

Einführung in Entity Framework Core 9

Ein erstes Beispiel

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

In diesem Abschnitt wird ein erstes relativ einfaches Projekt entwickelt. Es besteht aus einer Konsolen-Applikation und einer Klassenbibliothek (EFC01.ConsApp und EFC01.Lib).

Es wird gezeigt, wie die erforderlichen Entity Framework Core – und Microsoft Extension – Pakete mit Hilfe des NuGet-Paketmanagers installiert werden.

Nach dem Entwurf von Modell-Klassen (Abteilung und Mitarbeiter) sowie einer DbContext-Klasse auf der Basis der Entity Framework Core - Technologie wird gezeigt, wie per Forward Engineering die Datenbank-Tabellen generiert werden.

Mit Hilfe von Annotationen zu den Klasseneigenschaften und einer darauf folgenden erneuten Migration können die Feldeigenschaften weiter präzisiert werden.

Nachdem Testdaten hinzugefügt wurden und mit einer erneuten Migration die Datenbanktabellen gefüllt wurden, lassen sich CRUD-Operationen mit den Daten durchführen. Dabei wird gezeigt, wie Abfragen mit Hilfe von LINQ-To-Entity realisiert werden.

Wir erweitern die Konfiguration: In der Datei appsettings.json geben wir denConnectionString zum Aufbau der Verbindung zur SQL-Server-Datenbank an.

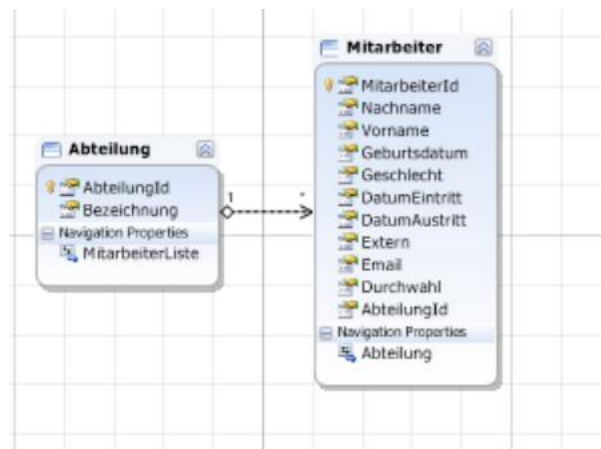


Einrichtung eines neuen Projekts in der Projektmappe EFCoreWorkshop

- Wir hatten eine Projektmappe EFCWorkshop eingerichtet, die unsere einzelnen Projekte und Bibliotheken sammeln soll
- Wir werden fortlaufend EFC01, EFC02 ... mit Konsolen-Applikationen einrichten und jeweils komplexere Fragestellungen (Konfigurationsmöglichkeiten, zusätzliche Entitäten, Beziehungen, Zugriff auf Datenbankobjekte usw.) untersuchen
- Innerhalb dieser Projektmappe sollen die einzelnen Projekte angelegt werden, und zwar immer eine Konsolen-Applikation mit dem Namen EFCxy.ConsApp und eine Klassen-Bibliothek mit dem Namen EFCxy.Lib
- Die Program.cs in der jeweiligen ConsApp benennen wir in Main_EFCxy.cs um

Ein einfaches Beispiel

- Wir entwickeln im Folgenden eine Applikation mit zwei einfachen Klassen (Abteilung) und (Mitarbeiter) und deren Beziehung (Abteilung – 1:n – Mitarbeiter). Mit diesen Klassen werden wir experimentieren, z.B. ein Forward-Engineering durchführen und verschiedene Operationen (CRUD) durchführen.
- Die beiden Entitäten sind Bestandteil eines größeren Projekts mit dem Namen ProjektDB. Inhaltlich steht dahinter ein fiktives Unternehmen (ein Softwarehaus, ein Architekturbüro), das Projekte mit und bei externen Auftraggebern abwickelt und dessen Mitarbeiter in diesen Projekten entsprechend ihrer Verfügbarkeit und ihrer Qualifikation eingesetzt werden



Forward Engineering
(Code First)

	Spaltenname	Datentyp	NULL-Werte zulassen
🔑	abteilungId	int	<input type="checkbox"/>
▶	bezeichnung	varchar(45)	<input checked="" type="checkbox"/>


	Spaltenname	Datentyp	NULL-Werte zulassen
🔑	mitarbeiterId	int	<input type="checkbox"/>
	prsnr	varchar(4)	<input checked="" type="checkbox"/>
	nachname	nvarchar(50)	<input checked="" type="checkbox"/>
	vorname	nvarchar(50)	<input checked="" type="checkbox"/>
	geschlecht	varchar(1)	<input checked="" type="checkbox"/>
	datum_eintritt	date	<input checked="" type="checkbox"/>
	datum_austritt	date	<input checked="" type="checkbox"/>
	extern	bit	<input checked="" type="checkbox"/>
	email	varchar(50)	<input checked="" type="checkbox"/>
	durchwahl	varchar(50)	<input checked="" type="checkbox"/>
	abteilungId	int	<input checked="" type="checkbox"/>

Einrichten eines neuen Konsolen-Projekts

EFC01.ConsApp

Neues Projekt hinzufügen

Zuletzt verwendete Projektvorlagen

 Konsolen-App

C#

Neues Projekt konfigurieren

Konsolen-App

C#

Linux

macOS

Windows

Konsole

Projektname

EFC01.ConsApp

Ort

E:\Kurse\GFU NET Core\Projekte\EFC01.ConsApp

Projekt wird in „E:\Kurse\GFU NET Core\Projekte\EFC01.ConsApp“ erstellt

Weitere Informationen

Konsolen-App


C#

Linux


macOS

Windows

Konsole

Framework 

.NET 7.0 (Standard-Laufzeitunterstützung)

☒ Keine Anweisungen der obersten Ebene verwenden 

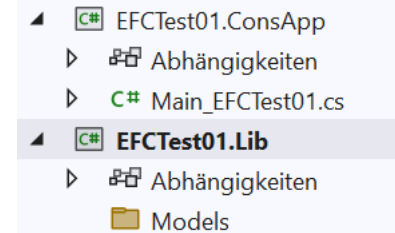
Wählen Sie Konsolen-App aus der Fülle der Vorlagen

Wählen Sie eine nachvollziehbare Bezeichnung

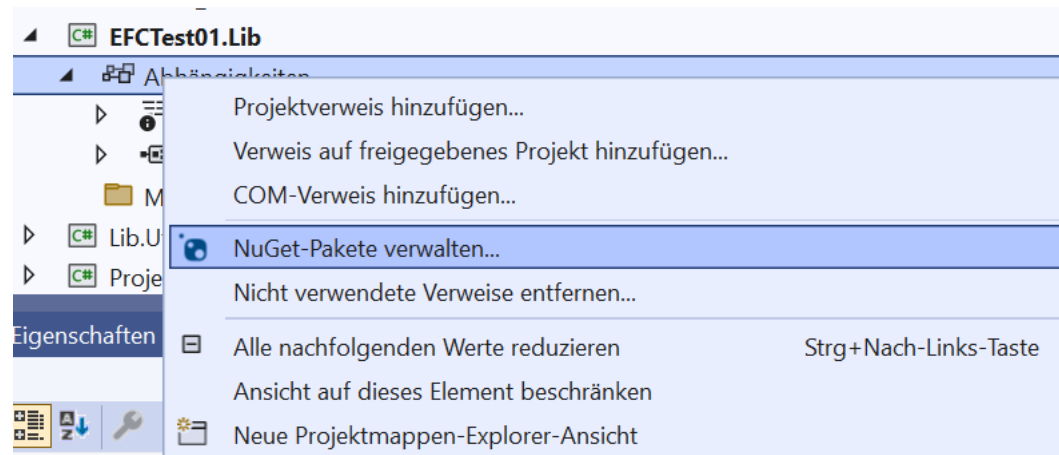
Um den „klassischen“ Stil nutzen zu können (Bisherige Klasse program mit main(args...) usw.) sollte der Haken bei „Keine Anweisungen der obersten Ebene verwenden“ gesetzt werden

Nach dem Erstellen:

```
namespace EFC01.ConsApp
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Display("Hello, World!");
        }
    }
}
```



Außerdem fügen wir der Projekt-Mappe die Klassenbibliothek EFC01.Lib hinzu. In dieser Bibliothek gehen wir mit der rechten Maustaste auf den Applikationstitel oder auf Abhängigkeiten und wählen im Kontext-Menü NuGet-Pakete verwalten:



Der Nuget-Paket-Manager

Wählen Sie „Durchsuchen“ und installieren Sie die Pakete:

Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.EntityFrameworkCore.Tools
Microsoft.EntityFrameworkCore.Design

Microsoft.Extensions.Configuration
Microsoft.Extensions.Configuration.EnvironmentVariables
Microsoft.Extensions.Configuration.Json
Microsoft.Extensions.Logging
Microsoft.Extensions.Logging.Console

Durchsuchen

Installiert


Wählen Sie die „Aktuellste stabile Version“:

Microsoft.entity



Vorabversion einbeziehen



Microsoft.EntityFrameworkCore  durch Microsoft, **669M** Downloads 7.0.5
Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL S...



Microsoft.EntityFrameworkCore 

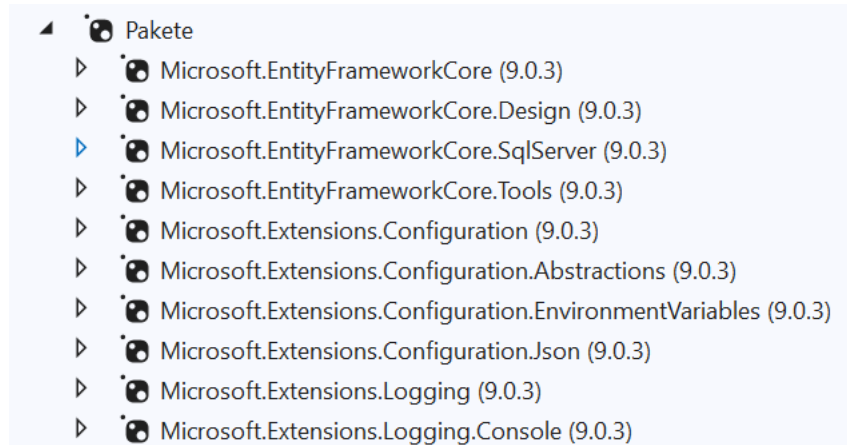


Version: Aktuellste stabile Version 7.0.5

Installieren

Nach der Installation der Pakete

- Unter Abhängigkeiten erkennt man, welche Pakete installiert wurden



Entity Framework Core ist ein moderner Objekt-Datenbank-Mapper für .NET. Er unterstützt LINQ-Abfragen, Änderungsverfolgung, Updates und Schemamigrationen. EF Core arbeitet mit SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, PostgreSQL und anderen Datenbanken über eine Provider Plugin API.

