

Entity Framework - Database First

Inhalt

1 EINFÜHRUNG	2
2 ERSTES PROJEKT	3
2.1 Datenbank	3
2.2 NuGet	3
2.3 Dotnet tools	4
2.3.1 Überprüfen	4
2.3.2 Installieren	4
2.3.3 Aktualisieren	4
2.4 Data Model erzeugen	5
2.4.1 Connectionstring	5
2.4.2 Windows Konsole	5
2.4.3 Sqlite	5
2.4.4 Package Manager Console	6
3 ERZEUGTER CODE	7
3.1 Zugriffsklassen	7
3.2 Datenbank-Kontext	8
3.3 OnConfiguring	8
3.4 Diagramm erzeugen	8
4 DATEN PRÜFEN	10
4.1 Server Explorer	10
4.2 MdfViewer	11
5 VERWENDEN	12
5.1 Library zu Projekt hinzufügen	12
5.2 NuGet	12
5.3 DbContext	12
5.4 Testen	13
5.5 Abfrage	13
5.5.1 Navigation Properties	13
5.5.2 Mit Joins	13
5.6 SQL-Statement	14
5.1 Log SQL	14
5.2 Include = Force Join	14
5.2.1 Falsch	15
5.2.2 Richtig	15
5.2.3 AutoJoin	15
5.3 Verzögerte Ausführung	16
5.3.1 Client-side evaluation	16
5.4 ConnectionString	17
5.4.1 appsettings.json	17
5.4.2 Nuget	17
5.4.3 ConfigurationBuilder	17
5.4.4 Relativer Pfad - DataDirectory	18
5.4.5 Sqlite	18

1 Einführung

Entity Framework ist eine Variante des Datenbankzugriffs von C# (eine andere wäre z.B. Typed DataSets). Dabei unterscheidet man drei Varianten:

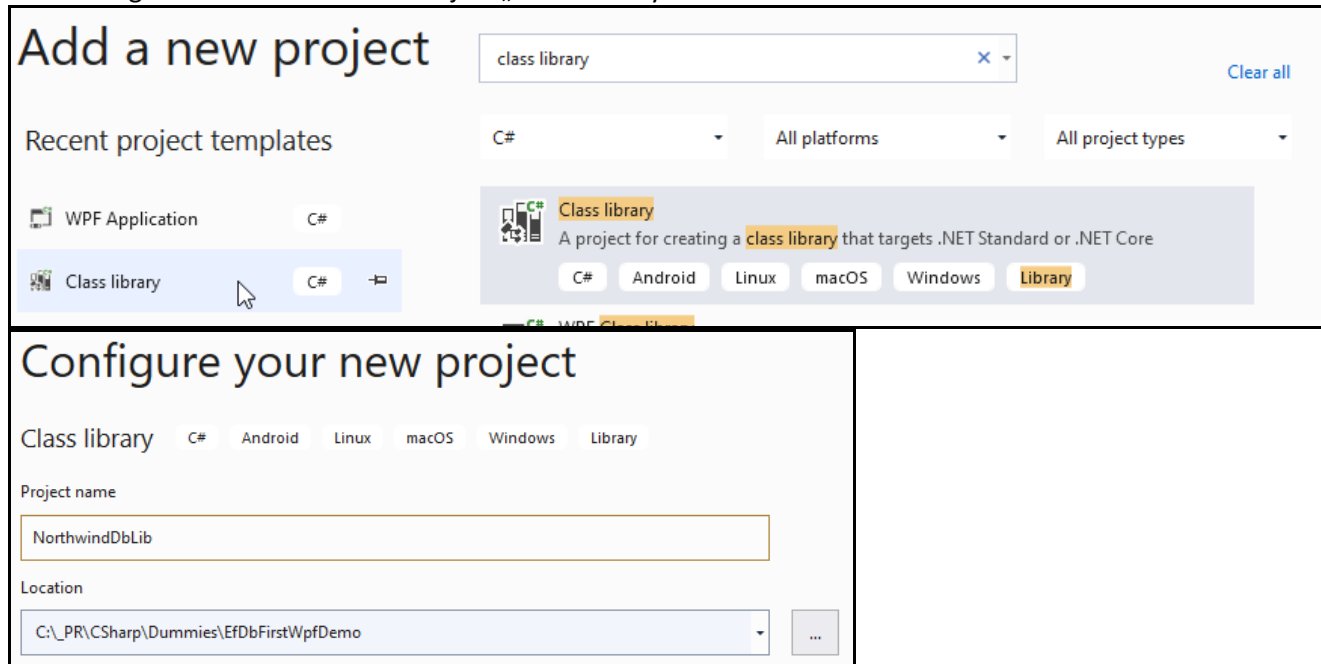
- **Database First:** die Datenbank existiert u. man lässt sich C#-Zugriffsklassen automatisch erzeugen
- **Model First:** man modelliert graphisch die Tabellen u. deren Abhängigkeiten. Daraus lässt man sich einerseits die Datenbank erzeugen (mit SQL-Statements), andererseits auch die C#-Zugriffsklassen
- **Code First:** Man beginnt mit einfachen C#-Klassen (POCOs), aus denen man sich dann eine Datenbank erzeugen lässt.

