

# ***Entity Framework Core***

## ***Code First***

1	EINLEITUNG	2
1.1	Voraussetzung	2
1.2	Vorgangsweise	2
2	VORBEREITUNG	4
2.1	Pakete	4
2.1.1	Installationsvarianten	4
2.1.2	Benötigte Pakete	4
3	TABELLENSTRUKTUR ERZEUGEN	6
3.1	Entity-Klassen	6
3.2	DbContext	6
3.2.1	Fallback mit OnConfiguring	6
3.2.2	ConnectionString	7
3.2.3	DbContext Pooling	7
3.3	Diagramm	7
4	WPF CORE	8
4.1	Datenbank erzeugen	8
4.2	Mögliche Fehler	9
4.2.1	SQL Server Object Explorer	9
4.2.2	Database name	9
4.3	appsettings.json (ohne Dependency Injection)	9
4.4	DataDirectory	10
4.5	Sqlite	10
4.5.1	Paket installieren	10
4.5.2	ConnectionString	10
4.5.3	Service registrieren	10
5	SEEDING	11
5.1	HasData	11
5.2	OnModelCreating + ExtensionMethode	11
5.2.1	Erzeugung erzwingen	11
5.2.2	CSV	11
5.2.3	Foreign Keys / Navigation Properties	12
6	CONVENTION OVER CONFIGURATION	13
6.1	Tabellennamen/Spaltennamen/Spaltentypen	13
6.2	Primary Key	13
6.3	NULL - NOT NULL	14
6.4	NotMapped	15
6.5	String-columns	15
6.6	Index	15

