Einführung in Entity Framework Core

Vertiefung am Beispiel EFC02

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

In diesem Abschnitt werden die Modelle aus dem ersten Projekt in ein neues Projekt übertragen. Dieses besteht wieder aus einer Konsolen-Applikation und einer Klassenbibliothek (EFC02.ConsApp und EFC02.Lib).

Wir diskutieren den Zusammenhang von Konventionen, Annotationen und dem Fluent-API in der Methode OnModelCreating() in der Kontextklasse. Hierzu erweitern wir u.a. das Modell durch eine abstrakte Klasse Audit.

Mit der Nutzung der Fluid-Api können Feldeigenschaften präzisiert werden wie die Konversion von Zeichenketten-Daten zu Enumerationsdaten, die Setzung von Default-Werten, die automatische Generierung eines Default-Datums (angelegtAm).

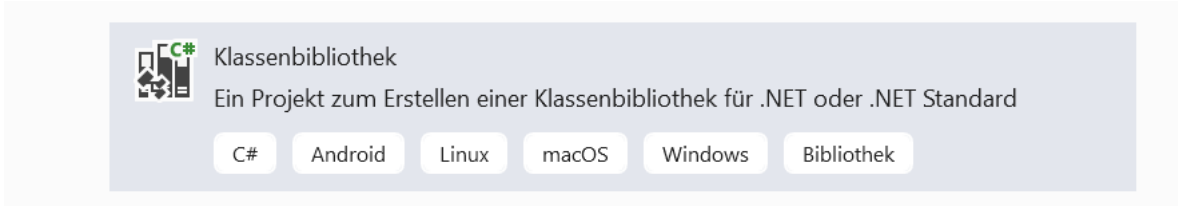
Nachdem Testdaten hinzugefügt wurden und mit einer erneuten Migration die Datenbanktabellen gefüllt wurden, lassen sich CRUD-Operationen mit den Daten durchführen. Dabei wird gezeigt, dass die Ergänzungen durch die Fluid Api saubere Lösungen ergeben.

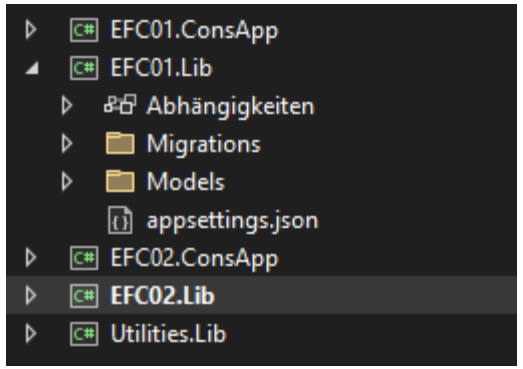
Offene Fragen

- 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 werden komplexe Properties abgebildet? …

Einrichtung des Projekts EFC02

- Wir richten zunächst in der Projektmappe eine neue Konsolen-Anwendung EFC02.ConsApp ein. Anschließend fügen wir eine Klassenbibliothek hinzu, die den Namen EFC02.Lib erhalten soll:





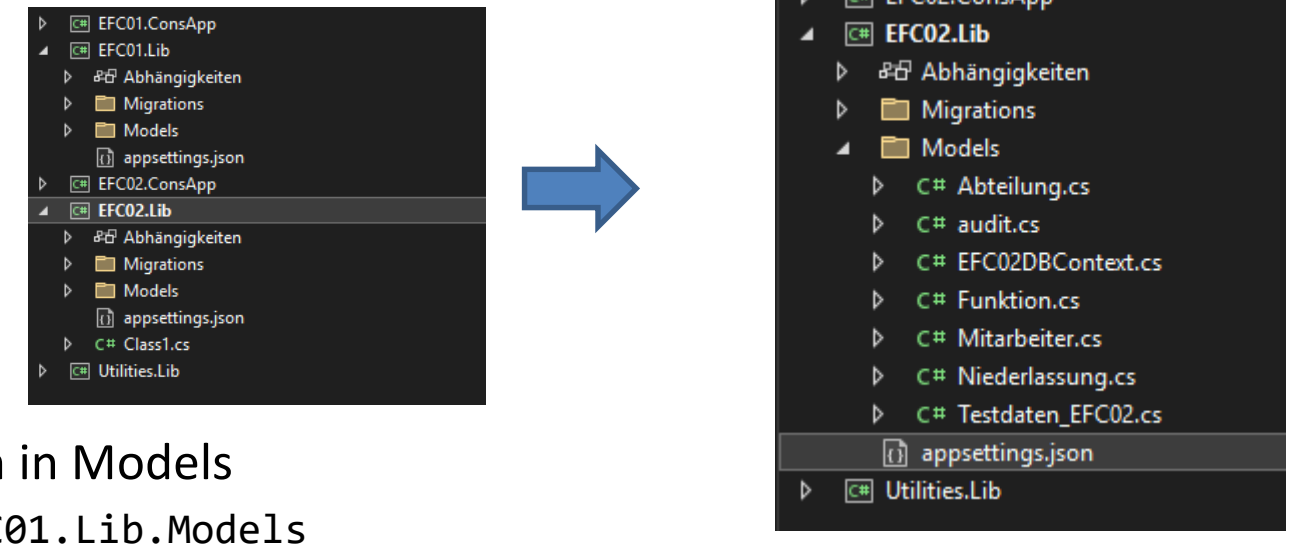
Einrichtung der neuen Klassenbibliothek

Wir wollen jetzt die Klassenbiblothek EFC02.Lib als Grundlage für weitere ORM-Aktivitäten aufbauen. Hierzu könnten wir wieder mit dem Paket-Manager sämtliche erforderliche Pakete installieren. Wir können aber auch aus der Projektdatei EFC01.Lib.csproj einfach den Inhalt in die Projektdatei EFC02.Lib.csproj per Copy und Paste übernehmen.