Die einfachste Variante ist die, dass eine Datenbank bereits zur Verfügung stellt und man sich C#-Zugriffsklassen automatisch erzeugen lässt. Mit diesen Klassen greift man über LINQ dann auf die Daten zu.

Wir werden jene Variante benutzen, bei der die Datenbank als **MDF**-Datei oder Sqlite-Datei vorliegt. Letztendlich wird aber über einen **Connection-String** festgelegt, auf welche Datenbank man zugreift.

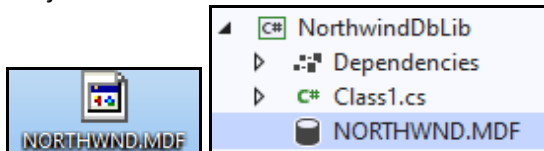
2 Erstes Projekt

Man beginnt mit einem ganz einfachen WPF-Projekt.
Diesem fügt man ein zusätzliches Projekt „Class Library“ hinzu.



2.1 Datenbank

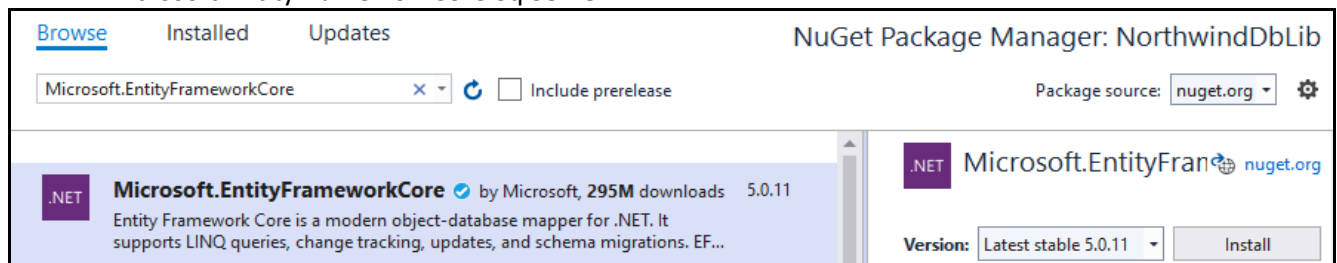
Dann fügt man diesem Projekt eine MDF-Datenbank-Datei hinzu. Dazu zieht man einfach die MDF-Datei in das Projekt:



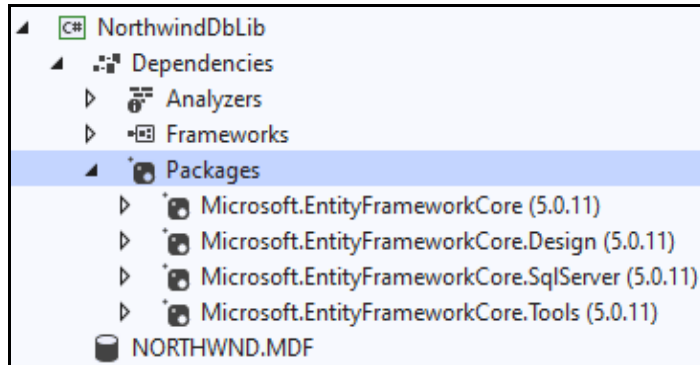
2.2 NuGet

Um jedoch Entity Framework verwenden zu können, müssen die entsprechenden Assemblies installiert werden. Das geht am besten mit NuGet.

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



Danach muss das Projekt mit **Build -> Rebuild Solution** erstellt werden.
Dadurch entstehen zusätzliche Referenzen bei diesen Projekten:



Die Information, welche Pakete installiert werden sollen, wird wie immer in das .csproj-File geschrieben:

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

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="5.0.11" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="5.0.11">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="5.0.11" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="5.0.11">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
  </ItemGroup>
</Project>
```

2.3 Dotnet tools

Das Erstellen der Zugriffsklassen auf die Datenbank erfolgt mit einem separaten Tool namens dotnet-ef. Dieses muss installiert werden, und zwar einmalig global (und nicht für jedes Projekt).

2.3.1 Überprüfen

dotnet-ef --version

```
C:\_PR\CSharp\PR4\EfDbFirstWpfDemo\NorthwindDbLib>dotnet-ef --version
Entity Framework Core .NET Command-line Tools
5.0.3
```

2.3.2 Installieren

siehe: <https://docs.microsoft.com/en-us/ef/core/cli/dotnet#installing-the-tools>

dotnet tool install --global dotnet-ef [--version 5.0.3]

```
C:\_PR\CSharp\PR4\EfDbFirstWpfDemo\NorthwindDbLib>dotnet tool install --global dotnet-ef --version 5.0.3
Tool 'dotnet-ef' is already installed.
```

2.3.3 Aktualisieren

siehe: <https://docs.microsoft.com/en-us/ef/core/cli/dotnet#update-the-tools>

dotnet tool update --global dotnet-ef

```
C:\_PR\CSharp\PR4\EfDbFirstWpfDemo\NorthwindDbLib>dotnet tool update --global dotnet-ef
Tool 'dotnet-ef' was successfully updated from version '5.0.3' to version '5.0.11'.
```

2.4 Data Model erzeugen

Mit diesen Vorbereitungen kann man sich jetzt die Zugriffsklassen erzeugen lassen. Das kann über die Windows Konsole oder über die Package Manager Console durchführen.

Dabei muss man als wichtigsten Parameter angeben, um welche Datenbank es sich handelt und wo diese liegt. Das erfolgt über einen sogenannten Connectionstring.

2.4.1 Connectionstring

Ein Connectionstring legt fest, welche Datenbank man verwenden möchte und ist verständlicherweise abhängig vom jeweiligen Datenbank-Provider. Für SQL Server sieht er so aus:

```
"Server=(LocalDB)\mssqllocaldb;  
attachdbfilename=C:\_PR\CSharp\PR4\EfDbFirstWpfDemo\NorthwindDbLib\Northwnd.m  
df;integrated security=True; MultipleActiveResultSets=True"
```

Alle Details zu den verschiedenen Optionen siehe <https://www.connectionstrings.com/all-sql-server-connection-string-keywords/>

2.4.2 Windows Konsole

Dazu ist es wichtig, dass man ins richtige Verzeichnis wechselt – nämlich ins Root-Verzeichnis des Db-Projekts (also dorthin, wo das csproj-File liegt).