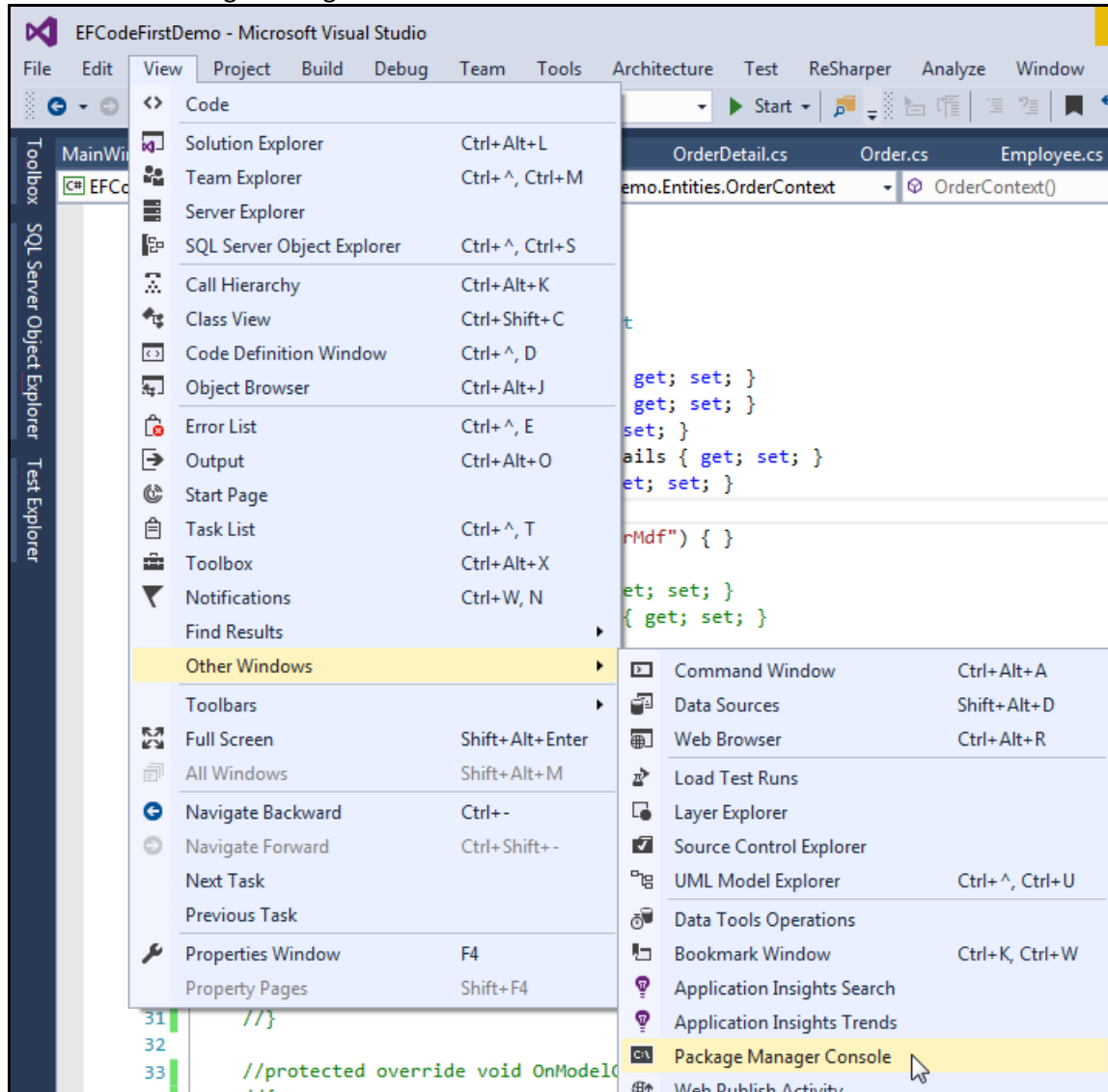
# 1 Einleitung

Greift man per Programmiercode auf eine Datenbank zu, gibt es immer 3 Ausgangsszenarien der Datenbank:

1. **Database First:** Datenbank ist vorhanden → Zugriffsklassen und Diagramm generieren lassen
2. **Code First:** Zugriffsklassen erstellen → Datenbank und Diagramm daraus erzeugen
3. **Model First:** Diagramm ist vorhanden → Zugriffsklassen und Datenbank generieren

Dieses Dokument beschäftigt sich jetzt mit der Variante „Code First“.

Für Migrationen in Visual Studio braucht man die **Package Manager Console**. Diese durch View → Other Windows → Package Manager Console einblenden:



Auto-Vervollständigung funktioniert in dieser Konsole mit der **<Tab>**-Taste.

## 1.1 Voraussetzung

Man braucht mindestens .Net Core Version >=3 sowie dotnet-ef in einer Version >=3. Idealerweise hat man aber die möglichst aktuelle Versionen installiert:

```
C:\>dotnet --version
6.0.200

C:\>dotnet-ef --version
Entity Framework Core .NET Command-line Tools
6.0.0
```

## 1.2 Vorgangsweise

Folgende Schritte müssen durchgeführt werden; sie werden in den weiteren Kapiteln genau erklärt:

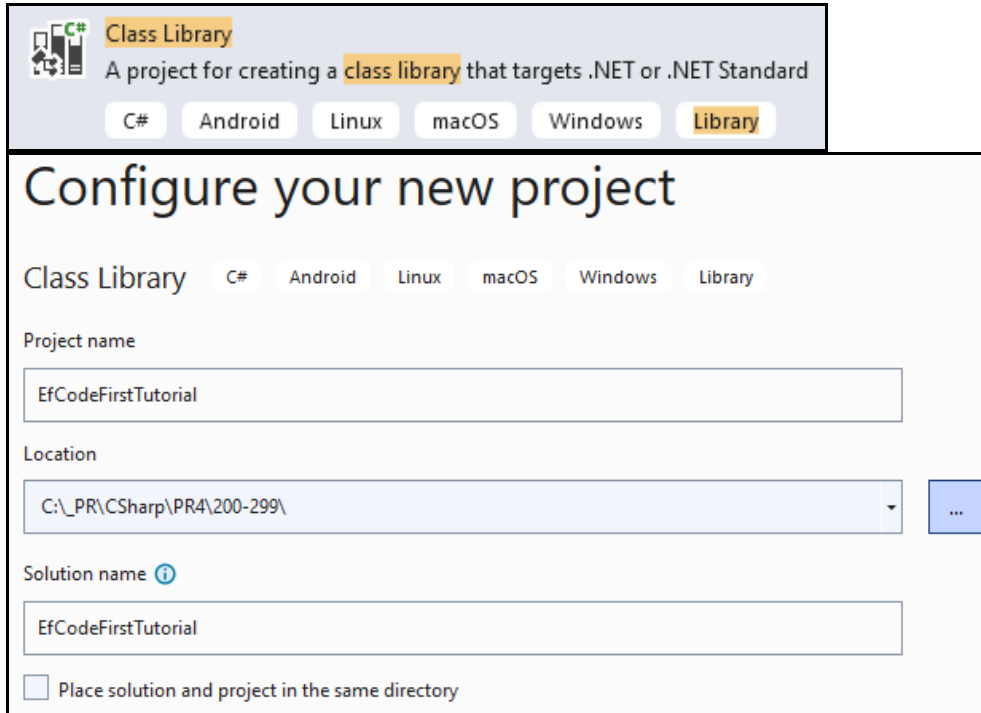
1. Pakete wie EntityFrameworkCore dem Projekt hinzufügen

2. Tabellen-**Klassen** erzeugen
3. **DbContext** erzeugen und bearbeiten
4. **ConnectionString** definieren
5. Stammdaten definieren
6. Datenbank erstellen (manuell oder per Code)

Mit Migrations kann man auf Änderung in der Datenbankstruktur reagieren – das wird in diesem Dokument nicht behandelt.