Extensions ermöglicht zum Beispiel die Auswertung einer externen Konfigurationsdatei im Json-Format (appsettings.json) , das Logging usw.

Die beiden wichtigsten Pakete

- **Microsoft.EntityFrameworkCore.SqlServer**
 - Dieses Paket ist der Entity Framework Core-Datenbankprovider für Microsoft SQL Server und Azure SQL Database. Die wichtigsten Aufgaben:
 - Datenbank-Verbindung: Ermöglicht es EF Core, Verbindungen zu einer SQL Server-Datenbank herzustellen.
 - SQL-Erzeugung: Übersetzt die LINQ-Abfragen aus der Anwendung in SQL-Abfragen, die der SQL Server verstehen kann.
 - Schema-Erzeugung: Sie ist für die Übersetzung der Entitätsdatenmodelle in SQL Server-Datenbankschemata verantwortlich. Dies ist besonders nützlich bei der Erstellung und Migration von Datenbanken.
 - Optimierung der Leistung: Das Paket enthält SQL Server-spezifische Optimierungen, die die Leistung von Datenzugriffsoperationen verbessern.
- **Microsoft.EntityFrameworkCore.Tools**
 - Dieses Paket bietet zusätzliche Werkzeuge für die Arbeit mit Entity Framework Core, die besonders während der Entwicklung nützlich sind. Diese Werkzeuge vereinfachen viele datenbankbezogene Aufgaben:
 - Migrationen: Es sind Befehle zur Erstellung und Verwaltung von Migrationen enthalten, mit denen das Datenbankschema im Laufe der Zeit ohne Datenverlust weiterentwickelt werden kann.
 - Datenbank-Update: Dieses Tool ermöglicht die Anwendung von Migrationen zur Aktualisierung des Datenbankschemas direkt über die Befehlszeile oder über die Package Manager Console in Visual Studio.
 - Datenbankschema-Entwicklung: Es kann ein Datenbankschema zurückentwickeln, um Entitätsmodellklassen und einen DbContext auf der Grundlage einer bestehenden Datenbank zu erstellen, was besonders in Datenbank-First-Szenarien nützlich ist.
 - Skripterstellung: Generierung von SQL-Skripten aus Migrationen, die für manuelle Überprüfungen oder für die Bereitstellung von Datenbanken über Skripte nützlich sein können.

Die Projektdatei EFC01.Lib.csproj

Die Projektdatei *.csproj enthält eine Reihe von Voreinstellungen, die beim Anlegen der Applikation automatisch gesetzt werden sowie die nach der Installation durch NuGet verfügbaren Pakete.


Eine der Voreinstellungen sollte man ändern: statt `<Nullable>enable</Nullable>` sollte man setzen: `<Nullable>disable</Nullable>`. Dadurch werden überflüssige Warnhinweise unterdrückt, die bei der Klassenentwicklung stören.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="9.0.4" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="9.0.4">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Proxies" Version="9.0.4" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="9.0.4" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="9.0.4">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.Extensions.Configuration" Version="9.0.0" />
    <PackageReference Include="Microsoft.Extensions.Configuration.EnvironmentVariables" Version="9.0.0" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Json" Version="9.0.0" />
    <PackageReference Include="Microsoft.Extensions.Logging" Version="9.0.0" />
    <PackageReference Include="Microsoft.Extensions.Logging.Console" Version="9.0.0" />
  </ItemGroup>

</Project>
```

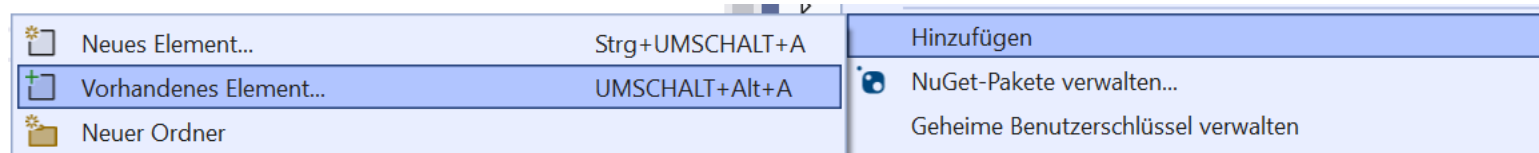


Im Folgenden:

EINRICHTUNG DER MODELL-KLASSEN UND DER DBCONTEXT-KLASSE

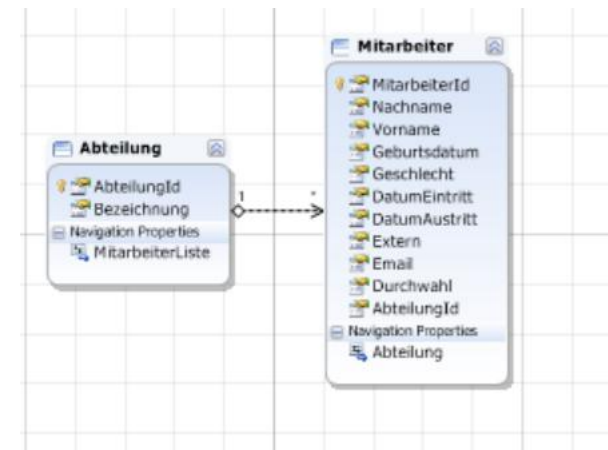
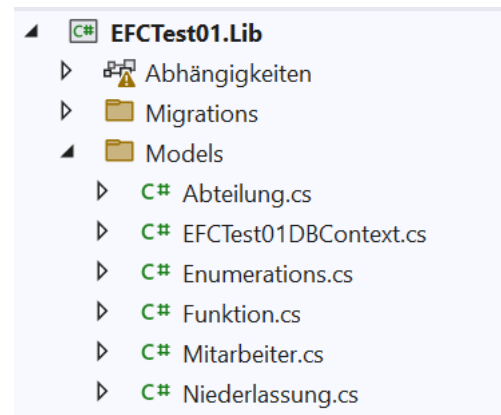
Einrichtung der Modellklassen und der DbContext-Klasse im Ordner Models

- Wir erzeugen im Projektmappenexplorer mittels Kontextmenü einen neuen Ordner und nennen ihn „Models“



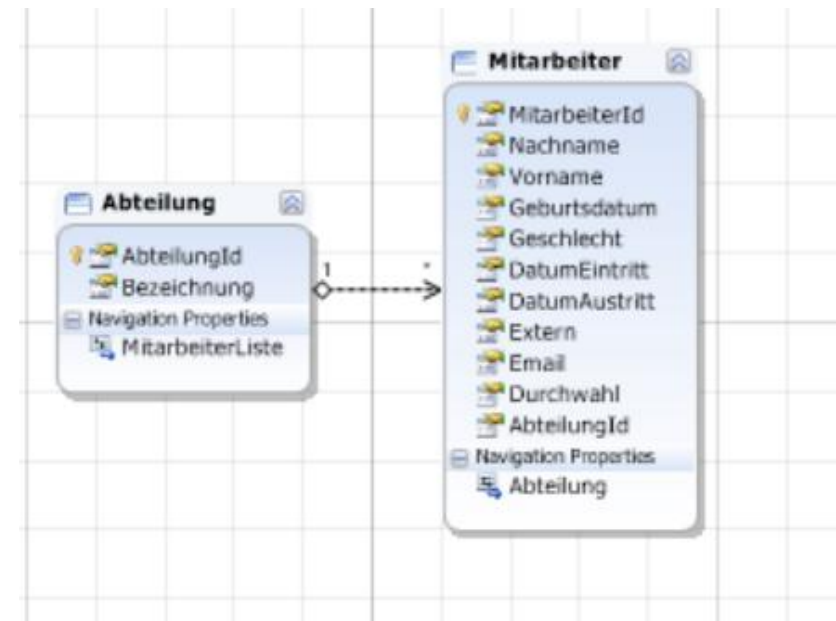
- In diesen Ordner fügen wir folgende Klassen hinzu:

- Funktion
- Abteilung
- Mitarbeiter
- EFC01DbContext



Die Master-Detail-Beziehung (Abteilung – 1:n – Mitarbeiter)

- Abteilung ist die Master-Entitätsklasse (assoziiert mit der Detail-Tabelle Mitarbeiter)
 - mit dem Primärschlüssel AbteilungId und
 - der Kollektions-Navigationseigenschaft MitarbeiterListe)
- Mitarbeiter ist die Details-Entitätsklasse (assoziiert mit der Master-Tabelle Abteilung)
 - mit dem Primärschlüssel MitarbeiterId und mit dem Fremdschlüssel AbteilungId
 - und der Referenz-Navigationseigenschaft Abteilung



DIE ENTITÄTSKLASSEN

Die Klasse Abteilung

- Beim Entwurf der Klasse wählen wir für die Properties den Modifikator virtual, um gegebenenfalls bei der Vererbung dieser Klasse das Überschreiben zu ermöglichen.

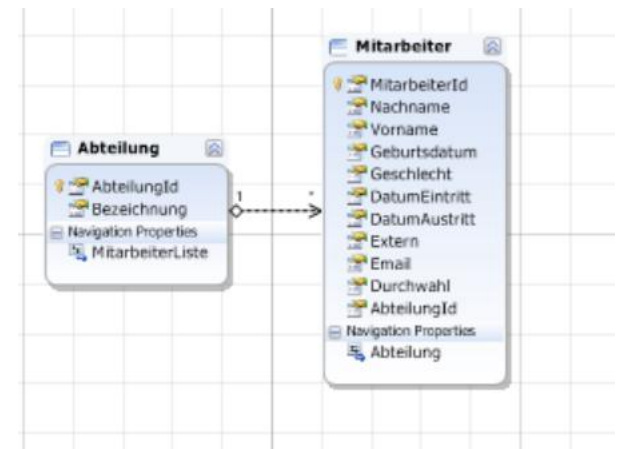
```
public partial class Abteilung
{
    public Abteilung()
    {
        this.MitarbeiterListe = new List<Mitarbeiter>();
    }

    public virtual int AbteilungId { get; set; }

    public virtual string Abteilungsbezeichnung { get; set; }

    public virtual IList<Mitarbeiter> MitarbeiterListe { get; }
}
```

So genannte Navigation Property

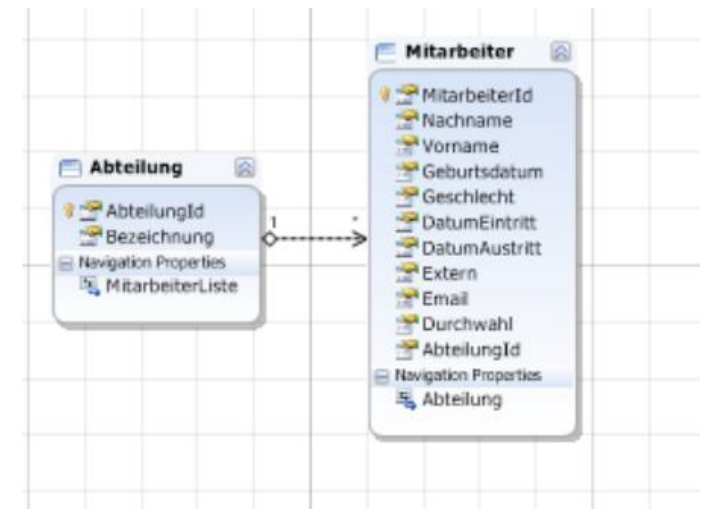


Die Klasse Mitarbeiter