```
C:\_PR\CSharp\PR4\EfDbFirstWpfDemo\NorthwindDbLib>dir /b  
bin  
NorthwindDbLib.csproj  
NORTHWND.MDF  
Northwnd_log.ldf  
obj
```

Dort muss dann folgender Befehl ausgeführt werden (Details siehe <https://docs.microsoft.com/en-us/ef/core/cli/dotnet>):

Allgemein: **dotnet ef dbcontext scaffold connectionstring provider options**

- **ConnectionString:** muss bestimmtes Format haben, da darf man sich nicht vertippen
- **Provider:** gibt an, welche Art von Datenbank man benutzt. In unserem Fall ist das eine SqlServer-Datenbank und daher ist der Provider **Microsoft.EntityFrameworkCore.SqlServer**
- **Options:**
 - **-f:** overwrite existing files
 - **-c:** The name of the DbContext class to generate.

Alle Optionen: <https://docs.microsoft.com/en-us/ef/core/cli/dotnet#common-options>

Damit ergibt sich folgender Befehl:

```
dotnet ef dbcontext scaffold "Server=(LocalDB)\mssqllocaldb;  
attachdbfilename=C:\_PR\CSharp\PR4\EfDbFirstWpfDemo\NorthwindDbLib\Northwnd.m  
df;integrated security=True; MultipleActiveResultSets=True"
```

```
Microsoft.EntityFrameworkCore.SqlServer -f -c NorthwindContext
```

```
C:\_PR\CSharp\PR4\EfDbFirstWpfDemo\NorthwindDbLib>dotnet ef dbcontext scaffold "Server=(LocalDB)\mssqllocaldb; attac  
Build started...  
Build succeeded.  
To protect potentially sensitive information in your connection string, you should move it out of source code. You c  
C:\_PR\CSharp\PR4\EfDbFirstWpfDemo\NorthwindDbLib>_
```

2.4.3 Sqlite

Für Sqlite muss das Paket Microsoft.EntityFrameworkCore.Sqlite installiert werden. Der Befehl sieht dann so aus:

```
dotnet ef dbcontext scaffold "data  
source=C:\_PR\CSharp\PR4\EfDbFirstWpfDemo\NorthwindDbLib\Northwnd.sqlite"  
Microsoft.EntityFrameworkCore.Sqlite -f -c NorthwindContext
```

Hinweis: Datumswerte werden nicht als DateTime erzeugt, sondern als byte[]!

Momentan gibt es derzeit meines Wissens keine andere Möglichkeit, als das manuell im generierten Code auszubessern.

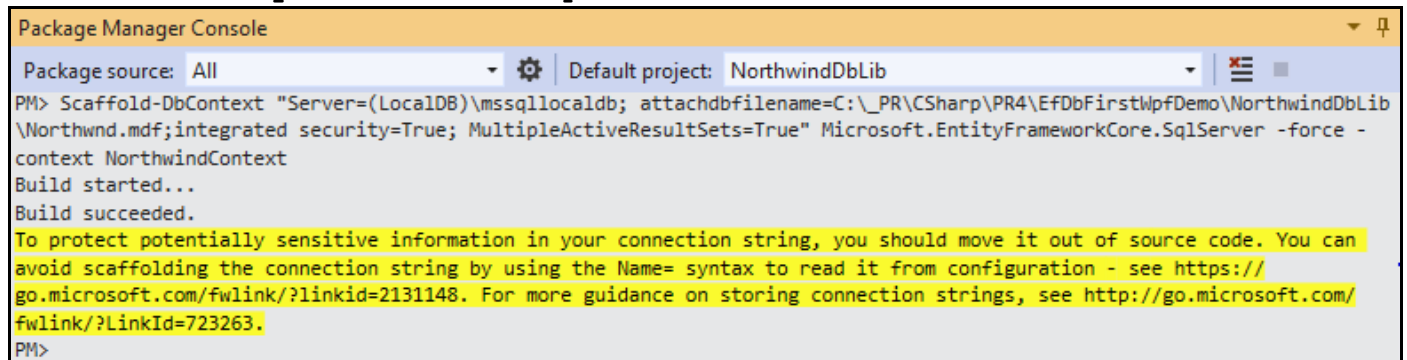
2.4.4 Package Manager Console

Sehr ähnlich geht es auch direkt im Visual Studio mit der Package Manager Console (View → Other Windows → Package Manager Console). Anstelle von `dotnet ef dbcontext scaffold` schreibt man **Scaffold-DbContext** und muss Folgendes anpassen:

- **-c** durch **-context**
- **-f** durch **-force** ersetzen
- Das richtige Projekt muss als „Startup Project“ ausgewählt werden (rechte Maustaste auf Projekt → „Set as Startup Project“)

Details: <https://docs.microsoft.com/en-us/ef/core/cli/powershell#scaffold-dbcontext>