## 2 Vorbereitung

Für die Datenbank-Library wir ein Projekt vom Typ „Class Library (.NET Core)“ erstellt.



**Class Library**  
A project for creating a **class library** that targets .NET or .NET Standard

C# Android Linux macOS Windows **Library**

### Configure your new project

Class Library C# Android Linux macOS Windows Library

Project name  
EfCodeFirstTutorial

Location  
C:\\_PR\CSharp\PR4\200-299\

Solution name ⓘ  
EfCodeFirstTutorial

☐ Place solution and project in the same directory

### 2.1 Pakete

Bei einem .Net Core-Projekt muss das Paket EntityFrameworkCore und einige weitere installiert werden.

#### 2.1.1 Installationsvarianten

Dabei hat man vier Möglichkeiten (das hat jetzt aber nichts mit EFCore zu tun, sondern ist Visual Studio bei .Net Core Projekten allgemein so):

- mit NuGet
- Direkt ins .csproj File schreiben (öffnen über Kontextmenü des Projekts)
- Package Manager Console: **Install-Package Microsoft.EntityFrameworkCore** (siehe auch <https://docs.microsoft.com/en-us/nuget/consume-packages/install-use-packages-powershell>)

```
PM> Install-Package Microsoft.EntityFrameworkCore
Restoring packages for C:\Users\Besitzer\source\repos
GET https://api.nuget.org/v3-flatcontainer/microsoft
OK https://api.nuget.org/v3-flatcontainer/microsoft
GET https://api.nuget.org/v3-flatcontainer/microsoft
```

- von der Commandline (vor allem interessant, wenn man nicht mit Visual Studio entwickelt), die Angabe der Version ist dabei optional:  
**dotnet add package Microsoft.EntityFrameworkCore**  
mit Version: **dotnet add package Microsoft.EntityFrameworkCore -v 5.0.2**

#### 2.1.2 Benötigte Pakete

Folgende Pakete werden benötigt - aktuelle Version (Stand 2022-02-28) ist 6.0.2, diese wird auch verwendet, wenn man keine Version angibt.

- **Microsoft.EntityFrameworkCore**
- **Microsoft.EntityFrameworkCore.Design**
- **Microsoft.EntityFrameworkCore.SqlServer**
- **Microsoft.EntityFrameworkCore.Tools**

Welche Variante auch immer man anwendet, sollte das Project-File am Ende so aussehen:

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

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="6.0.2" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.2">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="6.0.2" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="6.0.2" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.2">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
  </ItemGroup>

</Project>
```

Eventuell kann man für die Erstversion den Wert für **<Nullable>** auf **disable** stellen.

## 3 Tabellenstruktur erzeugen

Mit diesen Vorbereitungen kann man jetzt die Tabellenstruktur erzeugen. Die Regeln sind ziemlich einfach:

- pro Tabelle erstellt man eine eigene Klasse
- pro Tabellenspalte notiert man in der entsprechenden Klasse eine Property
- Foreign Keys entstehen automatisch durch Zuweisen einer Property auf eine andere Tabellenklasse

### 3.1 Entity-Klassen

Man erzeugt wie gewohnt ein POCO (Plain Old C# Object):

```
public class School
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

```
public class Person
{
    public int Id { get; set; }
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public int Age { get; set; }
    public School School { get; set; }
}
```

Die Klassennamen sind im Singular anzugeben, also School und nicht Schools.

**Achtung:** Die Column für die ID muss **Id** heißen oder **PersonId** (also Klassenname+Id), weil sie sonst nicht als Primary Key erkannt wird!

### 3.2 DbContext

Die DbContext-Klasse ist die eigentliche Verbindung der C#-Klassen mit der Datenbank. Diese Klasse muss folgendermaßen aufgebaut werden:

- ableiten von DbContext
- pro Tabelle eine Property vom Typ DbSet<>: dabei ist der Name der Property üblicherweise der Plural des generischen Typs. Diese Namen werden die Tabellennamen.
- zwei Konstruktoren
  - Defaultkonstruktor
  - Konstruktor mit DbContextOptions<T>, wobei T der Typ der aktuellen von DbContext abgeleiteten Klasse ist

```
public class PersonContext : DbContext
{
    public PersonContext(DbContextOptions<PersonContext> options): base(options) { }
    public PersonContext() { }

    public DbSet<Person> Persons { get; set; }
}
```

Das hat den Vorteil, dass z.B. auch zur Laufzeit der Datenbank-Provider geändert werden kann, z.B. von SqlServer auf Sqlite. Das DbContextOptions-Objekt wird über einen **DbContextOptionsBuilder** erzeugt, wie wir noch weiter unten sehen werden.

Der Default-Konstruktor wird vor allem für die Migrationen benötigt (siehe ebenfalls beim Thema Migrations).