```
public partial class Mitarbeiter
{
    public Mitarbeiter()
    {
    }

    public virtual int MitarbeiterId { get; set; }
    public virtual string Nachname { get; set; }
    public virtual string Vorname { get; set; }
    public virtual DateTime? Geburtsdatum { get; set; }
    public virtual string Geschlecht { get; set; }
    public virtual DateTime? Datum_Eintritt { get; set; }
    public virtual DateTime? Datum_Austritt { get; set; }
    public virtual bool? Extern { get; set; }
    public virtual string Email { get; set; }
    public virtual string Durchwahl { get; set; }

    public int AbteilungId { get; set; } //muss diskutiert werden
    //Navigationseigenschaft
    public virtual Abteilung Abteilung { get; set; }
}
```



Im Folgenden:

DIE DBCONTEXT-KLASSE

Allgemeines zur Klasse DBContext

- Eine Instanz der Klasse (in unserem Fall die Klasse EFC01DBContext, die von der Basisklasse DBContext erbt) repräsentiert eine Session mit der Datenbank.
- Sie kann genutzt werden, um Datenbank-Objekte zu generieren, abzufragen und Daten zu manipulieren (CRUD – Create, Read, Update, Delete)
- Dabei realisiert die Klasse zwei zentrale Muster (Pattern) der Datenbank-Entwicklung:
 - Unit of Work (UoW)
 - Repository-Pattern
- Wie wir sehen werden, steuert DBContext die Transaktionen, ohne dass wir hier wie noch bei ADO.NET etwas programmieren müssen.

Das Repository-Muster soll eine Abstraktionsschicht zwischen der Datenzugriffsschicht und der Geschäftslogikschicht einer Anwendung bilden. Es handelt sich um ein Datenzugriffsmuster, das einen eher lose gekoppelten Ansatz für den Datenzugriff vorsieht. Wir erstellen die Datenzugriffslogik in einer separaten Klasse oder einem Satz von Klassen, die als Repository bezeichnet werden und für die Persistierung des Geschäftsmodells der Anwendung verantwortlich sind.

Als Unit of Work wird eine einzelne Transaktion bezeichnet, die mehrere Operationen wie Einfügen/Aktualisieren/Löschen usw. umfasst. In einfachen Worten ausgedrückt bedeutet dies, dass für eine bestimmte Benutzeraktion (z. B. Registrierung auf einer Website) alle Transaktionen wie Einfügen/Aktualisieren/Löschen usw. in einer einzigen Transaktion durchgeführt werden, anstatt mehrere Datenbanktransaktionen durchzuführen. Das bedeutet, dass eine Arbeitseinheit hier Einfüge-, Aktualisierungs- und Löschvorgänge umfasst, die alle in einer einzigen Transaktion durchgeführt werden.

Wichtige Methoden der Klasse DbContext

Methode	Beschreibung
Add/AddAsync, AddRange	Ermöglicht das Einfügen der Entität in die Datenbank; beginnt die Verfolgung der Entität. Hinzugefügte Objekte werden nachverfolgt und hinzugefügt, sobald die Methode SaveChanges aufgerufen wird. Es gibt auch eine AddRange-Methode, die das gleichzeitige Hinzufügen von mehr als einer Entität ermöglicht.
Find/FindAsync	Findet eine spezifische Entity mit Hilfe des Primärschlüssel-Wertes (im Allg. Id).
OnConfiguring	Ermöglicht es uns, die Optionen und andere Informationen über die Datenbank zu überschreiben.
OnModelCreating	Ermöglicht die Verwendung der FluentAPI, um unsere Entitäten und ihre Beziehungen weiter zu definieren, indem die Modelle, ihre Eigenschaften und alle Beziehungen konfiguriert werden
Remove, RemoveRange	Löscht eine Entity aus der Datenbank. (auch RemoveRange)
SaveChanges/Save ChangesAsync	Anwendung der verfolgten Änderungen in einer einzigen Transaktion
Update UpdateRange	Wird verwendet, um eine Aktualisierung der verfolgten Entität durchzuführen. Es besteht auch die Möglichkeit, einen Bereich von verfolgten Entitäten mit UpdateRange zu aktualisieren

Die Klasse EFC01DBContext

```
public partial class EFC01DBContext: DbContext
{
    public EFC01DBContext()
    {
    }

    public EFC01DBContext(DbContextOptions<EFC01DBContext> options): base(options)
    {
    }

    public virtual DbSet<Abteilung> AbteilungListe {get;set;}
    public virtual DbSet<Funktion> FunktionListe {get;set;}
    public virtual DbSet<Mitarbeiter> MitarbeiterListe {get;set;}

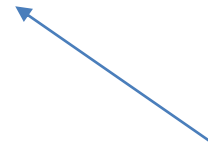
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
            optionsBuilder.UseSqlServer("data source=localhost;initial catalog=EFC01;Integrated Security=true;
                                         TrustServerCertificate=Yes;Encrypt=False; ");
        }
    }
}
```

← Vorläufig! Wir werden den Connectionstring später auslagern!

Die Aufgabe der DbSet-Methoden

- In der DbContext-Ableitung ist zu jeder Entitätsklasse (Konkretisierung des Typformalparameters TEntity) des EF Core - Modells (in unserem Fall also zunächst die Klassen Abteilung und Mitarbeiter) eine Eigenschaft vom Typ DbSet vorhanden
- Die Klasse DbSet<TEntity> implementiert die Schnittstelle IQueryable, so dass die in der statischen Klasse Queryable im Namensraum System.Linq definierten Erweiterungsmethoden genutzt werden können, um per LINQ-to-Entities Entitäten aus Datenbanken abzufragen bzw. dorthin zu sichern

```
public virtual DbSet<Abteilung> AbteilungListe { get; set; }  
public virtual DbSet<Funktion> FunktionListe { get; set; }  
  
public virtual DbSet<Mitarbeiter> MitarbeiterListe { get; set; }
```



Entitätsklasse

Die Methode OnModelCreating in der DbContext-Klasse

- Diese Methode dient der Ausführung von Anweisungen bei der Migration, z.B. zur Umsetzung der Properties aus den Entitätsklassen sowie deren Assoziationen per Fluid Api, aber auch zur Generierung von Testdaten.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    //Evtl. Mappingmethoden

    //Hier werden u.a. die Testdaten aufgerufen:
    Testdaten.TestdatenGenerierung_Abteilung(modelBuilder);
    Testdaten.TestdatenGenerierung_Mitarbeiter(modelBuilder);
}
```

Die Methode ConfiguringConventions in der DbContext-Klasse

- In C# 6 wurde die Datentypen DateOnly und TimeOnly eingeführt. Diese sollen mit den SQL Server-Datentypen date und time korrespondieren
- Um diesen Zusammenhang herzustellen, kann man in der DbContext-Klasse die Methode ConfigureConventions entsprechend einrichten:

```
protected override void ConfigureConventions(ModelConfigurationBuilder builder)
{
    builder.Properties<DateOnly>().HaveColumnType("date");
}
```

Im Folgenden:

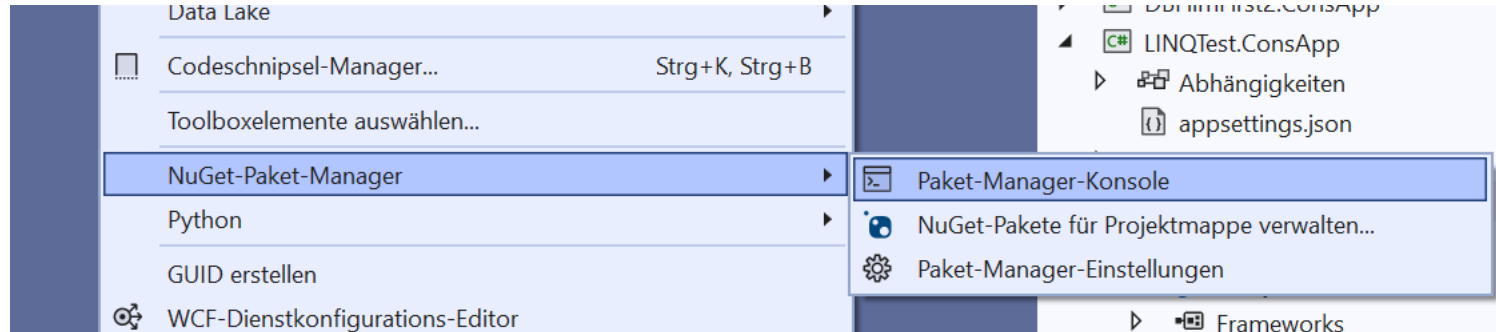
EINE ERSTE MIGRATION VOM MODELL ZUR DATENBANK (FORWARD ENGINEERING)

Erster Versuch einer Migration : Forward Engineering – Initiale Kreation

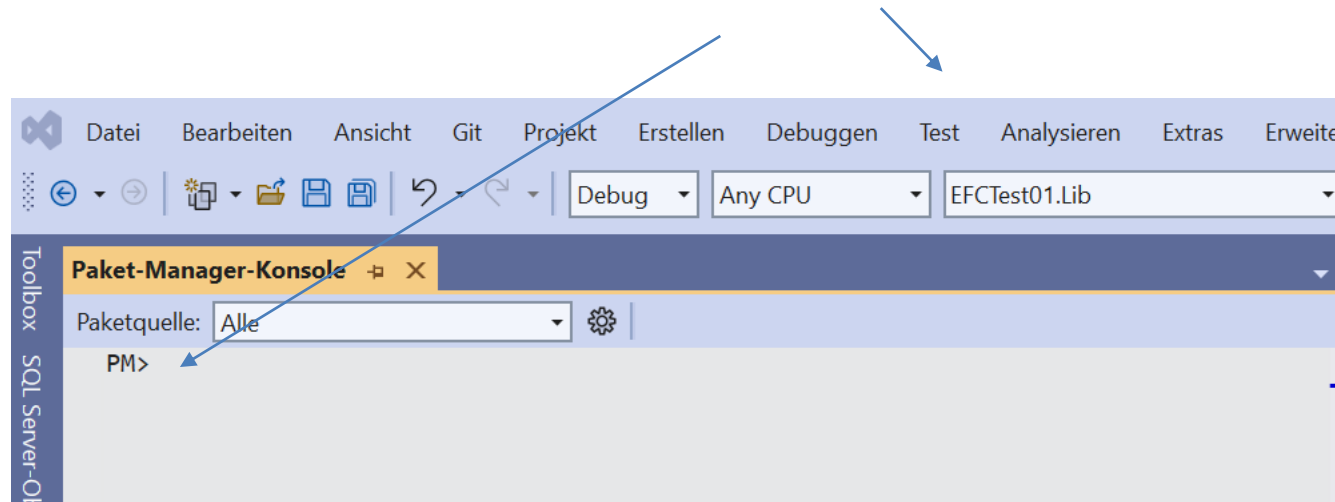
- Im Rahmen der EF Core – Technologie wird die Umsetzung eines Modells oder von Modelländerungen in eine Datenbankstruktur bzw. umgekehrt als **Migration** bezeichnet. Unterschieden werden
 - Forward Engineering (Model First) → Von den Modellklassen zu den Tabellen der geplanten (oder schon vorhandenen) Datenbank
 - Reverse Engineering (Database First) → Von den Tabellen einer Datenbank zu den Modellklassen (die entweder neu angelegt oder verändert werden)
- Auf der Grundlage der Entity-Klassen und der Festlegung des Connectionstrings können wir jetzt die Umsetzung des Modells in eine Datenbank EFC01 vornehmen.
- Hierzu gibt es folgende Möglichkeiten:
 - Im Visual Studio können Werkzeuge für die NuGet-Paket-Manager-Konsole verwendet werden, die in der englischen Literatur als PMC-Tools bezeichnet werden.
 - Die alternativen Konsolen-Werkzeuge werden in der englischen Literatur als CLI-Tools (Command Line Interface) bezeichnet.