```
Scaffold-DbContext "Server=(LocalDB)\mssqllocaldb; attachdbfilename=C:\_PR\CSharp\PR4\EfDbFirstWpfDemo\NorthwindDbLib\Northwnd.mdf;integrated security=True; MultipleActiveResultSets=True" Microsoft.EntityFrameworkCore.SqlServer -force -context NorthwindContext
```



The screenshot shows the Package Manager Console window with the following content:

```
Package source: All | Default project: NorthwindDbLib
PM> Scaffold-DbContext "Server=(LocalDB)\mssqllocaldb; attachdbfilename=C:\_PR\CSharp\PR4\EfDbFirstWpfDemo\NorthwindDbLib\Northwnd.mdf;integrated security=True; MultipleActiveResultSets=True" Microsoft.EntityFrameworkCore.SqlServer -force -context NorthwindContext
Build started...
Build succeeded.
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.
PM>
```

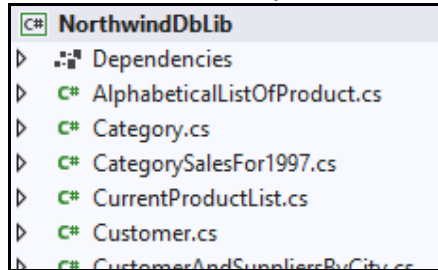
Das Ergebnis ist dann aber dasselbe.

3 Erzeugter Code

Folgender Code und Dateien wurden damit erzeugt.

3.1 Zugriffsklassen

Der Generator hat für jede Tabelle eine eigene C#-Klasse erzeugt:



Wie man sieht, werden die Klassen in das root-Verzeichnis erzeugt (falls man bei Erzeugen nicht mit -o ein anderes angegeben hat).

Diese Klassen sehen dann z.B. so aus:

```
public partial class Category
{
    public Category()
    {
        Products = new HashSet<Product>();
    }

    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public string Description { get; set; }
    public byte[] Picture { get; set; }

    public virtual ICollection<Product> Products { get; set; }
}
```

```
public partial class Product
{
    public Product()
    {
        OrderDetails = new HashSet<OrderDetail>();
    }

    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public int? SupplierId { get; set; }
    public int? CategoryId { get; set; }
    public string QuantityPerUnit { get; set; }
    public decimal? UnitPrice { get; set; }
    public short? UnitsInStock { get; set; }
    public short? UnitsOnOrder { get; set; }
    public short? ReorderLevel { get; set; }
    public bool Discontinued { get; set; }