#### 3.2.1 Fallback mit OnConfiguring

Bei EF Core ist für Dependency Injection ein Fallback-Mechanismus für DbContext implementiert. Die Methode OnConfiguring erhält dabei ein Objekt vom oben erwähnten Typ **DbContextOptionsBuilder**. Damit kann überprüft werden, ob der Context konfiguriert wurde und wenn nicht, kann das dann mit einem hart-kodierten ConnectionString nachgeholt werden.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    Console.WriteLine($"Db OnConfiguring: IsConfigured={optionsBuilder.IsConfigured}");
    if (!optionsBuilder.IsConfigured)
    {
        string connectionString = @"server=(LocalDB)\mssqllocaldb;attachdbfilename=D:\Temp\Persons.mdf;
        Console.WriteLine($"    Using connectionString {connectionString}");
        optionsBuilder.UseSqlServer(connectionString);
    }
}
```

Der Ausdruck `optionsBuilder.IsConfigured` gibt an, ob der DbContext über den Default-Konstruktor erzeugt wurde, oder über jenen mit den DbContextOptions. In ersterem Fall ist dieser Ausdruck **false** und der Connectionstring wird über einen hart kodierten String zugewiesen.

### 3.2.2 Connectionstring

Der Connectionstring wird dabei wie gewohnt angegeben:

```
server=(LocalDB)\mssqllocaldb;attachdbfilename=D:\Temp\Persons.mdf;
database=Persons;integrated security=True;MultipleActiveResultSets=True;
```

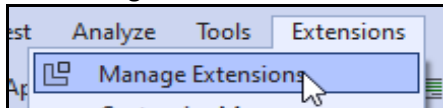
Alle Optionen: <https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlconnection.connectionstring?view=dotnet-plat-ext-6.0&viewFallbackFrom=net-6.0>

### 3.2.3 DbContext Pooling

**Achtung:** möchte man DbContext-Pooling verwenden (wird hier nicht besprochen), darf es nur einen einzigen **public** Konstruktor geben, daher sollte man diesen Defaultkonstruktor **internal** notieren.

## 3.3 Diagramm

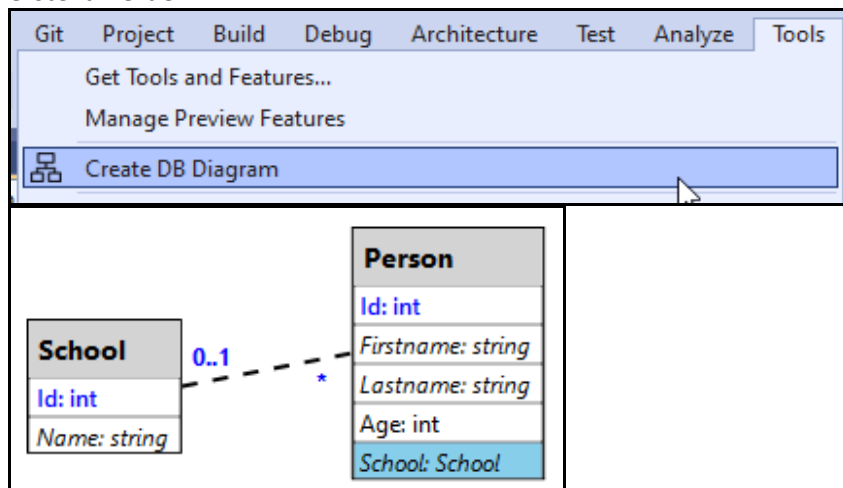
Zum Anzeigen der aktuellen Konfiguration der Tabellenklassen als Diagramm habe ich ein Visual Studio Plugin entwickelt, das die Tabellen so wie im MdfViewer darstellt. Dabei muss die Datenbank aber noch nicht existieren. Dieses Plugin kann mit Extensions → Manage Extensions installieren:



Nach dem Plugin DbDiagramPlugin suchen und installieren:

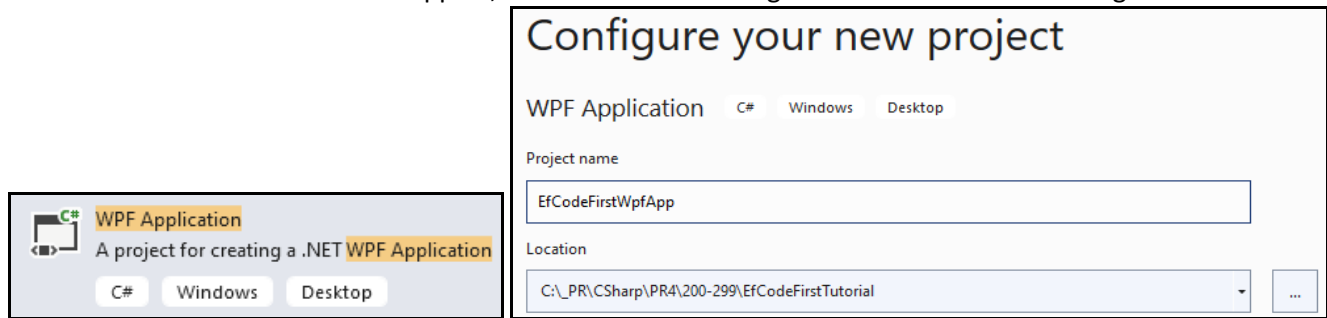
**DbDiagramPlugin**  
 Analyzes the DbContext class of the actual project and draws a database diagram (Tools --> Create DB Diagram) - according to the number of tables, all permutations of a genetic algorithm is used to find the best arrangement of the tables

Danach muss nur noch eine Datei in der Db-Library selektiert und mit Tools → Create DB Diagram das Diagramm erstellt werden:



## 4 WPF Core

Zum Testen bietet sich eine WPF-App an, die man aber unbedingt in der Core-Variante erzeugen muss!

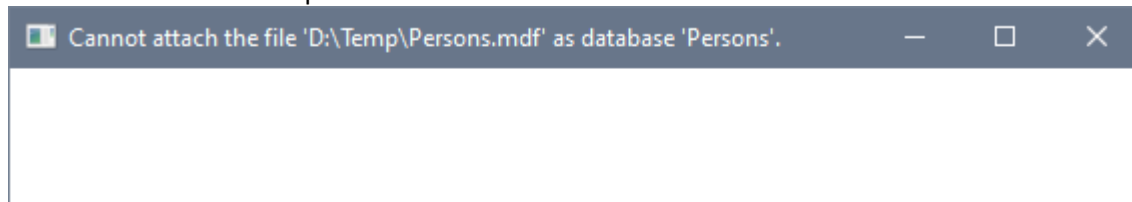


Um den Output auf der Konsole zu sehen, wie gewohnt den <OutputType> auf Exe umstellen.

Hier die Referenz auf das Datenbank-Projekt hinzufügen und im Window\_Loaded wie üblich zur Überprüfung auf die Datenbank zugreifen:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    try
    {
        var db = new PersonContext();
        int nr = db.Persons.Count();
        Title = $"{nr} Persons";
        db.Dispose();
    }
    catch (Exception exc)
    {
        Title = exc.Message;
    }
}
```