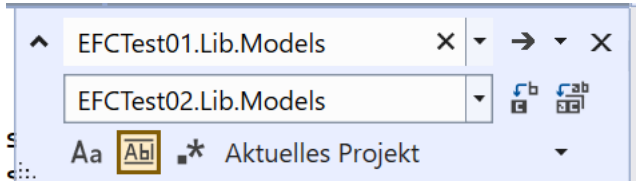
```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>disable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <Compile Remove="Models\Audit.cs" />
    <Compile Remove="Models\Records.cs" />
    <Compile Remove="Models\Testdaten_EFC01.cs" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="9.0.9" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="9.0.9">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Proxies" Version="9.0.9" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="9.0.9" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="9.0.9">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.Extensions.Configuration" Version="9.0.9" />
    <PackageReference Include="Microsoft.Extensions.Configuration.EnvironmentVariables" Version="9.0.9" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Json" Version="9.0.9" />
    <PackageReference Include="Microsoft.Extensions.Logging" Version="9.0.9" />
    <PackageReference Include="Microsoft.Extensions.Logging.Console" Version="9.0.9" />
  </ItemGroup>

</Project>
```

Einrichtung der neuen Klassenbibliothek

- Jetzt kopieren wir die Datei appsettings.json und den Ordner Models aus dem EFC01.Lib in die Klassenbibliothek EFC02.Lib

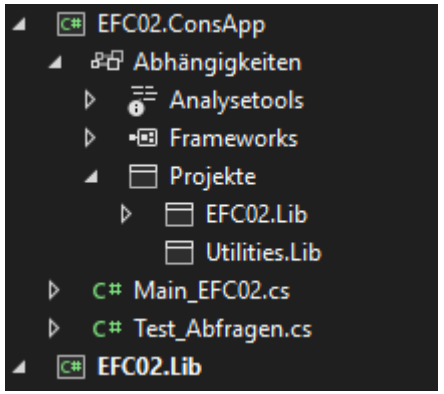


- Im nächsten Schritt müssen wir die Klassen in Models öffnen und den Namespace `namespace EFC01.Lib.Models` in `namespace EFC02.Lib.Models` sowie die Datei `EFC01DBContext.cs` in `EFC02DBContext.cs` umbenennen:



Einrichtung der Anwendung EFC02.ConsApp

- Nach der Einrichtung der Klassenbibliothek können wir uns nun der Konsolen-Anwendung zuwenden. Da wir Anwendungs- und Klassenprojekt getrennt haben, müssen wir in die Konsolen-Anwendung unter Abhängigkeiten\Projekte den Verweis auf die neue Klassenbibliothek aufnehmen:
- Jetzt können wir einen ersten Test durchführen. Greift unsere neue Anwendung auf die im ersten Abschnitt entwickelte Datenbank EFC01 zu? Da wir die Verbindungszeichenfolge in der appSettings noch nicht geändert haben, dürfte der Test funktionieren:




```
using (var dbctx = new EFC02.Lib.Models.EFC02Context())
{
    var mitProd = dbctx.MitarbeiterListe
    .Where(m => m.Abteilung.Bezeichnung == "Produktion" && m.Datum_Austritt == null);

    Display($"Mitarbeiter der Abt. Produktion:");
    foreach (var m in mitProd.ToList())
    {
          var dt = Convert.ToDateTime(m.Datum_Eintritt).ToShortDateString();
          Display($"{m.Nachname}, {m.Vorname}, Datum Eintritt: {dt}");
    }
}
```

MÖGLICHKEITEN ZUR KONFIGURIERUNG DES MODELLS

Festlegungen und Einschränkungen zur Konfiguration des Datenbankschemas

- Die Konfiguration des Datenbankschemas erfolgt bei Entity Framework Core auf drei Wegen:
  - mit **Konventionen**, Beispiel: Erkennung des Primärschlüssel auf der Basis <Klassenname>Id
  - mit **Datenannotationen** in den Entitätsklassen ([Key], [StringLength(50)] oder
  - dem **Fluent-API in der Methode OnModelCreating()** in der Kontextklasse (.HasKey(„ <Klassenname>Id“).
  - Es gelten dabei drei Grundregeln:
    - Konfiguration per Datenannotationen oder Fluent-API wiegt schwerer als die Konventionen, d.h. Konfiguration setzt Konventionen für einzelne Fälle außer Kraft. Microsoft spricht bei Entity Framework Core von "Konvention vor Konfiguration". Damit ist jedoch gemeint, dass es Ziel ist, durch Konventionen die explizite Konfiguration soweit es geht überflüssig zu machen.
    - Wenn es sich widersprechende Datenannotationen und Fluent-API-Aufrufe gibt, wiegt immer der Fluent-API-Aufruf schwerer.
    - Man kann alle Konfigurationsmöglichkeiten per Fluent-API ausdrücken. Eine Teilmenge davon ist auch per Datenannotation möglich

Konventionen in EF Core

- Einige Voreinstellungen sollte man kennen:

Objekt	Konvention	Anmerkung
Schemaname	Wenn nicht explizit angegeben, wird dbo als Schema gesetzt	
Tabellenname	Der Tabellenname wird aus der DBSet-Definition übernommen.	