Migration mit Hilfe der Paket-Manager-Console (PMC)

- In der Menü-Funktion Extras wählen wir NuGet-Paket-Manager und dann Paket-Manager-Konsole.

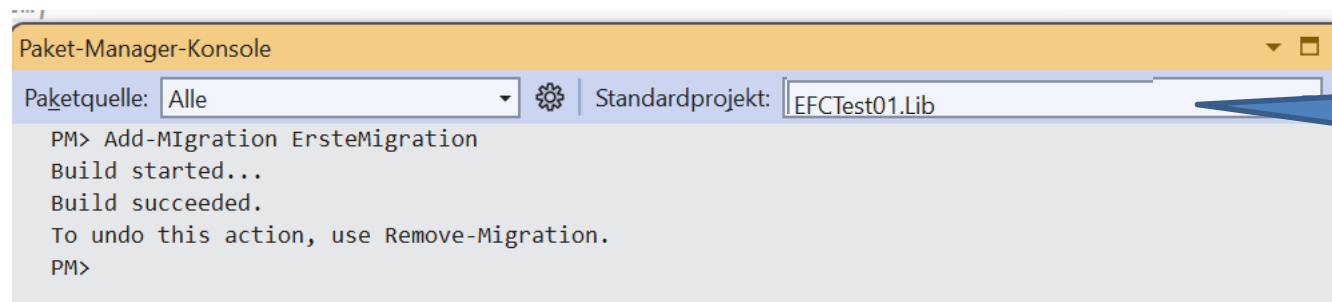


- Wir beachten, dass unser Bibliotheksprojekt als Standardprojekt angezeigt wird



Die Migration (Forward Engineering bzw. C#-Code First) erfolgt in zwei Schritten

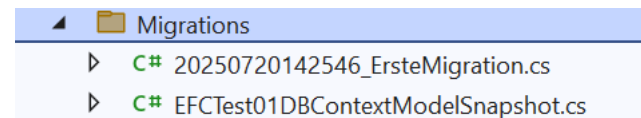
- Falls noch nicht vorhanden: Generieren Sie zunächst im Datenbankserver die Zieldatenbank (hier: EFC01)
- Wir wählen Extras > NuGet-Paket-Manager > Paket-Manager-Konsole und geben den Befehl Add-Migration <NameDerMigration> ein, z.B. **Add-Migration** ErsteMigration



```
PM> Add-Migration ErsteMigration
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>
```

Hier muss das aktuelle, als Startprojekt eingestellte Projekt erscheinen!

- Jetzt ist ein neuer Ordner namens Migrations mit zwei neuen Klassen, die den Migrationsstatus enthalten, eingefügt worden:



Wenn etwas nicht klappen sollte
... build failed ...
Fehlerliste analysieren mit Ctrl-Shift-B

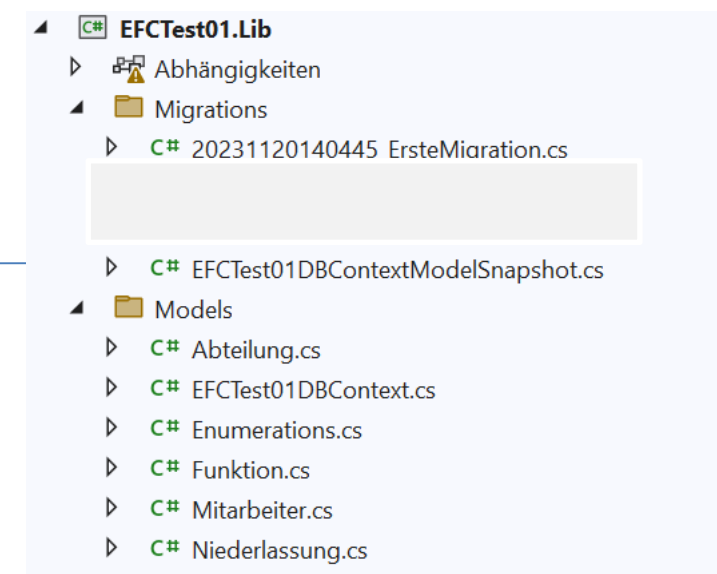
Ergebnis des ersten Schrittes der Migration: Die Methode Up

- Im Ordner Migrations befinden sich jetzt zwei Klassen. In der Klasse mit dem Zeitstempel und dem von uns vergebenen Namen befinden sich die beiden Methoden Up und Down
- Up enthält die Befehle zur Generierung der Tabellen, z.B.

```
migrationBuilder.CreateTable(  
    name: "Abteilung",  
    columns: table => new  
    {  
        AbteilungId = table.Column<int>(type: "int", nullable: false)  
            .Annotation("SqlServer:Identity", "1, 1"),  
        Bezeichnung = table.Column<string>(type: "nvarchar(max)", nullable: true)  
    },  
    constraints: table =>  
    {  
        table.PrimaryKey("PK_AbteilungListe", x => x.AbtteilungId);  
    }  
);
```



```
CREATE TABLE [dbo].[abteilung](  
    [abteilungId] [int] IDENTITY(1,1) NOT NULL,  
    [bezeichnung] [varchar](max) NULL)  
GO  
  
ALTER TABLE [dbo].[abteilung]  
ADD CONSTRAINT PK_abteilung PRIMARY KEY ([abteilungId])  
GO  
  
ALTER TABLE [dbo].[abteilung]  
ADD DEFAULT (NULL) FOR [bezeichnung]  
GO
```



- Hier erkennt man, in welche Weise die Umsetzung der Klasse Abteilung in eine SQL Server-Tabelle (hier abteilungsliste) vollzogen wird

Ergebnis des ersten Schrittes der Migration: Die Methode Down

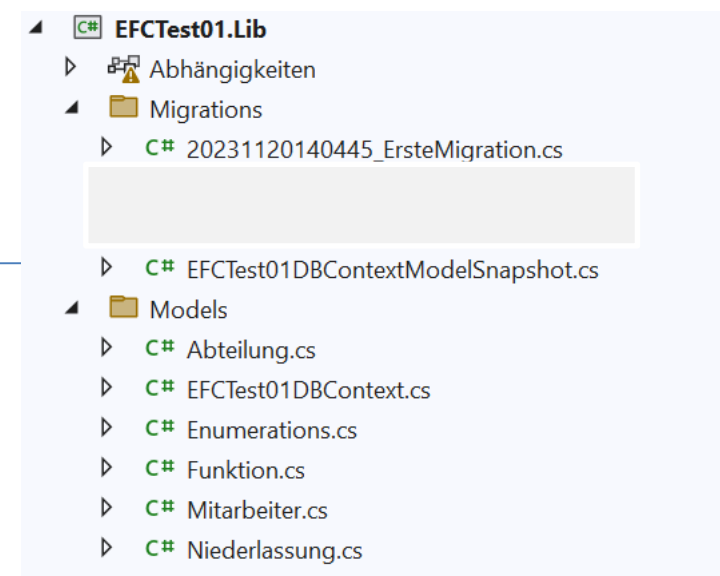
- Down enthält die Befehle zum Löschen der Tabellen

```
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(name: "MitarbeiterListe");
    migrationBuilder.DropTable(name: "AbteilungListe");
    migrationBuilder.DropTable(name: „FunktionListe“);
    migrationBuilder.DropTable(name: „NiederlassungListe“);
}
```



```
DROP TABLE [dbo].[abteilungsliste];
DROP TABLE [dbo].[mitarbeiterliste];
DROP TABLE [dbo].[funktionliste];
DROP TABLE [dbo].[niederlassungliste];
```

- Diese werden aktiviert, wenn der Befehl Remove-Migration eingegeben und vollzogen wird



Möglichkeit, zwischen add-migration (=Vorbereitung) und update-database (=Durchführung) ein Migrationsskript zu generieren

