Einführung in Entity Framework Core

Die Nutzung von Sichten, Funktionen und Prozeduren

Es wird getestet, ob und wie Sichten (Views) und Datenbankfunktionen (Anrede, Alter, Dienstjahre), eine Tabellenwertfunktion sowie eine Stored Procedure genutzt werden können, um die Vorzüge von Datenbanklösungen nutzen zu können.

Abschließend werden Möglichkeiten zur Abfrage skalarer (Nicht-Entitäts-) Typen erkundet, die mit EFC 7 eingeführt und mit EFC 8 erweitert wurden

Dozent: Dr. Thomas Hager, Berlin,
im Auftrag der Firma GFU Cyrus AG
20.10.2025 – 22.10.2025

Die Nutzung von Sichten (Views), Datenbank-Funktionen (Functions) und Berechneten Spalten (Computed Columns)

- Das Anlegen von Views, Funktionen, Stored Procdures und Berechneten Spalten in einer Datenbank hat das Ziel, aus den verfügbaren Daten zusammengesetzte/berechnete Daten zu gewinnen. Das hat vor allem bei immer wieder auftretenden Fragestellungen Rationalisierungseffekte, insbesondere, wenn man die Datenbank in ganz unterschiedlichen Applikationen nutzen will.
- Man kann spezielle Views und Funktionen auch mit C#-Mitteln erzeugen, lokal codieren. Das hat aber den Nachteil, dass z.B. bei Views erst alle Felddaten der beteiligten Tabellen über das Netz in den lokalen Arbeitsspeicher transportiert werden, statt nur die für einen bestimmten Zweck benötigten.

Verschiedene Möglichkeiten, Abfrageergebnisse in auswertbare Strukturen zu übernehmen

```
class MitarbeiterEinfach
{   public int Id { get; set; }
    public string Nachname { get; set; }
    public string Vorname { get; set; }
    public string Anrede { get; set;}
}


struct MitarbeiterStruct
{
    public string Nachname { get; set; }
    public string Vorname { get; set; }
    public string Anrede { get; set;}
}

record MitarbeiterRecord
(
    string Nachname,
    string Vorname,
    string Anrede
);
```

```
using (var dbctx = new EFCTest03DBContext())
{
    //Anonyme Klasse
    IQueryable<string> queryable = dbctx.MitarbeiterListe
          .Select(x => x.Nachname + ", " + x.Vorname);
    var mittiesAnonym = queryable;

    //Struct-Typ
    var mittiesStruct = dbctx.MitarbeiterListe
          .Select(x => new MitarbeiterStruct {
                Nachname = x.Nachname, Vorname = x.Vorname});

    //Tupel-Typ
    //var mittiesTuple = (dbctx.MitarbeiterListe.Select(x => (x.Nachname,
    //x.Vorname )));
    Tuple<int, string, bool> tuple = new Tuple<int, string, bool>(1, "Katze", true);
    Display(tuple.Item1.ToString());
    Display(tuple.Item2.ToString());

    //Record-Typ
    var mittiesRecord = dbctx.MitarbeiterListe
          .Select(x => new MitarbeiterRecord ( x.Nachname, x.Vorname,
          EFCTest03DBContext.Anrede(x.Geschlecht.ToString())));

}
```

Die Select-Funktion in LINQ

- Die Abfrage erzeugt eine große Anzahl an Feldern, die oft für die weitere Auswertung gar nicht benötigt werden:

```
var mitties =  dbctx.MitarbeiterListe
          .Include(x => x.Niederlassung)
          .Include(x => x.Funktion)
          .Include(x => x.Abteilung)
```

- Mit Hilfe einer Select-Anweisung kann die Menge der Felder eingeschränkt werden (so genannte Projektion).

```
var mittiesEingeschraenkt =  dbctx.MitarbeiterListe
   .Include(x => x.Abteilung).Include(x => x.Funktion).Include(x => x.Niederlassung)
   .Select (x => new MitarbeiterAuswahl
          (
              x.MitarbeiterId, x.Nachname + ',' + x.Vorname,
              x.Niederlassung.Niederlassungvorwahl + '-' + x.Durchwahl, //Telefonberechnung
              x.Abteilung.Abteilungsbezeichnung, .Funktion.Funktionsbezeichnung,
             x.Niederlassung.Niederlassungsbezeichnung)
          );
```