Primärschlüssel
 	
Fremdschlüssel
	public class Buch
	{
	public int BuchId { get; set; } //oder nur Id
	public string Titel { get; set; }
	//Navigationsproperty
	(public int FachgebietId { get; set; })
	public Fachgebiet Fachgebiet { get; set; }
	}
		Primärschlüssel und Fremdschlüssel werden erkannt. Grundlage dafür ist die Schreibweise <klassenname>Id.
		Auf dieser Basis werden dann auch Assoziationen in referentielle Beziehungen umgesetzt

Spaltennamen	Werden übernommen aus den Property-Namen	
Datentypen	Werden in die entsprechenden SQL Server Datentypen konvertiert	Bei string folgt ohne Überschreibung nvarchar(max); Bei DateTime folgt ohne Überschreibung datetime2(7)
Constraints und Default-Werte	Default-Werte teilweise	Alle Felder (außer Primärschlüssel) werden auf NULL gesetzt.

Zur Erinnerung: Die Datentypen

SQL-Server-Datentypen		.NET-Datentypen
tinyint, smallint, int, bigint	korrespondiert mit	byte, short, integer, long
real, float, decimal(18,2)		float, double, decimal
bit		Boolean
nvarchar(max), varchar, char		String
datetime2(7), datetime2(0)		DateTime
date, time		DateOnly, TimeOnly
varbinary(max)		byte[]
hierachieId		hierarchieid

Wiederholung: Festlegungen und Einschränkungen zur Konfiguration des Datenbankschemas

- Festlegung der Namen von Schema, Tabelle oder Spalte
- Festlegung des Datentyps bei Spalten
- Größenbeschränkungen – zum Beispiel minimale und/oder maximale Länge einer Zeichenkette
- Beschränkung des Wertebereiches – Festlegung Min, Max, Range von numerischen Feldern
- Setzen von Default Werten
- Fehlermeldungen bei Verletzung von Einschränkungen

In den Modellklassen muss auf die Bibliothek

```
System.ComponentModel.DataAnnotations;
```

verwiesen werden!

Festlegung der Namen von Schema, Tabellen und/oder der Namen und weiterer Merkmale von Spalten

Objekt der Festlegung	Datenannotation	Fluent-API (Methode in der DBContext-Klasse)
Tabellenname, Schemaname	Vor einer Klasse: [Table("Tabellenname")] oder unter zusätzlicher Angabe des Schemanamens [Table("Tabellenname", Schema = "Schemaname")]. Ohne die Schemanamensangabe landet die Tabelle immer im Standardschema "dbo".	modelBuilder.Entity() .ToTable("Tabellenname"); bzw. modelBuilder.Entity() .ToTable(@"Tabellenname", schema: @"Schemaname");
Spaltenname	Vor einem Property: [Column("Spaltenname ")]	modelBuilder.Entity() .Property(b => b.PropertyName) .HasColumnName(@"Spaltenname");
Position der Spalte	[Column(Order = 5)] (Die Zählung erfolgt ab 0 aufwärts)	modelBuilder.Entity() .Property(b => b. PropertyName) .HasColumnOrder(5);
Kommentar zur Spalte	[Comment(“Das ist ein Kommentar")] public string { get; set; }	modelBuilder.Entity() .Property(b => b. PropertyName) .HasComment("Das ist ein Kommentar");
Länge von Zeichenfolgen	[StringLength(200)] oder [MaxLength(200), MinLength(5)] oder [StringLength(200, MinimumLength = 5)]	modelBuilder.Entity() .Property(b => b.PropertyName) .HasColumnName(@"Nachname").HasColumnType(@"nvarchar(50)") .IsRequired().ValueGeneratedNever().HasMaxLength(50);
Wertebereich bei Zahlen	[Range(minWert, maxWert) [Range(minWert, maxWert, ErrorMessage = „Der Wert darf nicht kleiner 0 sein!)	
Datentyp eines Feldes	[Column(TypeName = "nvarchar(200)")]	modelBuilder.Entity() .Property(b => b.PropertyName) .HasColumnType(@“Datentyp");
Unicode	[Unicode(false)] (d.h. String wird nicht in nvarchar(), sondern in varchar() überführt)	modelBuilder.Entity() .Property(b => b.PropertyName) .IsUnicode(false);
Präzision und Scale bei Dezimalzahlen	[Precision(14, 2)] public decimal Nettopreis{ get; set; }	modelBuilder.Entity() .Property(b => b.Nettopreis) .HasPrecision(14, 2);
Präzision bei Datum/Zeit-Werten	[Precision(0)] public DateTime DatumEintritt{ get; set; } (zwischen 0 und 7 – 0 bedeutet Zeitangabe ohne Millisekunden)	.Property(b => b. DatumEintritt) .HasPrecision(0);