Man bekommt eine Exception.



Grund: die Datenbank wird nicht automatisch erstellt.

### 4.1 Datenbank erzeugen

Man muss also explizit dafür sorgen, dass die Datenbank erstellt wird. Das geht über die Methode **EnsureCreated()** der Property **Database** des DbContext:

```
try
{
    var db = new PersonContext();
    db.Database.EnsureCreated();
    int nr = db.Persons.Count();
    Title = $"{nr} Persons";
}
```

Und funktioniert damit auch:



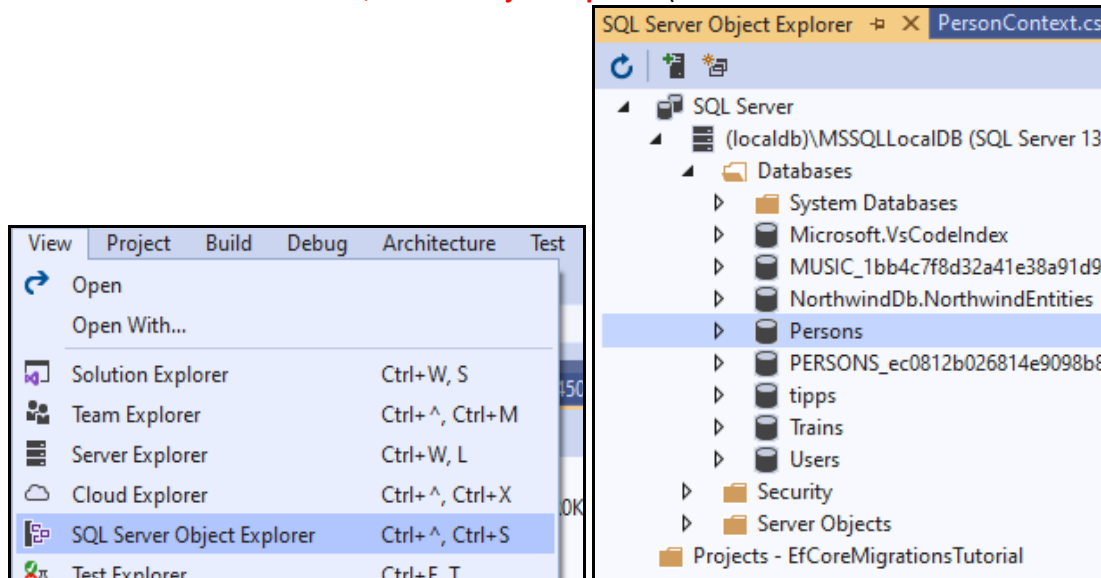


## 4.2 Mögliche Fehler

Falls das Erstellen der Datenbank fehlschlägt, liegt es meist an einem der folgenden beiden Fehler:

### 4.2.1 SQL Server Object Explorer

Löschen der Datenbank im **SQL Server Object Explorer** (markieren und <Entf>-Taste drücken).