- Hier kann man auf einen lokal oder in der Lib.Models definierten Record zurückgreifen, der die Daten aufnimmt und dann für Auswertungen (Tabelle, Berichte) genutzt werden kann:

```
private record MitarbeiterAuswahl(int MitarbeiterId, string Name, string Telefon,
          string Funktion, string Abteilung, string Niederlassung);
```

Nutzung einer View (Sicht)

- Wir legen in der Datenbank EFCTest03 auf dem SQL-Server eine View an. Sie soll für jede Abteilung eine aktuelle E-Mailliste generieren:

```
create view vAbteilungEMailListe as
select a.AbteilungId, a.Abteilung, STRING_AGG (m.Email,',') as EmailListe
from mitarbeiter m
inner join Abteilung a on a.AbteilungId=m.AbteilungId
group by a.Abteilung, a.AbteilungId

select * from vAbteilungEMailListe;
```

AbteilungId	Abteilung	EmailListe
1	Geschäftsleitung/RW	gpaul@projektdb.de,kunterlauf@projektdb.de,eschief...
2	Infrastruktur	ogeist@projektdb.de,hwendisch@projektdb.de,gkoritz...
3	Marketing	klorenz@projektdb.de,amuehle@projektdb.de
4	Produktion	sarendt@projektdb.de,oheinrich@projektdb.de,hrostig...
5	Vertrieb	heinz@haase.de,gthurow@projektdb.de,sfischer@pro...

- Dieses Ergebnis könnten wir auch mit C#-Mitteln erreichen, das wäre aber deutlich aufwändiger.
- Wie können wir diese Sicht in unserer App nutzen?

Nutzung der Datenbanksicht

- Wir richten unter Models eine Klasse/einen Record mit den Feldnamen der View ein

- In der DBContext-Klasse bilden wir einen DBSet, der auf diese Klasse/Record zugreift und den Namen der Sicht erhält

- Zum Testen gehen wir in die Consolen-App:

```
public record AbteilungEMails(
[Key]
int AbteilungId,
string Abteilung,
string EMailListe);


public virtual DbSet<AbteilungEMails>
          vAbteilungEMailListe { get; set; }
```

```
var abtMitEMails = dbctx.vAbteilungEMailListe.AsNoTracking().ToList();  //Abruf der View

Display(dbctx.vAbteilungEMailListe.ToQueryString());                    //Zeigt das SQL-Statement

abtMitEMails.ForEach(a => Display($"{a.Abteilung}: {a.EMailListe}"));   //Zeigt die Ergebnisse
```

Nutzung von Funktionen – Voraussetzungen in der Datenbank

- In der DB entwickeln wir mehrere Funktionen (udf: user defined function), die helfen, typische Aufgaben zu lösen: Alter eines Mitarbeiters, Dienstjahre, Anrede und vollständiger Name in Abhängigkeit vom Geschlecht usw.

```
create function dbo.udfBerechneAlter(@dt date)
returns int
begin
  declare @alter int = datediff(year,@dt,getdate());
  declare @dtgeburtstagInDiesemJahr date = dateadd(year,@alter, @dt);
  if @dtgeburtstagInDiesemJahr > getdate()
     set @alter=@alter-1;
  return @alter;
end;
```

```
create function dbo.udfBerechneDienstjahre(@eintrittsdatum date,
@austrittsdatum date = getdate())
returns int
begin
  declare @dienstjahre int=0;
  if @eintrittsdatum is not null
   begin
    if @austrittsdatum is null
      set @dienstjahre = datediff(year,@eintrittsdatum,getdate());
    else
     set @dienstjahre = datediff(year,@eintrittsdatum,@austrittsdatum);
   end
  return @dienstjahre;
end;
```

```
create function dbo.udfBestimmeAnrede(@geschlecht
varchar(1))
returns varchar(10)
begin
  declare @anrede varchar(10);
  if @geschlecht ='m'
    set @anrede ='Herr';
   else if @geschlecht ='f' or @geschlecht ='w'
     set @anrede ='Frau';
   else
     set @anrede ='';
   return @anrede;
end
```

Nutzung von Funktionen – DBContext-Klasse

- In die DBContext-Klasse werden Bezüge zu den Datenbank-Funktionen als eigenständige öffentliche Methoden eingebaut:

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

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

[DbFunction("udfBestimmeAnrede", "dbo")]
public static string Anrede(string Geschlecht)
            => throw new InvalidOperationException($"{nameof(Anrede)} hat zu einem Fehler geführt");
```

Nutzung von Funktionen in Abfragen

- In der Anwendung kann jetzt auf die Datenbank-Funktionen zugegriffen werden:

```
Display("Mitarbeiter mit Anrede, Altersberechnung und Dienstjahrberechnung per Datenbank-Funktionen:");
var mittiesMitDBFunktion2 =
   dbctx.MitarbeiterListe
     .Include(x => x.Niederlassung)
     .Include(x => x.Funktion)
     .Include(x => x.Abteilung)
     .Select(m => new
                  {
                      Anrede = EFCTest03DBContext.Anrede(m.Geschlecht),
                      Name = m.Vorname + ' ' + m.Nachname,
                      Abteilung = m.Abteilung.Abteilungsbezeichnung,
                      Funktion = m.Funktion.Funktionsbezeichnung,
                      Datum_Eintritt= m.DatumEintritt,
                      Dienstjahre = EFCTest03DBContext.Dienstjahre(m.DatumEintritt,m.DatumAustritt)
                  });
```

Hier wird eine anonyme Klasse deklariert

Abfragen mit einem Record

- Statt eine anonyme Klasse (Select new { … }) zu nutzen, kann auch ein vorbereiteter Record zur Sammlung der Felder eingesetzt werden:

```
private record MitarbeiterAuswahl(
            int MitarbeiterId, string Name, string Anrede,
             int Dienstjahre, string Telefon, string Funktion,
            string Abteilung, string Niederlassung);
```

```
var mitListeMitRecord = dbctx.MitarbeiterListe
                  .Include(m => m.Funktion)
                  .Include(m => m.Niederlassung)
                  .OrderBy(m => m.Nachname)
                  //Record
                  .Select(m => new MitarbeiterAuswahl(
                      m.MitarbeiterId,
                      m.Vorname + ' ' + m.Nachname,
                       EFCTest03DBContext.Dienstjahre(m.DatumEintritt, m.DatumAustritt),
                      m.Niederlassung.Niederlassungvorwahl + '-' + m.Durchwahl,
                      m.Funktion.Funktionsbezeichnung,
                      m.Abteilung.Abteilungsbezeichnung,
                      m.Niederlassung.Niederlassungsbezeichnung
                  ));
```

Hier wird ein Record genutzt

Beispiel für die Nutzung einer Tabellenwertfunktion

- Eine zur Datenbank EFCTest03 implementierte Tabellenwertfunktion `dbo.udfProjektentwicklung` zeigt die an einem ausgewählten Projekt Mitarbeitenden mit ihrem jeweiligen Einsatzzeitraum
- Zur Aufnahme der Daten wird ein Record `ProjektentwicklungDTO` eingerichtet

```
public record ProjektentwicklungDTO
{
    [Key]
    public int id { get; set; }
    public string projekt { get; set; }
    public string mitarbeiter { get;set; }
    public DateTime beginn { get; set; }
    public DateTime ende { get; set; }
    public DateTime mitarbeiterbeginn { get; set; }
    public DateTime mitarbeiterende { get; set; }
}
```

```
using (var dbctx = new EFCTest03DBContext())
{
    int projektId = 24;
    Display($"Projektentwicklung für das Projekt mit der ID = {projektId}:");
    var proj = dbctx.ProjektentwicklungDTOListe.FromSql($"select * from
          dbo.udfProjektentwicklung({projektId})");
    foreach (var p in proj)
    {
        Display($"{p.projekt}, {p.mitarbeiter}, {p.mitarbeiterbeginn.ToShortDateString()},
          {p.mitarbeiterende.ToShortDateString()}");
    }
}
```

Nutzung von Stored Procedures

- Auch die Nutzung von SPs, die auf dem Datenbank-Server möglicherweise viel schneller Ergebnisse liefern als vergleichbare C#-Programmbausteine in der App, ist möglich.
- Hierzu siehe z.B. How to Execute Stored Procedures With EF Core 7 - Code Maze (code-maze.com)
- Im Beispiel wird eine SP aufgerufen, die die Mitarbeiter einer Abteilung zurückgibt:

```
using (var dbctx = new ProjektDBContext())
{
    int abteilungId = 1;
    Display($"Mitarbeiter in der Abteilung mit der ID = {abteilungId}:");
    var abtMit = dbctx.MitarbeiterListe.FromSql($"EXEC dbo.uspGetMitarbeiterAbteilung {abteilungId}");
    foreach (var m in abtMit)
    {
        Display($"{m.AbteilungId}, {m.Nachname}, {m.Vorname}");
    }