Festlegung weiterer Merkmale von Spalten

Objekt der Festlegung	Datenannotation	Fluent-API
Obligatorisch (NOT NULL)	[Required]	modelBuilder.Entity() .Property(b => b.PropertyName) .IsRequired()
Primärschlüssel Id	Wird automatisch erkannt, wenn z.B. der Spaltenname entsprechend lautet: public int tabellennameId {get; set;} Wenn nicht: [Key]	modelBuilder.Entity() .Property(b => b.PropertyName) .HasColumnName(@"…Id") .HasColumnType(@"int") .IsRequired().ValueGeneratedOnAdd().HasPrecision(10, 0);
Primärschlüssel nicht Id-Feld, z.B. auch zusammengesetzte Schlüssel	[Key]	modelBuilder.Entity<Auto>() .HasKey(c => new { c.Land, c.Autokennzeichen });
Fremdschlüssel	[ForeignKey("Schlüsselname")]	
Vergabe von Namen für Schlüssel		modelBuilder.Entity<Auto>() .HasKey(c => new { c.Land, c.Autokennzeichen}) .HasName("PrimaryKey_LandAutokennzeichen");
Standardwert	[DefaultValue(konkreterWert)]	modelBuilder.Entity() .Property(b => b.PropertyName) .HasDefaultValue(konkreterWert);
Automatisch generierter Standardwert		modelBuilder.Entity() .Property(b => b.DatumErstellung) .HasDefaultValueSql(@"getdate()")
Berechnete Spalte (Computed Column)		modelBuilder.Entity<Mitarbeiter>() .Property(p => p.AnzeigeName) .HasComputedColumnSql("[Nachname] + ', ' + [Vorname]");

Festlegung weiterer Merkmale von Spalten: Indexe und Check-Constraints

Objekt der Festlegung	Datenannotation	Fluent-API
Index	[Index(nameof(NachName), nameof(VorName))] public class Mitarbeiter { public int MitarbeiterId { get; set; } public string NachName { get; set; } public string VorName { get; set; } }	modelBuilder.Entity<Mitarbeiter>() .HasIndex(p => new { p.VorName, p.NachName });
Unique Index	Wird vor dem Klassennamen eingetragen: [Index(nameof(Bezeichnung), IsUnique = true)] public class Funktion { public int FunktionId { get; set; } public string Bezeichnung { get; set; } }	modelBuilder.Entity<Abteilung>() .HasIndex(b => b.Bezeichnung) .IsUnique();
Name des Index	[Index(nameof(Bezeichnung), Name="Ix_Bezeichnung"]	modelBuilder.Entity<Abteilung>() .HasIndex(b => b.Bezeichnung) .HasDatabaseName("Ix_Bezeichnung");
Check		modelBuilder .Entity<Gehalt>() .ToTable(b => b.HasCheckConstraint("Ck_Gehalt", "[Gehalt] > 0"));
		modelBuilder .Entity<Geschlecht>() .ToTable(b => b.HasCheckConstraint("Ck_Geschlecht", "[Geschlecht] in ('m','w','d', '')");
Konversionen	Value Conversions - EF Core | Microsoft Learn	entity.Property(e => e.Geschlecht) .HasMaxLength(1) .IsUnicode(false) .HasConversion(v => v.ToString(), v => (EGeschlecht)Enum.Parse(typeof(EGeschlecht), v)) .HasDefaultValueSql("'");

Vgl. GitHub - efcore/EFCore.CheckConstraints: An Entity Framework Core plugin to automatically add check constraints in various situations

Generierung einer GUID

- Annotation

```
[DatabaseGenerated(DatabaseGeneratedOption.Identity)]
public string Key { get; set; }
```

- Fluent Api

```
entity.Property(e => e.Key)
          .HasMaxLength(20)
          .HasDefaultValueSql("('A')")
          .ValueGeneratedOnAdd();
```

Festlegung der Namen von Schema, Tabellen und/oder der Namen und weiterer Merkmale von Spalten in den einzelnen Modell-Klassen

- Beispiel: Klasse Niederlassung.
- Links: Datenannotation, rechts: Fluent AP in der DBContext-Methode OnModelCreating

```
[Table("niederlassung")]
[Index(nameof(Niederlassungsbezeichnung), IsUnique = true)]
public partial  class Niederlassung
{
        public Niederlassung()
        {
            this.MitarbeiterListe = new List<Mitarbeiter>();
        }
        [Key]
        public virtual int NiederlassungId { get; set;}
        [StringLength(50)]
        [Required]
        public virtual string Niederlassungsbezeichnung { get; set;}
        [StringLength(20)]
        [Unicode(false)]
        public virtual string Niederlassungsvorwahl { get; set;}
        public virtual IList<Mitarbeiter> MitarbeiterListe { get;}
}
```