### 4.2.2 Database name

Oft vergisst man die Angabe eines Namens im Connectionstring:

**server=(LocalDB) \mssqllocaldb;attachdbfilename=D:\Temp\Persons.mdf;database=Persons;integrated security=True;MultipleActiveResultSets=True;**

## 4.3 appsettings.json (ohne Dependency Injection)

Um einen Connectionstring in WPF Core zu verwenden erzeugt man eine **appsettings.json**-Datei, wobei nur der Eintrag bei Connectionstrings notwendig ist („Copy if newer“ nicht vergessen):

```
{
  "ConnectionStrings": {
    "PersonsMdf": "data source=(LocalDB)\\mssqllocaldb; attachdbfilename=D:\\Temp\\Persons.mdf; databa
  }
}
```

Mit Dependency Injection funktioniert es wie dort besprochen.

Ohne DI verwendet wird der ConnectionString dann so, wie bei Entity Framework Database First gezeigt:

```
private static PersonContext CreateContext()
{
    var config = new ConfigurationBuilder()
        .SetBasePath(AppContext.BaseDirectory)
        .AddJsonFile("appsettings.json")
        .Build();
    var optionsBuilder = new DbContextOptionsBuilder<PersonContext>();

    string connStr = config.GetConnectionString("PersonsMdf");
    Console.WriteLine($"Using database {connStr}");
    optionsBuilder.UseSqlServer(connStr);

    var db = new PersonContext(optionsBuilder.Options);
    return db;
}
```

Und dann die so erzeugte Datenbank verwenden:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    try
    {
        //var db = new PersonContext();
        var db = CreateContext();
        db.Database.EnsureCreated();
        int an = db.Persons.Count();
    }
}
```

Hinweis: man braucht das NuGet-package **Microsoft.Extensions.Configuration.Json**.

## 4.4 DataDirectory

Leider wird DataDirectory von EF Core (meines Wissens) nicht unterstützt. Das ist aber kein allzu großes Problem, weil man das sehr einfach manuell erledigen kann:

```
string dataDirectory = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
string connStr = config.GetConnectionString("PersonsMdf");
if (connStr.Contains("|DataDirectory|")) connStr = connStr.Replace("|DataDirectory|", dataDirectory);
Console.WriteLine($"Using database {connStr}");
optionsBuilder.UseSqlServer(connStr);
```

## 4.5 Sqlite

Jetzt ist es sehr einfach, den Datenbank-Provider auszutauschen. Für Sqlite ist z.B. folgendes zu tun:

### 4.5.1 Paket installieren

Das Paket **Microsoft.EntityFrameworkCore.Sqlite** muss installiert werden (auf eine der vier oben beschriebenen Arten), und zwar nicht in der Library sondern im auszuführenden Projekt:

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="5.0.2" />
</ItemGroup>
```

### 4.5.2 Connectionstring

In appsettings einen zusätzlichen Connectionstring eintragen:

```
"ConnectionStrings": {
  "PersonsMdf": "server=(LocalDB)\\mssqllocaldb; attachdbfi
  "PersonsMdf_": "server=(LocalDB)\\mssqllocaldb; attachdbf
  "PersonsSqlite": "data source=D:\\Temp\\Persons.sqlite"
}
```

### 4.5.3 Service registrieren

Anstelle von UseSqlServer jetzt UseSqlite verwenden:

```
string dataDirectory = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
//string connStr = config.GetConnectionString("PersonsMdf");
string connStr = config.GetConnectionString("PersonsSqlite");
if (connStr.Contains("|DataDirectory|")) connStr = connStr.Replace("|DataDirectory|", dataDirectory);
Console.WriteLine($"Using database {connStr}");
//optionsBuilder.UseSqlServer(connStr);
optionsBuilder.UseSqlite(connStr);
```

## 5 Seeding

Neben dem Befüllen der Datenbank mit einer normalen C#-Methode (z.B. in `Window_Loaded`) kann das Initialisieren mit Stammdaten auch in der Methode **OnModelCreating** des `DbContext` erfolgen.

### 5.1 HasData

Die Idee dabei ist, dass man in der `DbContext`-Klasse festschreibt, welche Daten die Datenbank enthalten soll. Sinnigerweise heißt die Methode dazu dann auch **HasData**. Beim Erstellen einer Migration (siehe später) wird diese Information dann automatisch in entsprechenden C#-Code „übersetzt“, der bei `Up()` bzw. `Down()` ausgeführt wird.

