u.AnredeImAdressFenster)
  .HasMaxLength(100)
  .HasComputedColumnSql("dbo.ufnAnredeImAdressfenster([Geschlecht],[Vorname],[Nachname])");
```

Alternative zu LINQ: FromSqlRaw()

- Wenn sich eine Abfrage nicht in LINQ formulieren lässt, oder wenn sich der zu einer LINQ-Abfrage generierte SQL-Code als ineffektiv erweist, dann kommt bei einer relationalen Datenbank ein mit Hilfe der Erweiterungsmethode FromSqlRaw() aus der statischen Klasse RelationalQueryableExtensions (im Namensraum Microsoft.EntityFrameworkCore) an die Datenbank gesendetes SQL-Kommando in Frage, sofern der EF Core - Provider dies unterstützt.

Die Einrichtung und Nutzung einer View

- Beispiel: Wir wollen eine View in der Datenbank generieren und dann per LINQ auswerten, die die Projekte mit den Mitarbeitern in aggregierter Form ausgibt

```
create view ProjekteMitMitarbeiternAgg As
select p.ProjektId, p.projektbezeichnung [Projekt],p.projektstart,p.Projektende,p.Fakturierbar,
ps.Projektstatus, pt.Projekttyp,
STRING_AGG(concat (m.Nachname,', ', m.Vorname),'; ') as Mitarbeiterliste
from projektmitarbeiter pm
inner join projekt p on (p.ProjektId=pm.ProjektId)
inner join Projektstatus ps on (p.projektstatusId=ps.projektstatusId)
inner join Projekttyp pt on (pt.projekttypId=p.projekttypId)
inner join Mitarbeiter m on (m.MitarbeiterId=pm.MitarbeiterId)
group by
p.ProjektId, p.projektbezeichnung,p.Projektstart,p.Projektende,p.Fakturierbar,
ps.Projektstatus, pt.Projekttyp
```

- Damit erhalten wir eine übersichtliche Liste der Projekte, bei denen in einem Feld alle Mitarbeiter pro Projekt aufgelistet sind
- Hierzu müssen wir folgende Schritte gehen:
  - Einrichtung eines Records ProjekteMitMitarbeiternAgg, die die obigen Feldnamen als Properties enthält
  - In die DBContext-Klasse wird public virtual DbSet<ProjekteMitMitarbeiternAgg> ProjekteMitMitarbeiternAggs { get; set; } eingefügt
  - In die Methode OnModelCreating wird eingefügt

```
modelBuilder
  .Entity<ProjekteMitMitarbeiternAgg>()
  .ToView(nameof(ProjekteMitMitarbeiternAggs))
  .HasKey(t => t.ProjektId);
```