Manuell

```
private void NiederlassungMapping(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Niederlassung>(entity =>
    {
        entity.ToTable("Niederlassung", "dbo");

        entity.Property(x => x.NiederlassungId)
            .HasColumnName(@"NiederlassungId").HasColumnType(@"int")
            .IsRequired().ValueGeneratedOnAdd().HasPrecision(10, 0);
        entity.Property(x => x.Niederlassungsbezeichnung)
            .HasColumnName(@"Niederlassungsbezeichnung")
            .HasColumnType(@"nvarchar(50)").IsRequired()
            .ValueGeneratedNever().HasMaxLength(50);
        entity.Property(x => x.Niederlassungsvorwahl)
            .HasColumnName(@"Niederlassungsvorwahl")
            .HasColumnType(@"varchar(20)")
            .IsUnicode(false)
            .ValueGeneratedNever().HasMaxLength(20);
        entity.HasKey(@"NiederlassungId");
        entity.HasIndex(@"Niederlassungsbezeichnung").IsUnique(true);
    });
}
```

Migration: Forward Engineering

niederlassung

	Spaltenname	Datentyp	NULL-Werte zulassen
🔑	NiederlassungId	int	☐
	Niederlassungsbezeichnung	nvarchar(50)	☐
	Niederlassungsvorwahl	varchar(20)	☑
			☐

Festlegung der Namen von Schema, Tabellen und/oder der Namen und weiterer Merkmale von Spalten in den einzelnen Modell-Klassen

- Beispiel: Klasse Niederlassung.
- Links: Datenannotation, rechts: Fluent AP in der DBContext-Methode OnModelCreating

```
[Table("niederlassung")]
[Index(nameof(Niederlassungsbezeichnung), IsUnique = true)]
public partial  class Niederlassung
{
        public Niederlassung()
        {
            this.MitarbeiterListe = new List<Mitarbeiter>();
        }
        [Key]
        public virtual int NiederlassungId { get; set;}
        [StringLength(50)]
        [Required]
        public virtual string Niederlassungsbezeichnung { get; set;}
        [StringLength(10)]
        [Unicode(false)]
        public virtual string Niederlassungsvorwahl { get; set;}
        public virtual IList<Mitarbeiter> MitarbeiterListe { get;}
}
```

```
private void NiederlassungMapping(ModelBuilder modelBuilder)
{
        modelBuilder.Entity<Niederlassung>().ToTable(@"niederlassung", @"dbo");
        modelBuilder.Entity<Niederlassung>().Property(x => x.NiederlassungId).
                HasColumnName(@"NiederlassungId").HasColumnType(@"int")
                .IsRequired().ValueGeneratedOnAdd().HasPrecision(10, 0);
        modelBuilder.Entity<Niederlassung>().Property(x => x.Niederlassungsbezeichnung)
                .HasColumnName(@"Niederlassungsbezeichnung")
                .HasColumnType(@"nvarchar(50)").IsRequired()
                .ValueGeneratedNever().HasMaxLength(50); …
        modelBuilder.Entity<Niederlassung>().HasKey(@"NiederlassungId");
        modelBuilder.Entity<Niederlassung>()
                .HasIndex(@"Niederlassungsbezeichnung").IsUnique(true);
}
```

Migration: Forward Engineering

niederlassung

	Spaltenname	Datentyp	NULL-Werte zulassen
🔑	NiederlassungId	int	☐
	Niederlassungsbezeichnung	nvarchar(50)	☐
	Niederlassungsvorwahl	varchar(20)	☑
			☐

per
Reverse Engineering

Modell-Klasse Mitarbeiter

```
[Table("mitarbeiter", Schema = "dbo")]
[Index(nameof(prsnr), IsUnique = true)]
public class Mitarbeiter
{
    public int MitarbeiterId { get; set; }
    [StringLength(4)]
    [Required]
    public string? prsnr { get; set; }
    [StringLength(50)]
    public string? Vorname { get; set; }
    [StringLength(50)]
    [Required]
    public string Nachname { get; set; }
    [StringLength(1)]
    [Unicode(false)]
    public string Geschlecht { get; set; }
    [Precision(0)]
    public DateOnly Geburtsdatum { get; set; }
    [Precision(0)]
    public DateOnly? DatumEintritt { get; set; }
    [Precision(0)]
    public DateOnly? DatumAustritt { get; set; }
    public Boolean Extern { get; set; }
    [StringLength(50)]
    [Unicode(false)]
    public string Email { get; set; }
    [StringLength(20)]
    [Unicode(false)]
    public string Durchwahl { get; set; }
    public int FunktionId { get; set; }
    public int AbteilungId { get; set; }
    public int NiederlassungId { get; set; }
    public Funktion Funktion { get; set; }
    public Abteilung Abteilung { get; set; }
    public Niederlassung Niederlassung { get; set;}
}
```

mitarbeiter

	Spaltenname	Datentyp	NULL-Werte zulassen
🔑	MitarbeiterId	int	☐
	prsnr	nvarchar(4)	☐
	Vorname	nvarchar(50)	☑
	Nachname	nvarchar(50)	☐
	Geschlecht	varchar(1)	☑
	Geburtsdatum	datetime2(0)	☐
	DatumEintritt	datetime2(0)	☑
	DatumAustritt	datetime2(0)	☑
	Extern	bit	☐
	Email	varchar(50)	☑
	Durchwahl	varchar(10)	☑
	FunktionId	int	☐
	AbteilungId	int	☐
	NiederlassungId	int	☐