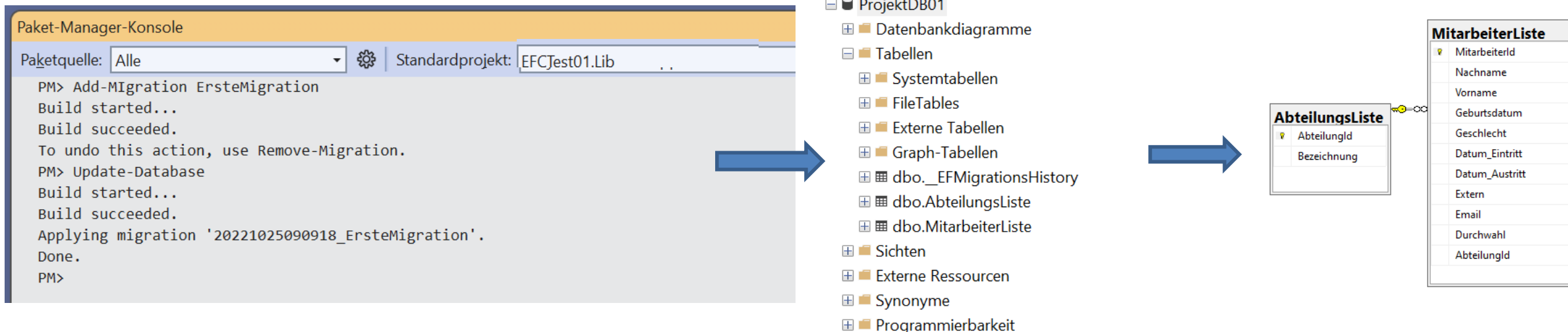
- Mit dem Befehl **Script-Migration** kann nach add-migration ein SQL-Skript erzeugt werden, das die Umsetzung der Entities in Befehle zur Generierung der Datenbanktabellen enthält.

```
CREATE TABLE [AbteilungListe] (  
    [AbteilungId] int NOT NULL IDENTITY,  
    [Abteilungsbezeichnung] nvarchar(max) NULL,  
    CONSTRAINT [PK_AbteilungListe] PRIMARY KEY ([AbteilungId])  
);  
CREATE TABLE [FunktionListe] (  
    [FunktionId] int NOT NULL IDENTITY,  
    [Funktionsbezeichnung] nvarchar(max) NULL,  
    CONSTRAINT [PK_FunktionListe] PRIMARY KEY ([FunktionId])  
);  
CREATE TABLE [MitarbeiterListe] (  
    [MitarbeiterId] int NOT NULL IDENTITY,  
    [Prsnr] nvarchar(max) NULL,  
    [Nachname] nvarchar(max) NULL,  
    [Vorname] nvarchar(max) NULL,  
    [Geburtsdatum] datetime2 NULL,  
    [Geschlecht] nvarchar(max) NULL,  
    [DatumEintritt] datetime2 NULL,  
    [DatumAustritt] datetime2 NULL,  
    [Extern] bit NULL,  
    [Email] nvarchar(max) NULL,  
    [Durchwahl] nvarchar(max) NULL,  
    [AbteilungId] int NOT NULL,  
    [FunktionId] int NOT NULL,  
    CONSTRAINT [PK_MitarbeiterListe] PRIMARY KEY ([MitarbeiterId]),  
    CONSTRAINT [FK_MitarbeiterListe_AbteilungListe_AbteilungId] FOREIGN KEY ([AbteilungId]) REFERENCES [AbteilungListe] ([AbteilungId]) ON DELETE CASCADE,  
    CONSTRAINT [FK_MitarbeiterListe_FunktionListe_FunktionId] FOREIGN KEY ([FunktionId]) REFERENCES [FunktionListe] ([FunktionId]) ON DELETE CASCADE  
);  
  
CREATE INDEX [IX_MitarbeiterListe_AbteilungId] ON [MitarbeiterListe] ([AbteilungId]);  
CREATE INDEX [IX_MitarbeiterListe_FunktionId] ON [MitarbeiterListe] ([FunktionId]);
```

Damit können wir prüfen,
ob die Befehle zur
Generierung der
Datenbankobjekte
unserem Konzept
entsprechen!

Abschluss der Migration

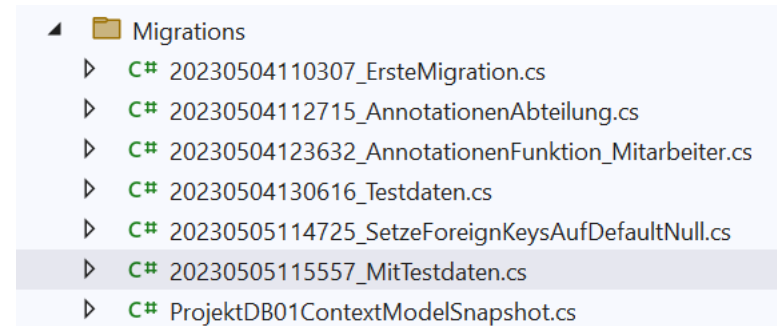
- Nachdem mit dem Befehl **Add-Migration** ErsteMigration die Migrationsklassen erzeugt wurden (und evtl. auch mit **Script-Migration** die SQL-Statements zur Generierung der Datenbank-Objekte geprüft wurden) kann mit dem zweiten Befehl **Update-Database** die Migration vollzogen werden



- Nach diesem Schritt erscheint die Datenbank im Objekt-Explorer des Managementstudios mit den beiden Tabellen. Nach Erzeugung eines Diagramms erkennt man auch, dass die 1:n-Beziehung auf Grund unserer Vorgaben (Navigationseigenschaften) zwischen der Abteilung und der Mitarbeiterliste korrekt erkannt wurde

Remove Migration

- Hat man nach der Realisierung einer Migration vor dem Befehl update-database nach einer Inspektion des SQL-Statements (script-migration) oder der Migrationsklasse ein Problem festgestellt, kann man die Migration rückgängig machen:

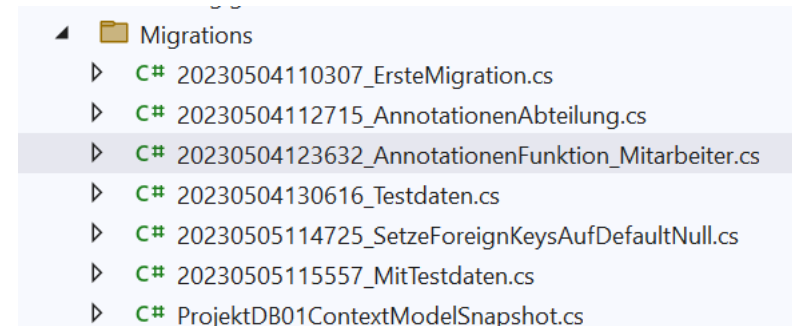


- Hierzu gibt man z.B. ein: `remove-migration MitTestdaten`
- Oder generell, um auch die generierte Datenbank zu entfernen:

```
remove-migration -force
```

Roll-Back

- Im Unterschied zum remove-Befehl (ohne –force) kann man mit einem Roll-Back nach dem update-database den Stand nach einer bereits realisierten Migration wiederherstellen.
- Hierzu wählt man aus dem Verzeichnis Migrations den Namen einer entsprechende Migration aus (der Name besteht aus dem Zeitstempel und dem ursprünglich vergebenen Namen).



- Beispiel: `update-database -migration 20230504123632_AnnotationenFunktion_Mitarbeiter`
- Will man sämtliche Änderungen „zurückrollen“, d.h. den Stand vor der ersten Migration wieder herstellen, so gibt man ein:

```
update-database -migration 0
```


Im Folgenden:

NACH DER ERSTEN MIGRATION: DISKUSSION DER ERGEBNISSE. KORREKTUREN MITTELS ANNOTATIONEN

Zwischenbilanz: Was auffällt

- Tabellennamen werden auf der Basis der DBSet<TEntity>-Namen (AbteilungListe usw.) gebildet, wenn nicht eine Annotation [Table(„tabellenname“)] vor die Klassendefinition gestellt wird
- Spaltennamen werden 1:1 übertragen.
- Primär- und Fremdschlüssel werden korrekt erkannt, die 1:n Beziehung wird korrekt aufgebaut – damit ist die referentielle Integrität der Daten gesichert. Wie ist das aber mit dem Löschen einer Abteilung? Werden dann auch alle Mitarbeiter gelöscht? Das wäre fatal!
- Die Identitätsspalten (AbteilungId und MitarbeiterId) werden erkannt und im Falle des jeweiligen Primärschlüssels wird die Eigenschaft IDENTITY(1,1), also die automatische Generierung des eindeutigen Schlüsselwertes umgesetzt!
- Die Übertragung der Datentypen int, bool funktioniert ohne Probleme. Wie ist das mit Default-Werten?
- Die Übertragung des Datentyps DateTime erfolgt durch Umsetzung zu datetime2(7). Das ist z.B. bei einem Geburtsdatum eigentlich eine überflüssige Genauigkeit. Evtl. sollte man für einfache Datumswerte (ohne Zeitangabe) datetime2(0) setzen. Den neuen Datentyp in C# DateOnly kennt erst EF Core 8
- Die Attribute mit dem Datentyp String werden alle auf nvarchar(max) gesetzt. Hier wäre eine Differenzierung sicher sinnvoll:
 - Nicht alle String-Properties verlangen den Unicode-Zeichensatz (z.B. PLZ, Telefonnummern, Versicherungsnummern, Geschlecht usw.)
 - Nur wenige String-Properties verlangen die Reservierung des maximalen Speicherplatzes, oft genügt eine kleine Zahl

Nach der ersten, nur einigermaßen gelungenen Migration: Abteilung

- Wir haben einige Probleme festgestellt, die wir jetzt beseitigen wollen. Hierzu nutzen wir die Möglichkeiten der Attribut-Setzung (System.ComponentModel.DataAnnotations;)
- Zunächst am Beispiel der Entitätsklasse Abteilung
 - Der Tabellename soll – wie in Datenbanken üblich – in der Einzahl gesetzt werden, also Abteilung
 - Die max. Länge des Feldes Abteilungsbezeichnung soll auf 50 Byte festgelegt werden
 - Das Feld Abteilungsbezeichnung soll die Beschränkung (Constraint) NOT NULL erhalten und als Unique-Index festgelegt werden

Resultat ohne Annotationen

```
dbo.AbteilungListe
└─ Spalten
   └─ AbteilungId (PS, int, Nicht NULL)
   └─ Abteilungsbezeichnung (nvarchar(max), NULL)
```

```
[Table("abteilung", Schema = "dbo")]
[Index(nameof(Abteilungsbezeichnung), IsUnique = true)]
public partial class Abteilung
{
    public Abteilung()
    {
        this.MitarbeiterListe = new List<Mitarbeiter>();
    }

    public int AbteilungId { get; set; }

    [Required]
    [StringLength(50)]
    //[ColumnName("Abteilung")]
    public string Abteilungsbezeichnung { get; set; }

    public virtual IList<Mitarbeiter> MitarbeiterListe { get; set; }
}
```

Resultat mit Annotationen

```
dbo.Abteilung
└─ Spalten
   └─ AbteilungId (PS, int, Nicht NULL)
   └─ Abteilungsbezeichnung (nvarchar(50), Nicht NULL)
```

Nach der ersten, nur einigermaßen gelungenen Migration: Funktion

- Wir haben einige Probleme festgestellt, die wir jetzt beseitigen wollen. Hierzu nutzen wir die Möglichkeiten der Attribut-Setzung (System.ComponentModel.DataAnnotations;)
- Jetzt am Beispiel der zusätzlichen Entitätsklasse Funktion
 - Der Tabellename soll – wie in Datenbanken üblich – in der Einzahl gesetzt werden, also Funktion
 - Die max. Länge des Feldes Funktionsbezeichnung soll auf 50 Byte festgelegt werden
 - Das Feld Funktionsbezeichnung soll die Beschränkung (Constraint) NOT NULL erhalten und unique sein

Resultat ohne Annotationen

Resultat mit Annotationen

```
[Table("Funktion", Schema = "dbo")]
[Index(nameof(Funktionsbezeichnung), IsUnique = true)]
public class Funktion
{
    public Funktion() {
        this.MitarbeiterListe = new List<Mitarbeiter>();
    }
    public int FunktionId { get; set; }

    [Required]
    [StringLength(50)]
    //[ColumnName("Funktion")]
    public string Funktionsbezeichnung { get; set; }

    public virtual IList<Mitarbeiter> MitarbeiterListe { get; set; }
}
```

dbo.FunktionListe

Spalten

- FunktionId (PS, int, Nicht NULL)
- Funktionsbezeichnung (nvarchar(max), NULL)

dbo.Funktion

Spalten

- FunktionId (PS, int, Nicht NULL)
- Funktionsbezeichnung (nvarchar(50), Nicht NULL)