### 5.2 OnModelCreating + ExtensionMethode

Um den Code der Klasse nicht zu unübersichtlich zu gestalten, bietet sich an, das in einer Extensionmethode zu programmieren.

Es wird die Methode `OnModelCreating` in `PersonContext.cs` überschrieben:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Seed();
}
```

Das Eintragen in die Datenbank erfolgt dann mit `modelBuilder.Entity<Person>().HasData` (anstelle von `db.Persons.Add`):

```
public static class DbSeederExtension
{
    public static void Seed(this ModelBuilder modelBuilder)
    {
        Console.WriteLine("DbSeederExtension::Seed");
        modelBuilder.Entity<Person>().HasData(new Person
        {
            Id = 1,
            Firstname = "Hansi",
            Lastname = "Huber",
            Age = 66,
        });
    }
}
```

Nachteil: man muss einen Wert für `Id` angeben!

#### 5.2.1 Erzeugung erzwingen

Noch funktioniert es nicht, weil diese `Seed` nur – wie der Name `OnModelCreating` vermuten lässt – beim Erstellen der Datenbank ausgeführt wird. Man muss also erzwingen, dass die Datenbank neu erstellt wird. Das geht mit `db.Database.EnsureCreated()`.

```
var db = new PersonContext();
db.Database.EnsureDeleted();
db.Database.EnsureCreated();
int nr = db.Persons.Count();
Title = $"{nr} Persons";
```



#### 5.2.2 CSV

Mit einer CSV-Datei funktioniert es genauso, d.h. man schreibt in der Schleife einfach `HasData()` statements.

### 5.2.3 Foreign Keys / Navigation Properties

Aufpassen muss man bei Foreign Keys. Man muss neben der Navigation Property auch noch die Id als Foreign Key angeben. Diese ForeignKey-Id muss einer bestimmten Struktur entsprechen, nämlich im einfachsten Fall `{TargetType}{TargetTypeKey}` (es gibt auch noch andere Möglichkeiten).

```
public class Person
{
    public int Id { get; set; }
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public int Age { get; set; }
    public School School { get; set; }
    public int SchoolId { get; set; }
```

Hier darf man nicht das Objekt zuweisen, sondern muss die Id benutzen:

```
var school = new School { Id = 1, Name = "HTL Grieskirchen" };
modelBuilder.Entity<School>().HasData(school);

modelBuilder.Entity<Person>().HasData(new Person
{
    Id = 1,
    Firstname = "Hansi",
    Lastname = "Huber",
    Age = 66,
    //School=school, //<----- NO!!!!!!
    SchoolId = 1,
});
```

Falls die Id in der referenzierten Tabelle nicht Id ist, sondern anders heißt, muss auch die Property entsprechend angepasst werden:

```
class Person
{
    public int PersonId { get; set; }
    public string Firstname { get; set; }
    public int CompanyCompanyId { get; set; }
    public Company Company { get; set; }
}

class Company
{
    public int CompanyId { get; set; }
    public virtual List<Person> Persons { get; set; }
}
```

## 6 Convention over Configuration

Wenn aus den Klassen/Properties später Tabellen/Spalten in der Datenbank erzeugt werden, müssen logischerweise bestimmte Regeln angewendet werden, nach denen dies geschieht. Dabei wird bei Entity Framework Code First der Ansatz gewählt, dass möglichst wenig zusätzliche Information für diese Generierungsschritte angegeben werden muss. Es werden also allgemeine **Konventionen** angewendet, statt alles zu **konfigurieren** - kurz: Convention over Configuration.

Durch **Annotations** kann dieses Standardverhalten aber überschrieben werden. Diese Annotations befinden sich im Namespace **System.ComponentModel.DataAnnotations** bzw.

**System.ComponentModel.DataAnnotations.Schema**.

### 6.1 Tabellennamen/Spaltennamen/Spaltentypen

#### Convention:

- **Tabellen** heißen wie die **Klassen**, werden aber automatisch pluralisiert (Customer --> Customers).  
Achtung: aus Person wird nicht Persons, sondern People!
- **Spalten** heißen wie die **Properties**
- Der **Datentyp** der Spalten ergibt sich automatisch aus dem **C#-Typ**

#### Configuration:

- Mit **[Table("MyTableName")]** kann man den Tabellennamen festlegen
- Mit **[Column("Quaxi")]** kann man den Spaltennamen konfigurieren
- Mit **[Column(TypeName="xxx")]** kann man einen (datenbankspezifischen) Typ angeben

Beispiele:

#### Convention

```
class Person
{
    public int PersonId { get; set; }
    public string Firstname { get; set; }
    public int age { get; set; }
    public DateTime Birthdate { get; set; }
}
```