DIE UMSETZUNG DER ASSOZIATIONEN (1:N, M:N, 1:1)

Die 1:n-Beziehung zwischen Abteilung, Funktion bzw. Niederlassung und Mitarbeiter Vermeidung von kaskadierendem Löschen

- Abteilung, Funktion, Niederlassung sind so genannte Nachschlage- oder Lookup-Tabellen mit deren Hilfe gemeinsame Merkmale der Mitarbeiter festgelegt werden können
- Für die Beziehung zwischen Mastertabellen (Abteilung, Funktion, Niederlassung) und Detail-Tabelle (Mitarbeiter) (auch Prinzipal/Dependent Entities oder Parent/Child) gilt in diesem Fall: Wenn eine Abteilung/Funktion/Niederlassung gelöscht werden sollte, dürfen natürlich die zugehörigen Mitarbeiter nicht gelöscht werden! Sollte eine Abteilung/Funktion/Niederlassung gelöscht werden, muss der entsprechende Fremdschlüssel in den betroffenen Mitarbeitersätzen auf NULL gesetzt werden oder es muss eine Fehlermeldung auftreten.
- Per Konvention kann dies dem EFCore mitgeteilt werden, indem der Datentyp int des Fremdschlüssels AbteilungId, FunktionId bzw. NiederlassungId auf int? gesetzt wird

```
public int? AbteilungId { get; set; }
public int? FunktionId { get; set; }
public int? NiederlassungId { get; set; }
```

Die Variante Fluent API

- Eine andere Variante ist die Nutzung der Fluent API in der DBContext-Methode OnModelCreating(…) zur expliziten Festlegung des Löschverhaltens:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Mitarbeiter>(entity =>
    {
       entity.ToTable("Mitarbeiter","dbo");
       entity.HasOne(d => d.Abteilung)
                    .WithMany(p => p.MitarbeiterListe)
                    .HasForeignKey(d => d.AbteilungId)
                    //.OnDelete(DeleteBehavior.SetNull);

       entity.HasOne(d => d.Funktion)
                    .WithMany(p => p.MitarbeiterListe)
                    .HasForeignKey(d => d.FunktionId)
                    //.OnDelete(DeleteBehavior.SetNull);

       entity.HasOne(d => d.Niederlassung)
                    .WithMany(p => p. MitarbeiterListe)
                    .HasForeignKey(d => d.NiederlassungId)
                    //.OnDelete(DeleteBehavior.SetNull);

    });
}
```

ERWEITERUNG DES MODELLS: DEFAULT-WERTE, ENUMERATIONEN

Erweiterung des Modells:
Eine abstrakte Klasse zum Auditieren der Erzeugung und Änderung von Datensätzen

- Tabellen in einer produktiven Datenbank sollten Felder enthalten, in denen das aktuelle Datum und der aktuelle Nutzer der Erzeugung bzw. Änderung eines Datensatzes gespeichert werden. Das erlaubt in bestimmten Fällen ein Nachverfolgen von Prozessen
- In einer abstrakten Klasse werden die entsprechenden Properties gesammelt
- Da es sich um eine abstrakte Klasse handelt, wird diese nicht in die Datenbank migriert. Das gilt auch für Interfaces
- Alle Klassen des Modells, die in Tabellen überführt werden, können von dieser abstrakten Klasse erben, so dass keine Redundanz entsteht und die Einheitlichkeit dieser Audit-Felder gewahrt bleibt

```
public abstract class Audit
{
    public DateTime angelegtAm { get; set; } =  DateTime.Now;
    public DateTime geaendertAm { get; set; }
    [StringLength(50)]
    public String angelegtVon { get; set; } = Environment.UserName;

    [StringLength(50)]
    [DefaultValue("")]
    public String geaendertVon { get; set; }
    //[DefaultValue(true)] - Funktioniert nicht!!!
    public bool istAktiv { get; set; } = true;
    //[DefaultValue(false)]
    public bool istGeloescht { get; set; } = false;
}
```

Vererbung der Properties der abstrakten Klasse Audit

```
[Table("Niederlassung")]
[Index(nameof(Niederlassungsbezeichnung), IsUnique = true)]
public partial  class Niederlassung : Audit
{
        public Niederlassung()
        {
            this.MitarbeiterListe = new List<Mitarbeiter>();
        }
        [Key]
        public virtual int NiederlassungId { get; set; }

        [StringLength(50)]
        [Required]
        public virtual string NIederlassungsbezeichnung { get; set; }

        [StringLength(10)]
        [UniCode(false)]
        public virtual string Niederlassungsvorwahl { get; set; }

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

Nach der Migration

	Spaltenname	Datentyp	NULL-Werte zulassen
🔑	NiederlassungId	int	☐
	Bezeichnung	nvarchar(50)	☐
	Telefon	nvarchar(10)	☑
	angelegtAm	datetime2(7)	☐
	angelegtVon	nvarchar(10)	☑
	geaendertAm	datetime2(7)	☐
	geaendertVon	nvarchar(10)	☑
	istAktiv	bit	☐
	istGeloescht	bit	☐