Nach der ersten, nur einigermaßen gelungenen Migration: Mitarbeiter

- Wir haben einige Probleme festgestellt, die wir schrittweise beseitigen wollen. Hierzu nutzen wir die Möglichkeiten der Attribut-Setzung (`System.ComponentModel.DataAnnotations`)
- Jetzt am Beispiel der Entitätsklasse `Mitarbeiter`
 - Der Tabellename soll – wie in Datenbanken üblich – in der Einzahl gesetzt werden, also `mitarbeiter`
 - Die max. Länge verschiedener String-Felder soll der erwarteten Länge angepasst werden
 - Das Feld `Nachname` soll die Beschränkung (Constraint) `NOT NULL` erhalten (nicht `unique`!)
 - Das Feld `Extern` sollte den Default-Wert `0` haben
 - Die Felder `AbteilungId` und `FunktionId` sollten den Wert `Null` zulassen, weil u.U. neu eingestellte Mitarbeiter noch gar nicht eingeordnet wurden
 - Felder wie `Email` oder `Geschlecht` oder `Durchwahl` müssen nicht im Unicode-Zeichensatz gesetzt werden
 - Datumsfelder sollten nicht die max. mögliche Präzision (7) haben, sondern können (z.B. bei Geburtstag) ohne Zeitangabe auskommen → Datentyp `DateOnly` (ab EFC8)

Annotation in der Klasse Mitarbeiter

Resultat ohne Annotationen

dbo.MitarbeiterListe

Spalten

- MitarbeiterId (PS, int, Nicht NULL)
- Prsnr (nvarchar(max), NULL)
- Nachname (nvarchar(max), NULL)
- Vorname (nvarchar(max), NULL)
- Geburtsdatum (datetime2(7), NULL)
- Geschlecht (nvarchar(max), NULL)
- DatumEintritt (datetime2(7), NULL)
- DatumAustritt (datetime2(7), NULL)
- Extern (bit, NULL)
- Email (nvarchar(max), NULL)
- Durchwahl (nvarchar(max), NULL)
- AbteilungId (FS, int, Nicht NULL)
- FunktionId (FS, int, Nicht NULL)

```
[Table("Mitarbeiter", Schema ="dbo")]
[Index(nameof(Prsnr), IsUnique = true)]
public class Mitarbeiter
{
    public int MitarbeiterId { get; set; }

    [StringLength(4)]
    public string Prsnr { get; set; }
    [Required]
    [StringLength(50)]
    public string Nachname { get; set; }
    [StringLength(50)]
    public string? Vorname { get; set; }
    [Precision(0)]
    public DateOnly? Geburtsdatum { get; set; }
    [StringLength(1)]
    [Unicode(false)]
    public string Geschlecht { get; set; }
    [Precision(0)]
    public DateOnly? DatumEintritt { get; set; }
    [Precision(0)]
    public DateOnly? DatumAustritt { get; set; }
    [DefaultValue(0)]
    public bool? Extern { get; set; }
    [StringLength(50)]
    [Unicode(false)]
    public string? Email { get; set; }
    [StringLength(10)]
    [Unicode(false)]
    public string? Durchwahl { get; set; }

    public virtual Abteilung Abteilung { get; set; }
    public virtual Funktion Funktion { get; set; }
}
```

Resultat mit Annotationen

dbo.Mitarbeiter

Spalten

- MitarbeiterId (PS, int, Nicht NULL)
- Prsnr (nvarchar(4), NULL)
- Nachname (nvarchar(50), Nicht NULL)
- Vorname (nvarchar(50), NULL)
- Geburtsdatum (datetime2(0), NULL)
- Geschlecht (varchar(1), NULL)
- DatumEintritt (datetime2(0), NULL)
- DatumAustritt (datetime2(0), NULL)
- Extern (bit, NULL)
- Email (varchar(50), NULL)
- Durchwahl (varchar(10), NULL)
- AbteilungId (FS, int, NULL)
- FunktionId (FS, int, NULL)

Einfügung der Klasse Niederlassung

- Das Modell soll durch eine Klasse Niederlassung erweitert werden, die mit der Mitarbeiterklasse in einer 1:n – Assoziation steht
- Die Properties sind: NiederlassungId, Niederlassungsbezeichnung, Niederlassungsvorwahl

```
[Table("niederlassung", Schema = "dbo")]
[Index(nameof(Niederlassungsbezeichnung), IsUnique = true)]
public class Niederlassung
{
    public int NiederlassungId { get; set; }
    [StringLength(50)]
    [Required]
    public string Niederlassungsbezeichnung { get; set; }
    [StringLength(10)]
    [Unicode(false)]
    public string Niederlassungsvorwahl { get; set; }

    //Navigation-Property:
    public List<Mitarbeiter> MitarbeiterListe { get; set; } = new List<Mitarbeiter>();
}
```

Im Folgenden:

MIGRATION MIT TESTDATEN

Nach dem Forward Engineering – nächste Migration: Bereitstellung von Testdaten

- Testdaten können in die DbContext-Klasse eingebaut werden, um nach der Migration auch sofort testen zu können. Da bei der Migration die Datenbank-Objekte neu generiert werden, hat man dann auch gleich Daten zur Verfügung
- Die Testdaten werden mit eigenen Methoden in die DbContext-Klasse eingefügt. Die Methoden mit den Testdaten werden innerhalb der DbContext-Methode OnCreating aufgerufen:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    //evtl. diverse andere Aufrufe
    TestdatenGenerierung_Funktion(modelBuilder);
    TestdatenGenerierung_Abteilung(modelBuilder);
    TestdatenGenerierung_Niederlassung(modelBuilder);

    TestdatenGenerierung_Mitarbeiter(modelBuilder);
}
```

```
private static void TestdatenGenerierung_Funktion(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Funktion>().HasData(
        new Funktion { FunktionId = 1, Funktionsbezeichnung = "Berater/in" },
        new Funktion { FunktionId = 2, Funktionsbezeichnung = "Bereichsleiter/in" },
        new Funktion { FunktionId = 3, Funktionsbezeichnung = "Entwickler/in" },
        new Funktion { FunktionId = 4, Funktionsbezeichnung = "Freie/r Mitarbeiter/in" },
        new Funktion { FunktionId = 5, Funktionsbezeichnung = "Geschäftsführer/in" },
        new Funktion { FunktionId = 6, Funktionsbezeichnung = "Mitarbeiter/in" },
        new Funktion { FunktionId = 7, Funktionsbezeichnung = "Projektleiter/in" });
}
```

Testdatengenerierung Abteilung

- Die Abteilungsdaten

```
public static void Testdatengenerierung_Abteilung(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Abteilung>().HasData(
        new Abteilung { AbteilungId = 1, Abteilungsbezeichnung = "Geschäftsleitung/RW" },
        new Abteilung { AbteilungId = 2, Abteilungsbezeichnung = "Infrastruktur" },
        new Abteilung { AbteilungId = 3, Abteilungsbezeichnung = "Marketing" },
        new Abteilung { AbteilungId = 4, Abteilungsbezeichnung = "Produktion" },
        new Abteilung { AbteilungId = 5, Abteilungsbezeichnung = "Vertrieb" });
};
```

Testdatengenerierung Mitarbeiter

- Die Mitarbeiterdaten sind etwas komplexer ...

```
public static void TestdatenGenerierung_Mitarbeiter(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Mitarbeiter>().HasData(
        new Mitarbeiter { MitarbeiterId = 1, Prsnr = "1001", Nachname = "Paul", Vorname = "Gabriele", Geschlecht = "f",
            DatumEintritt = new DateOnly(2006, 01, 01), Extern = false, Email = "gpaul@projektdb.de", Durchwahl = "20",
            AbteilungId = 1, FunktionId = 5, Geburtsdatum = new DateOnly(1993, 08, 23) },
        new Mitarbeiter { MitarbeiterId = 2, Prsnr = "1002", Nachname = "Unterlauf", Vorname = "Karin", Geschlecht = "f",
            DatumEintritt = new DateOnly(2012, 05, 15), Extern = false, Email = "kunterlauf@projektdb.de", Durchwahl = "22",
            AbteilungId = 1, FunktionId = 6, Geburtsdatum = new DateOnly(1970, 02, 06) },
        new Mitarbeiter { MitarbeiterId = 3, Prsnr = "1003", Nachname = "Schiefner", Vorname = "Emi", Geschlecht = "f",
            DatumEintritt = new DateOnly(2010, 10, 01), Extern = false, Email = "eschiefner@projektdb.de", Durchwahl = "21",
            AbteilungId = 1, FunktionId = 6, Geburtsdatum = new DateOnly(1980, 05, 14) },
        new Mitarbeiter { MitarbeiterId = 4, Prsnr = "1004", Nachname = "Geist", Vorname = "Ottmar", Geschlecht = "m",
            DatumEintritt = new DateOnly(2010, 10, 15), Extern = false, Email = "ogeist@projektdb.de", Durchwahl = "4567",
            AbteilungId = 2, FunktionId = 6, Geburtsdatum = new DateOnly(1979, 05, 13) }

        usw.
    );
}
```

New Data Seeding Methods in Entity Framework Core 9 – Felipe Gavilán

- Mit der neuen Methode UseSeedings (UseSeedingsAsync) kann man die Datenbank bereits bei deren Aufbau mit Initialdaten versehen:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
=> optionsBuilder.UseSeeding((context, _) =>
{
    var testPerson = context.Set<Person>().FirstOrDefault(p => p.Name == "Test_Person");
    if (testPerson == null)
    {
        context.Set<Person>().Add(new Person(HierarchyId.Parse("/1/"),
            "Test_Person")); context.SaveChanges(); }
    })
```

Im Folgenden:

LINQ-ABFRAGEN

Abfragen mit LINQ to Entities mit Nutzung der migrierten Testdaten

- Nach erfolgreicher Migration kann man jetzt die DbContext-Klasse (EFC01DbContext) nutzen, um Daten per LINQ abzufragen und um Daten zu manipulieren (d.h. INSERT, UPDATE und DELETE-Befehle zu generieren) – so genannte CRUD-Operationen (Create, Read, Update, Delete)
- 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

Grundsätzliches Verfahren zur „Kontaktaufnahme“ mit der Datenbank mittels der DBContextklasse

- Um auf die Datenbank über unsere EFC-DBContextklasse (EFC01DBContext) zugreifen zu können, erzeugen wir – am besten in einer Consolen-App - in einer entsprechenden Testmethode einen using-Block, in dem ein konkretes Objekt der Klasse generiert wird:

Unser konkretes Objekt der Kontext-Klasse

Unsere Kontext-Klasse

```
using (var dbctx = new EFC01DBContext())  
{  
    //Abfragen und/oder Änderungsoperationen – CRUD  
}
```

Grundsätzlicher Aufbau einer Abfrage (CRUD)