}
```

Der „direkte“ Zugriff auf Datenbankobjekte

- Während der Zugriff auf Sichten oder Tabellenwertfunktionen oder Store Procedures über die jeweilige DbSet<Entity> EntityListe erfolgt (dbctx.EntityListe.FromSql(…) usw.) , gibt es auch die Möglichkeit, direkt auf Datenbankobjekte einzuwirken über dbctx.Database.ExecuteSql(…) oder Inhalte aus Tabellen oder Sichten abzurufen mittels dbctx.Database.SqlQuery<T>(…) .
- In EFC 7 wurde die Technik `dbctx.Database.SqlQuery<T>` eingeführt, T ist ein primitiver Datentyp. In EFC 8 wird diese Beschränkung aufgehoben. T kann jetzt eine Entity sein.
- Beispiel: In der Datenbank gibt es eine Tabelle Test mit den Feldern TestId und TestText. Eine Stored Procedure ermöglicht Insert- oder Update-Operationen mit dieser Tabelle.

```
CREATE PROCEDURE [dbo].[uspInsertOrUpdateTest] (@testId int, @testtext nvarchar(50))
AS
BEGIN
	SET NOCOUNT ON;
	DECLARE @anz int=0;
	SET @anz = (SELECT count(*) FROM dbo.Test WHERE testId=@testId);
	IF @anz = 0
		INSERT INTO dbo.Test (testtext) VALUES (@testtext);
	else
		UPDATE dbo.Test SET testtext = @testtext WHERE testId=@testId;
END
```

Der „direkte“ Zugriff auf Datenbankobjekte

- Während der Zugriff auf Sichten oder Tabellenwertfunktionen oder Store Procedures über die jeweilige DbSet<Entity> EntityListe erfolgt (dbctx.EntityListe.FromSql(…) usw.) , gibt es auch die Möglichkeit, direkt auf Datenbankobjekte einzuwirken über dbctx.Database.ExecuteSql(…) oder Inhalte aus Tabellen oder Sichten abzurufen mittels dbctx.Database.SqlQuery<T>(…) .
- In EFC 7 wurde die Technik `dbctx.Database.SqlQuery<T>` eingeführt, T ist ein primitiver Datentyp. In EFC 8 wird diese Beschränkung aufgehoben.

```
using (var dbctx = new EFC03DBContext())
{
    for (int testId=1;i<=5;i++)
    {
          string testText = $"Das ist der {testId}. Insert-Test!";
          Display($"Insert oder Update der Tabelle dbo.test mit der ID = {testId}:");
          var returnWert = dbctx.Database.ExecuteSql($"dbo.uspInsertOrUpdateTest @testId = {testId}, @testtext = {testText}");
          Display($"Rückgabe: {returnWert} ");
    }
    Display("Abruf der Daten:");
    var liste = dbctx.Database.SqlQuery<Test>($"Select testId, testText from dbo.test").ToList();
    foreach (var item in liste)
    {
          Display($"{item.testId}, {item.testtext}");
    }
}
```

```
public record Test {
    [Key]
    public int TestId { get; set; }
    public string TestText{ get; set; }
}
```

Transactions

- Die DBContext-Klasse ermöglicht auch die Durchführung von Transaktionen
- Verschiedene, aufeinanderfolgende Delete-, Update, und/oder Insert-Operationen, die nur als gemeinsamer Block durchgeführt werden dürfen
- Im Beispiel wird ein neues Projekt pNeu angelegt es werden die zugehörigen Mitarbeiter (mitListe) zugeordnet

```
public static void Transaktionsbeispiel(Projekt pNeu,
          List<Mitarbeiter> mitListe)
{
    using (var dbctx = new EFCTest03DBContext())
    {
        using (var transaction = dbctx.Database.BeginTransaction())
        {
            try
            {
                dbctx.Add(pNeu);
                foreach (var m in mitListe)
                {
                    Projektmitarbeiter pm = new Projektmitarbeiter()
                    {
                        Projekt = pNeu,
                        Mitarbeiter = m,
                        Beginn = pNeu.Projektstart
                    };
                    dbctx.Add(pm);
                }
                dbctx.SaveChanges();
                transaction.Commit();
            }
            catch (Exception ex)
            {
                Display(ex.Message);
                transaction.Rollback();
            }
        }
    }
}
```