Übertragung von Default-Werten und Constraints

- Beispiel: Entitätsklasse Niederlassung jetzt ohne Annotationen

```
public partial class Niederlassung:Audit
{
  public Niederlassung()
  {
      MitarbeiterListe = new List<Mitarbeiter>();
  }

  public int NiederlassungId { get; set; }
  public string Niederlassungsbezeichnung { get; set; }
  public string Niederlassungsvorwahl { get; set; }


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

Die Audit-Properties werden bei der Migration (Forward Engineering) in die Tabellen übernommen. Aber leider kann man so z.B. für das Feld angelegtAm keine automatische Belegung mit dem Datum generieren.

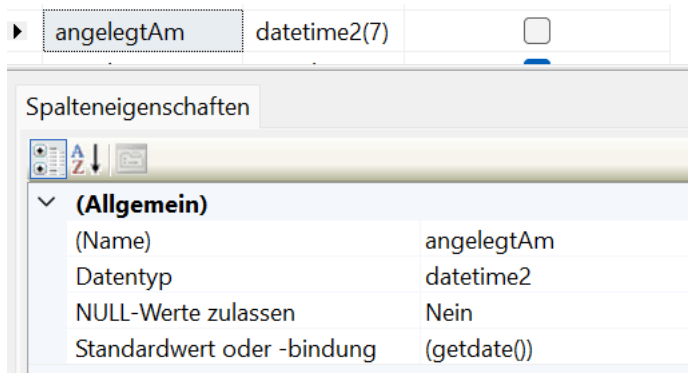
Ein Problem: Keine Annotation für das Feld angelegtAm

- Man erwartet, dass das Audit-Feld angelegtAm automatisch den Datum/Zeit-Wert des Einfügens eines neuen Datensatzes einträgt. In einer Datenbank-Tabelle kann man dies unter den Spalteneigenschaften dieses Feldes mit dem Befehl getdate() festlegen.
- In der Audit-Klasse kann man das leider nicht allgemeingültig auf alle generierten Tabellen durch Vererbung übertragen. Man hat zwei Möglichkeiten zu einem Work-Around:
  - Man überträgt beim SaveChanges-Befehl neben den eigentlichen Objektdaten auch den aktuellen Datum/Zeit-Wert, löst das Problem also im Bereich der Anwendungsprogrammierung (siehe Gorman, S.210)
  - Man trägt in der Klasse DBContext in der Methode OnModelCreating manuell für jede Entity ein:

```
entity.Property(e => e.AngelegtAm)
  .HasColumnName("angelegtAm")
  .HasDefaultValueSql("(getdate())");
```

  - Man kann aber auch in der Datenbank ein Skript schreiben und ausführen, das diese Bindung für jede Tabelle der Datenbank realisiert und danach ein Reverse Engineering durchführen.

```
ALTER TABLE <table> ADD  CONSTRAINT [DF_<table>_GetDate]  DEFAULT
(getdate()) FOR [angelegtAm]
```



Ein Beispiel: Die Mapping-Methode für die Entitätsklasse Niederlassung in der DBContext-Klasse

- Diese Methode kann z.B. mit dem Tool Devart Entity Developer direkt aus der Datenbank abgeleitet werden. Sie wird innerhalb der Methode OnModelCreating aufgerufen

```
private void NiederlassungMapping(ModelBuilder modelBuilder)
{
   modelBuilder.Entity<Niederlassung>().ToTable(@"Niederlassung", @"dbo");
   modelBuilder.Entity<Niederlassung>().Property(x => x.NiederlassungId).HasColumnName(@"NiederlassungId")
          .HasColumnType(@"int").IsRequired().ValueGeneratedOnAdd().HasPrecision(10, 0);
   modelBuilder.Entity<Niederlassung>().Property(x => x.Niederlassungsbezeichnung).HasColumnName(@"Bezeichnung")
          .HasColumnType(@"nvarchar(50)").IsRequired().ValueGeneratedNever().HasMaxLength(50);
   modelBuilder.Entity<Niederlassung>().Property(x => x.Telefon).HasColumnName(@"Telefon")
          .HasColumnType(@"varchar(20)").ValueGeneratedNever().HasMaxLength(20);

   modelBuilder.Entity<Niederlassung>().Property(x => x.AngelegtAm).HasColumnName(@"angelegtAm")
          .HasColumnType(@"datetime2").IsRequired().ValueGeneratedOnAdd().HasDefaultValueSql(@"getdate()");
   modelBuilder.Entity<Niederlassung>().Property(x => x.GeaendertAm).HasColumnName(@"geaendertAm")
          .HasColumnType(@"datetime2").IsRequired().ValueGeneratedNever();
   modelBuilder.Entity<Niederlassung>().Property(x => x.AngelegtVon).HasColumnName(@"angelegtVon")
          .HasColumnType(@"nvarchar(10)").ValueGeneratedNever().HasMaxLength(10);
   modelBuilder.Entity<Niederlassung>().Property(x => x.GeaendertVon).HasColumnName(@"geaendertVon")
          .HasColumnType(@"nvarchar(10)").ValueGeneratedNever().HasMaxLength(10);
   modelBuilder.Entity<Niederlassung>().Property(x => x.IstAktiv).HasColumnName(@"istAktiv")
          .HasColumnType(@"bit").ValueGeneratedOnAdd().HasDefaultValueSql(@"1");
   modelBuilder.Entity<Niederlassung>().Property(x => x.IstGeloescht).HasColumnName(@"istGeloescht")
          .HasColumnType(@"bit").ValueGeneratedOnAdd().HasDefaultValueSql(@"0");