- Eine Struktur, die den Bezug zur Datenbank offen hält, so dass nachfolgende Operationen (CRUD) durchgeführt werden

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

Quelle

LINQ-und/oder EFC-Befehle

- Eine Struktur, die nur die Daten empfängt:

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

Quelle

LINQ-und/oder EFC-Befehle

Ausführung eines Kommandos

- Eine Struktur, die auf das Tracking (Nachverfolgen) verzichtet und nur die Daten empfängt:

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


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.ToString());
```



```
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 [Mitarbeiter] AS [m]
INNER JOIN [Abteilung] 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.ToString());
```



```
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 [Mitarbeiter] AS [m]
INNER JOIN [Abteilung] AS [a] ON [m].[AbteilungId] =
[a].[AbteilungId]
WHERE [a].[Bezeichnung] = N'Produktion'
ORDER BY [m].[Nachname]
```

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

- Variante „Abfragekalkül“

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

```

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



```
SELECT (COALESCE([m].[Nachname], N'') + N', ' + COALESCE([m].[Vorname], N'') AS [Mitarbeitername],
[a].[Bezeichnung] AS [Abteilung]
FROM [Mitarbeiter] AS [m]
INNER JOIN [Abteilung] 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
    .Include(m => m.Abteilung)
    .Where(m => m.Abteilung.Abteilungsbezeichnung == "Produktion")
    .OrderBy(m => m.Nachname)
    .Select (m => new {
        Mitarbeitername = m.Nachname + ", " + m.Vorname,
        Abteilung = m.Abteilung.Abteilungsbezeichnung });

```

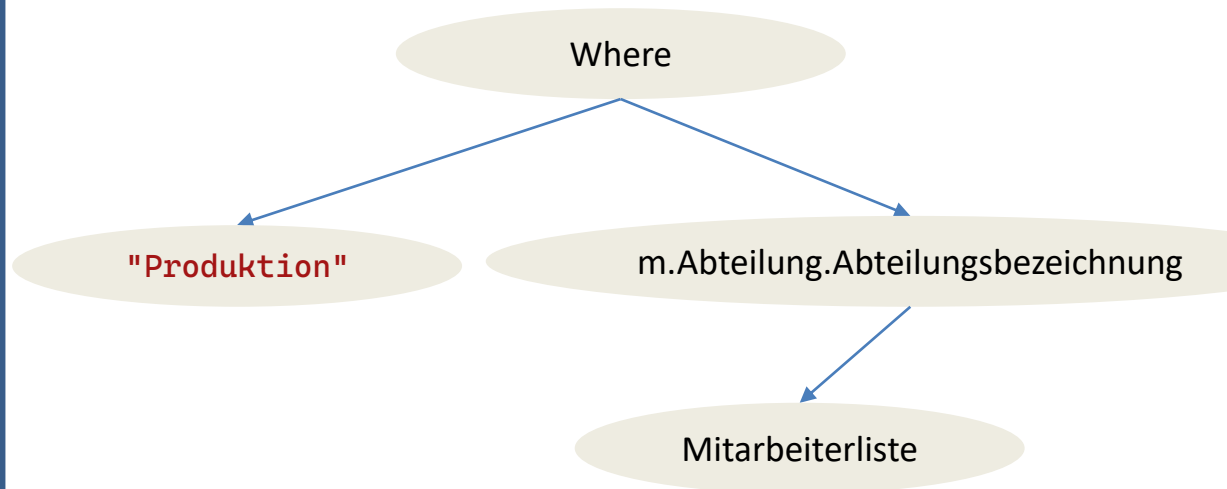
```
Display(mitarbeiterProduktion_2.ToQueryString());
```



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

Abfrage einer EF Core Datenbank mit Nutzung von LINQ

- EFCore übersetzt den LINQ-Ausdrucksbaum in eine interne Form, die für den speziellen Datenbanktreiber lesbar ist.
- Der spezielle EFCore-Datenbanktreiber konvertiert den Ausdrucksbaum in das korrekte SQL-Statement



```
var mitarbeiterProduktion = dbctx.MitarbeiterListe
    .Where(m => m.Abteilung.Abteilungsbezeichnung == "Produktion");
```

Transformierter LINQ-
Ausdrucksbaum

Datenbanktreiber

SQL-Statement, z.B.

```
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';
```

SQL Datenbank

Wichtige Befehle

LINQ Methoden	
Select()	
Where()	
Find()	Spezifische Abfrage (auf Id gerichtet)
GroupBy	
OrderBy	
ThenBy	
OrderByDescending	
Include	
ThenInclude	

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

Ladestrategien

- Bei der Formulierung von LINQ-Abfragen kommt es darauf an, möglichst nur tatsächlich benötigte Daten zu laden. Oft beachtet man nicht, dass bei einem sehr komplexen Datenmodell „alles mit allem“ zusammenhängt und deshalb u.U. überflüssige Daten geladen werden.
- Entity Framework Core bietet verschiedene Ladestrategien für verbundene Datensätze an:
 - **Standardverhalten**: Im Standard werden verbundene Datensätze im Falle von Mastertabellen nicht geladen, bei Setzen eines Kriteriums wird der entsprechende JOIN zwar erzeugt, aber die Daten der Mastertabelle werden nicht mitgenommen
 - **Eager Loading**: Mit einem Include() bzw. folgendem ThenInclude() kann der Entwickler im LINQ-Befehl bereits das Laden verbundener Datensätze erzwingen. Im resultierenden SQL-Befehl findet man entsprechende Joins, die im Datenbankmanagementsystem ausgewertet werden (Server Join).
 - **Explicite Loading**: Der Entwickler kann verbundene Datensätze mit der Ausführung der Methode Load() nachladen. Load() versteckt sich in der Klasse ReferenceEntry() bzw. CollectionEntry(), die man über die Methode Entry().Reference() bzw. Entry().Collection() der Kontextklasse erreicht.
 - **Preloading**: Zu verbindende Datensätze können einzeln per LINQ-Befehl geladen werden; Entity Framework Core setzt die materialisierten Objekte im RAM automatisch zu einem Objektbaum zusammen (Preloading / Client-Join).
 - **Lazy Loading**: Durch Aktivierung der Lazy Loading-Proxies kann der Entwickler erreichen, dass verbundene Datensätze beim Zugriff auf Navigationseigenschaften automatisch geladen werden. Hierbei entsteht aber die oben beschriebene Gefahr der Überlast.

Laden im Standardmodus

- Fall 1: Beim Laden der Master-Tabelle AbteilungsListe (AbteilungsListe – 1:n – MitarbeiterListe) werden die Detaildaten (Mitarbeiter) nicht mitgeladen:

```
var produktionsAbteilung = dbctx.AbteilungsListe
    .FirstOrDefault(a => a.Bezeichnung.Contains("Prod"));

Display(dbctx.AbteilungsListe
    .ToQueryString());
```

```
SELECT [a].[AbteilungId], [a].[Bezeichnung]
FROM [Abteilung] AS [a]
WHERE [a].[Bezeichnung] LIKE N'%Prod%'
```

```
produktionsAbteilung.ForEach(x => Display($"{x.Bezeichnung},
    Anzahl der Mitarbeiter: {x.MitarbeiterListe.Count()}"));
```

- Fall 2: Beim Laden der Detail-Tabelle MitarbeiterListe mit einer Einschränkung bzgl. der Abteilung werden aber die Daten der Master-Tabelle Abteilung nicht mitgeladen:

```
var mitarbeiterProd = dbctx.MitarbeiterListe
    .Where(m => m.Abteilung.Bezeichnung.Contains("Prod"));

Display(dbctx.MitarbeiterListe
    .ToQueryString());
```

```
SELECT m.* FROM [Mitarbeiter] AS [m]
INNER JOIN [Abteilung] AS [a]
    ON [m].[AbteilungId] = [a].[AbteilungId]
WHERE [a].[Bezeichnung] LIKE N'%Prod%'
```

Eager-Loading

- Mit dem Include-Befehl erhalten wir jetzt auch die Felder der Tabelle Abteilung:

```
SELECT [m].[MitarbeiterId], [m].[AbteilungId], [m].[DatumAustritt], [m].[DatumEintritt], [m].[Durchwahl],  
[m].[Email], [m].[Extern], [m].[FunktionId], [m].[Geburtsdatum], [m].[Geschlecht], [m].[Nachname],  
[m].[NiederlassungId], [m].[Prsnr], [m].[Vorname], [a].[AbteilungId], [a].[Abteilungsbezeichnung]  
FROM [dbo].[mitarbeiter] AS [m]  
LEFT JOIN [dbo].[abteilung] AS [a] ON [m].[AbteilungId] = [a].[AbteilungId]  
WHERE [a].[Abteilungsbezeichnung] LIKE @__abt_0_contains ESCAPE N'\'
```

```
var mitListe = dbctx.MitarbeiterListe  
    .Include(m => m.Abteilung)  
    .Where(m => m.Abteilung.Abteilungsbezeichnung.Contains(abt));  
  
Display(mitListe.ToQueryString());  
  
foreach (var mit in mitListe.ToList())  
{  
    Display($"Abteilung: {mit.Abteilung.Abteilungsbezeichnung} - {mit.Nachname}, {mit.Vorname}");  
}
```

Im Folgenden:

HINZUFÜGEN, ÄNDERN UND LÖSCHEN VON DATEN

Hinzufügen von Daten

- Wir fügen der AbteilungsListe eine neue Abteilung Rechnungswesen hinzu:

```
Abteilung a = new Abteilung()  
{ Abteilungsbezeichnung = "Rechnungswesen" };  
  
using (var dbctx = new EFC01DBContext())  
{  
    var anz = dbctx.abteilungsListe.Add(a);  
    Display(anz);  
    dbctx.SaveChanges();  
}
```

- Die Methode SaveChanges() dient der Speicherung von Änderungen in der Datenbank. Sie ist in der Basisklasse DbContext realisiert und wird von dort auf unsere eigene Kontextklasse vererbt.
- Im SQL-Profiler können die Insert-, Update- und Delete-Statements abgerufen werden:

```
exec sp_executesql N'SET IMPLICIT_TRANSACTIONS OFF;  
SET NOCOUNT ON;  
INSERT INTO[dbo].[Abteilung] ([Abteilung]) OUTPUT INSERTED.[AbteilungId] VALUES(@p0);  
,N'@p0 nvarchar(50)',@p0=N'Rechnungswesen'
```