```
CREATE TABLE [People]
(
    [PersonId] INT NOT NULL IDENTITY (1,1),
    [Firstname] NVARCHAR(4000),
    [age] INT NOT NULL,
    [Birthdate] DATETIME NOT NULL
);
```

#### Configuration

```
[Table("Persons")]
class Person
{
    public int PersonId { get; set; }
    public string Firstname { get; set; }
    [Column("Alter")]
    public int age { get; set; }
    public DateTime Birthdate { get; set; }
}
```

```
CREATE TABLE [Persons]
(
    [PersonId] INT NOT NULL IDENTITY (1,1),
    [Firstname] NVARCHAR(4000),
    [Alter] INT NOT NULL,
    [Birthdate] DATETIME NOT NULL
);
```

### 6.2 Primary Key

#### Convention:

- heißt eine Property **Id** bzw. **{Classname}Id**, so wird daraus die **Primary Key**-Spalte erzeugt
- Ist die Id-Property vom Typ **int**, wird automatisch eine **Autonumber-Spalte** generiert

#### Configuration:

- Durch die Annotation **[Key]** kann man eine Spalte, die nicht automatisch als Primary Key erkannt wird, zu einem **Primary Key** machen

Beispiele:

```
class Person
{
    public int PersonId { get; set; }
    public string Firstname { get; set; }
    public Company Company { get; set; }
}
```

```
CREATE TABLE [People]
(
    [PersonId] INT NOT NULL IDENTITY (1,1),
    [Firstname] NVARCHAR(4000),
    [Company_CompanyId] INT
);
ALTER TABLE [People] ADD CONSTRAINT [PK__People__000000000000004E] PRIMARY KEY ([PersonId]);
```

```
class Person
{
    public int Personalnummer { get; set; }
    public string Firstname { get; set; }
    public Company Company { get; set; }
}
```

System.Data.Entity.ModelConfiguration.ModelValidationException: One or more validation errors were detected during model generation:

System.Data.Edm.EdmEntityType: : EntityType 'Person' has no key defined. Define the key for this EntityType.

```
class Person
{
    [Key]
    public int Personalnummer { get; set; }
    public string Firstname { get; set; }
    public Company Company { get; set; }
}
```

```
CREATE TABLE [People]
(
    [Personalnummer] INT NOT NULL IDENTITY (1,1),
    [Firstname] NVARCHAR(4000),
    [Company_CompanyId] INT
);
ALTER TABLE [People] ADD CONSTRAINT [PK__People__000000000000004E] PRIMARY KEY ([Personalnummer]);
```

## 6.3 NULL - NOT NULL

### Convention:

- **Value** Types sind **NOT NULL**
- **Reference** Types sind **NULL**
- **Nullable Types**, wie z.B. `int?`, sind **NULL**

### Configuration:

- Annotiert man eine Property mit `[Required]`, wird diese als **NOT NULL** column generiert

### Beispiele:

```
class Person
{
    public int PersonId { get; set; }
    public string Firstname { get; set; }
    public int age { get; set; }
    public DateTime Birthdate { get; set; }
}
```

```
CREATE TABLE [People]
(
    [PersonId] INT NOT NULL IDENTITY (1,1),
    [Firstname] NVARCHAR(4000),
    [age] INT NOT NULL,
    [Birthdate] DATETIME NOT NULL
);
```

```
class Person
{
    public int PersonId { get; set; }
    [Required]
    public string Firstname { get; set; }
    public int? age { get; set; }
    public DateTime? Birthdate { get; set; }
}
```

```
CREATE TABLE [People]
(
    [PersonId] INT NOT NULL IDENTITY (1,1),
    [Firstname] NVARCHAR(4000) NOT NULL,
    [age] INT,
    [Birthdate] DATETIME
);
```

## 6.4 NotMapped

Manchmal möchte man in Entity-Klassen Properties definieren, die nicht als Spalten in der Tabelle enden sollen. Das erreicht man durch die Annotation **NotMapped**:

```
public class Student
{
    public int Id { get; set; }
    public string Firstname { get; set; }
    public string Lastname { get; set; }

    [NotMapped]
    public string Name => $"{Firstname} {Lastname}";
}
```

## 6.5 String-columns

Für Strings gibt es noch folgende weitere Attribute:

**MinLength** Minimale Länge des String.

**MaxLength** Maximale Länge des String

## 6.6 Index

Möchte man explizit für eine Spalte einen Index vergeben, kann man das mit dem Attribut Index konfigurieren.

```
[Index("IDX_Lastname", IsUnique = false)]
public string Lastname { get; set; }
```

Die Parameter können weggelassen werden, der Name ist dann **IX\_Propertyname**. Der Default für **IsUnique** ist false.