   modelBuilder.Entity<Niederlassung>().HasKey(@"NiederlassungId");
}
```

Die Übertragung von Check-Contraints am Beispiel der Enumeration EGeschlecht

- In Klassenmodellen werden häufig Enumerations benutzt, um problemlos auf diskrete, festgelegte Werte zugreifen zu können
- Beispiel: Geschlecht mit den Ausprägungen m, w, d

```
public enum EGeschlecht:int
{
        m,w,d,x
}
//Nutzung in der Klasse Mitarbeiter:
public virtual EGeschlecht? Geschlecht { get; set; }
```

- In der Datenbanktabelle würde die entsprechenden Anweisung zur Festlegung des Wertebereichs im Feld Geschlecht lauten:

```
ALTER TABLE [dbo].[Mitarbeiter]  WITH CHECK ADD  CONSTRAINT [CK_Mitarbeiter_Geschlecht_Enum]
CHECK  (([Geschlecht]='d' OR [Geschlecht]= 'w' OR [Geschlecht]='m' OR [Geschlecht]=OR
[Geschlecht]='x'));

ALTER TABLE [dbo].[Mitarbeiter] CHECK CONSTRAINT [CK_Mitarbeiter_Geschlecht_Enum];
```

- Die Umsetzung im C#-Projekt erfolgt wieder mittels Fluid Api durch Konversion (vgl. Value Conversions - EF Core | Microsoft Learn).

```
modelBuilder.Entity<Mitarbeiter>(entity =>
{
   entity.ToTable("Mitarbeiter","dbo");

….
```

```
entitiy.Property(e => e.Geschlecht)
        .HasColumnName(@"geschlecht")
        .HasMaxLength(1)
        .HasColumnType(@"varchar(1)")
        .HasDefaultValueSql(@"NULL")
        .IsUnicode(false)
        .IsRequired(false)
        .HasConversion(v => v.ToString(), v =>
(EGeschlecht)Enum.Parse(typeof(EGeschlecht), v)); }
```

Die Übertragung von Check-Contraints am Beispiel einer String-Property

- In Klassenmodellen werden Einschränkungen häufig benutzt, um problemlos auf diskrete, festgelegte Werte zugreifen zu können
- Beispiel: Geschlecht mit den Ausprägungen

```
public class Mitarbeiter
{
    [AllowedValues("w", "m", "d" )]
    public string Geschlecht { get; set;}

}
```

- Die Umsetzung im C#-Projekt erfolgt wieder mittels Fluid Api durch Konversion (vgl. Value Conversions - EF Core | Microsoft Learn).

```
modelBuilder.Entity<Mitarbeiter>(entity =>
{
    entity.ToTable("Mitarbeiter","dbo")
            .ToTable(b =>
            b.HasCheckConstraint("CK_Geschlecht",
            "[Geschlecht ] IN ('m','w','d']"));
….
```

```
entitiy.Property(e => e.Geschlecht)
        .HasColumnName(@"geschlecht")
        .HasMaxLength(1)
        .HasColumnType(@"varchar(1)")
        .HasDefaultValueSql(@"NULL")
        .IsUnicode(false)
        .IsRequired(false)
```

Nach der Migration mit Testdaten: Test mit Abfragen

- Wir prüfen:
  - Funktioniert die Konversion der Werte des Feldes Geschlecht aus der Mitarbeiter-Tabelle in eine Enumeration EGeschlecht auf der Klassen-Seite?
  - Funktioniert die automatische Erstellung des Audit-Datums: angelegt am?
  - Sind die Felder istAktiv und istGeloescht richtig belegt?

```
using (var dbctx = new EFC02DBContext())
{
          Display("Erster Test: Mitarbeiterliste, Geschlecht-Enum, angelegtAm, istAktiv, istGeloescht");
          var mitProd = dbctx.MitarbeiterListe
                    .Include(m => m.Niederlassung)
                    .Include(m=> m.Funktion)
                    .Where(m => m.Abteilung.Abteilungsbezeichnung.Contains(abteilung) && m.DatumAustritt == null);

          Display($"Mitarbeiter der Abt. {abteilung}:");
          foreach (var m in mitProd.ToList())
          {
                    var dt = Convert.ToDateTime(m.Niederlassung.angelegtAm).ToShortDateString();
                     Display($"{m.Nachname}, {m.Vorname},{m.Geschlecht},
                    +$"Niederlassung: {m.Niederlassung.Niederlassungsbezeichnung},
                     +$" angelegt am: {dt},
                     +$" Aktiv: {m.Niederlassung.istAktiv},
                     +$" Gelöscht: {m.Niederlassung.istGeloescht} ");
          }
}
```