Ändern von Daten

- In der Abteilung Geschäftsführung/RW befinden sich derzeit 3 Mitarbeiter, davon ist Gabriele Paul die Geschäftsführerin
- Wir stellen fest, welche Mitarbeiter sich in der Abteilung Geschäftsführung/RW befinden und nicht Gabriele Paul heißen. Diese erhalten eine neue AbteilungId.

```
var mitarbeiterRW_alt = dbctx.MitarbeiterListe
    .Where(m => m.Abteilung.Bezeichnung.Contains("Gesch") && m.Nachname != "Paul");


var abteilungRW = dbctx.AbteilungsListe
    .Where(a => a.Bezeichnung == "Rechnungswesen").FirstOrDefault();
```

```
foreach (var m in mitarbeiterRW_alt.ToList())
{
    Display($"... {m.Nachname} ... bisher Abteilung {m.Abteilung.Bezeichnung}");
    //Die beiden Mitarbeiter erhalten eine neue AbteilungId:
    m.Abteilung = abteilungRW;
}

db.SaveChanges();

var mitarbeiterRW_neu = db.MitarbeiterListe
    .Where(m => m.Abteilung.Bezeichnung.Contains("Rechnung"));

foreach (var m in mitarbeiterRW_neu.ToList())
{
    Display($"... {m.Nachname} ... Abteilung {m.Abteilung.Bezeichnung}");
}
```



Die zugehörigen SQL-Statements

- Wie man im SQL-Profiler sieht, werden die entsprechenden Update-Befehle im Hintergrund generiert:

```
exec sp_executesql N'SET NOCOUNT ON;  
UPDATE [dbo].[Mitarbeiter] SET [AbteilungId] = @p0  
OUTPUT 1  
WHERE [MitarbeiterId] = @p1;  
UPDATE [dbo].[Mitarbeiter] SET [AbteilungId] = @p2  
OUTPUT 1  
WHERE [MitarbeiterId] = @p3;  
,N'@p1 int,@p0 int,@p3 int,@p2 int',@p1=2,@p0=6,@p3=3,@p2=6
```

Im Folgenden:

ZUSAMMENFASSUNG, AUSBLICK

Zusammenfassung zum Projekt EFC01

- Wir richten zunächst ein Projekt mit den Teilprojekten EFC01.Lib und EFC01.ConsApp ein.
- Am Beispiel des einfachen Modells (Abteilung - 1: n - Mitarbeiter) haben wir die Grundzüge des ORM mit Hilfe von EF Core kennengelernt:
 - Wir starten mit einer einfachen Konsolen-Anwendung und einer Klassenbibliothek
 - In der Klassenbibliothek installieren wir mit Hilfe des NuGET-Paket-Managers die erforderlichen Pakete Microsoft.EntityFrameworkCore (und Microsoft.Extensions.Configuration)
 - Wir geben denConnectionString zum Aufbau der Verbindung zur Datenbank an
 - Das Modell besteht aus mehreren Entity-Klassen (Abteilung, Funktion, Niederlassung und Mitarbeiter) und einer DbContext-Klasse (EFC01DbContext)
- Nach dem Forward Engineering (Code First) entsteht auf dem SQL Server eine Datenbank EFC01 mit den Tabellen abteilung, funktion, niederlassung und mitarbeiter. Nach dem wir in den Klassen Attribute/Annotationen eingefügt haben, entstehen nach erneuter Migration akzeptable Datenstrukturen
- Nachdem wir mit Hilfe von Testdaten einige Daten mit Hilfe einer weiteren Migration in die Tabellen eingepflegt haben, können wir mit dieser Datenbank auf der Basis der Funktionalität von LINQ to Entities verschiedene CRUD-Operationen durchführen:
 - Einfache Abfragen
 - Hinzufügen, Ändern, Löschen von Daten

Ausblick

- Wie können wir Default-Werte und Constraints in die Datenstrukturen übertragen?
- Wir haben uns bisher nur mit 1:n-Beziehungen beschäftigt.
Was ist mit 1:1 – Beziehungen und n:m-Beziehungen?
- Wie steuern wir das Lösch-Verhalten?
- Wie gehen wir mit nutzerdefinierten Sichten, Datenbank-Funktionen und Stored Procedures um?
- Was bringt Reverse Engineering?
- Wie wird Vererbung abgebildet?
- Wie unterscheiden sich die Ladestrategien?

Im Folgenden

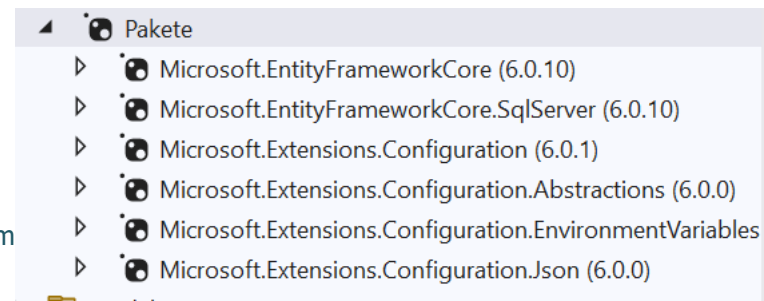
ERWEITERUNG DER ONCONFIGURING- METHODE IN DER DBCONTEXT-KLASSE – AUSLAGERUNG DES CONNECTIONSTRINGS USW.

Die Methode OnConfiguring in der Klasse EFC01DBContext

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        #warning To protect potentially sensitive information in your connection string, you should move it out
        #of source code. You can avoid scaffolding the connection string by using the Name= syntax to read
        #it from configuration - see #https://go.microsoft.com/fwlink/?linkid=2131148.
        #For more guidance on storing connection strings, see http://go.microsoft.com/fwlink/?LinkId=723263.

        optionsBuilder.UseSqlServer(" data source=localhost;initial catalog=EFC01;Integrated Security=true;
            TrustServerCertificate=Yes;Encrypt=False; ");
    }
}
```

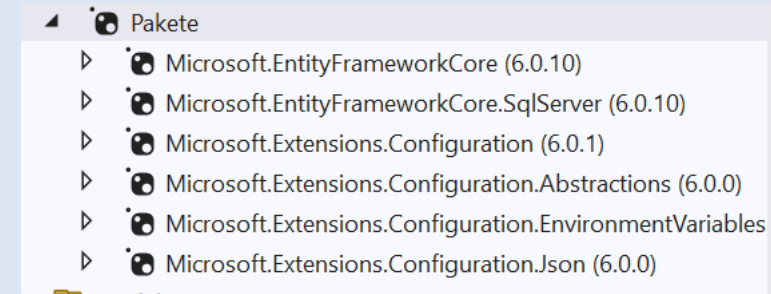
- In der Klasse wird die Methode OnConfiguring automatisch generiert (etwa beim Reverse Engineering oder im Devart Entity Developer)
- In dieser Methode wird der Bezug zum SQL Server explizit codiert. Das sollte man vermeiden.
- Durch Nutzung der **appsettings.json** wird der Connectionstring ausgelagert: [Read Configuration from appsettings.json in Entity Framework - JD Bots \(jd-bots.com\)](#). Hierzu müssen die NuGet-Pakete Microsoft.Extensions.... geladen werden



Die korrigierte Methode OnConfiguring in der Klasse EFC01DBContext

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    IConfiguration config = new ConfigurationBuilder()
        .AddJsonFile("appsettings.json")
        .AddEnvironmentVariables()
        .Build();

    var connectionString = config.GetConnectionString("DBConnectionString");
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }
}
```

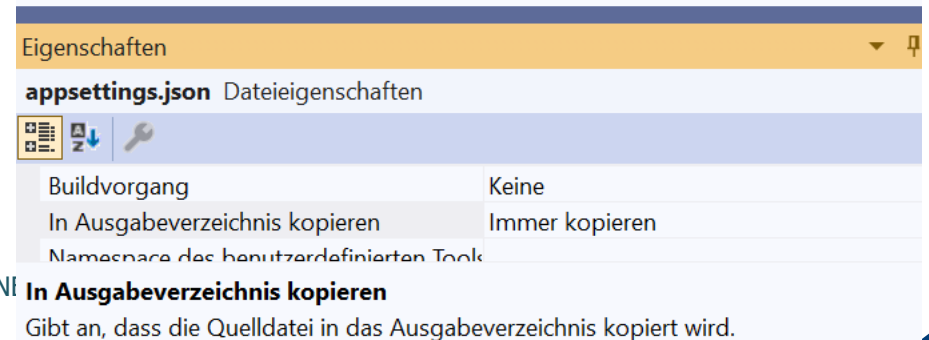
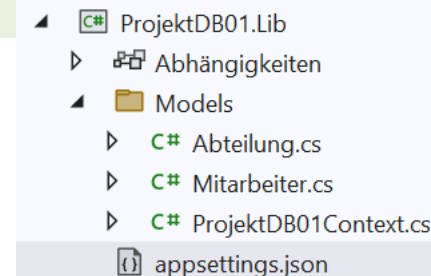


- Durch Nutzung der appsettings.json wird der Connectionstring ausgelagert: [Read Configuration from appsettings.json in Entity Framework - JD Bots \(jd-bots.com\)](#). Hierzu müssen die NuGet-Pakete Microsoft.Extensions.... geladen werden

Die appsettings.json - Datei

- Die appsettings.json-Datei ersetzt die früher in .NET-Applikationen übliche app.config-Datei. Hier kann der Connection-String zur (vorhandenen oder geplanten) SQL Server – Datenbank eingetragen werden.
- Die Datei wird am Besten im Verzeichnis gespeichert, in der nach dem Erstellen des Projekts die EFC01.Lib.dll entstanden ist, z.B.
... EFCoreWorkshop\EFC01.Lib\bin\Debug\net8.0

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DBConnectionString": "data source=localhost;initial
                           catalog=EFC01;Integrated Security=true;
                           TrustServerCertificate=Yes;Encrypt=False; "
  }
}
```



Die Klasse `EFC01DBContext` nach der Korrektur

```
public partial class EFC01DBContext: DbContext
{
    public EFC01DBContext()
    {
    }

    public EFC01DBContext(DbContextOptions< EFC01DBContext > options): base(options)
    {
    }

    public virtual DbSet<Abteilung> AbteilungListe { get; set; }
    public virtual DbSet<Funktion> FunktionListe { get; set; }
    public virtual DbSet<Niederlassung> NiederlassungListe { get; set; }
    public virtual DbSet<Mitarbeiter> MitarbeiterListe { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        IConfiguration config = new ConfigurationBuilder()
            .AddJsonFile("appsettings.json")
            .AddEnvironmentVariables()
            .Build();

        var connectionString = config.GetConnectionString("DBConnectionString");
        if (!optionsBuilder.IsConfigured)
        {
            optionsBuilder.UseSqlServer(connectionString);
        }
    }
}
```

Microsoft.Extension-Klassen

Erweiterung: Logging-Konfiguration in der DbContext-Klasse, Methode OnConfiguring

- Variante 1: Ausgabe von Logging-Daten in die Konsole:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    .....
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(connectionString);
        optionsBuilder.LogTo(Console.WriteLine, LogLevel.Information);
        //optionsBuilder.EnableSensitiveDataLogging();
    }
}
```

- Variante 2: Ausgabe in eine Log-Datei.

```
private readonly StreamWriter _logStream = new StreamWriter("efcworkshop.log", append: true);

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    ...
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder
            .UseSqlServer(connectionString)
            .LogTo(_logStream.WriteLine)
            //optionsBuilder.EnableSensitiveDataLogging();
    }
}
```