    public virtual Category Category { get; set; }
    public virtual Supplier Supplier { get; set; }
    public virtual ICollection<OrderDetail> OrderDetails { get; set; }
}
```

Man sieht auch, dass die Foreign Keys durch sogenannte **Navigation Properties** umgesetzt werden, z.B. **Category Category**. Zusätzlich wurde auch noch der Foreign Key selbst erzeugt (also z.B. **int? CategoryID**).

3.2 Datenbank-Kontext

In einer separaten Klasse, dem sog. DbContext, wird ein **DbSet** für jede Tabelle erzeugt. Das kann man sich wie eine Collection vorstellen. Der Code dazu steht in der Datei NorthwindContext.cs (bzw. wie man eben das Model bei der Erzeugung genannt hat).

Die Properties selbst sind dann im Plural (**Categories**), die Klassen selbst im Singular (**Category**).

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

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

    public virtual DbSet<AlphabeticalListOfProduct> AlphabeticalListOfProducts { get; set; }
    public virtual DbSet<Category> Categories { get; set; }
    public virtual DbSet<CategorySalesFor1997> CategorySalesFor1997s { get; set; }
    public virtual DbSet<CurrentProductList> CurrentProductLists { get; set; }
    public virtual DbSet<Customer> Customers { get; set; }
    public virtual DbSet<CustomerAndSuppliersByCity> CustomerAndSuppliersByCities { get; set; }
    ...
    public virtual DbSet<SummaryOfSalesByYear> SummaryOfSalesByYears { get; set; }
    public virtual DbSet<Supplier> Suppliers { get; set; }
    public virtual DbSet<Territory> Territories { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
    }

    partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
}
```

Hier kommt das zweite Häkchen ins Spiel: die Property heißt **Categories**, die einzelnen Objekte **Category**.

3.3 OnConfiguring

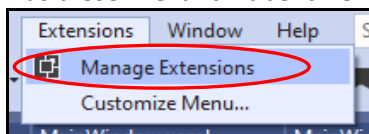
Die Frage ist jetzt noch, wo eigentlich gespeichert ist, welche MDF-Datei benutzt wird.

Diese Info wird ebenfalls in die Datei **NorthwindContext.cs** geschrieben, und zwar in die Methode OnConfiguring (mit einem entsprechenden Hinweis, dass das keine gute Idee ist...):

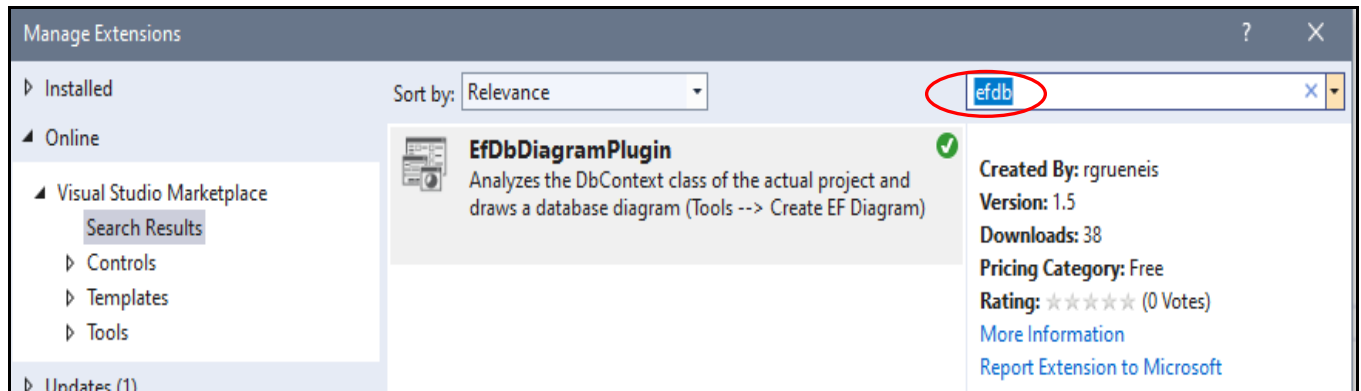
```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        #warning To protect potentially sensitive information in your connection string, you should move it out of source code and use environment configuration or set values in an app.json file.
        optionsBuilder.UseSqlServer("data source=(LocalDB)\\mssqllocaldb; attachdbfilename=C:\\_PR\\CSharp\\PR4\\EfDb\\NorthwindContext.mdf");
    }
}
```

3.4 Diagramm erzeugen

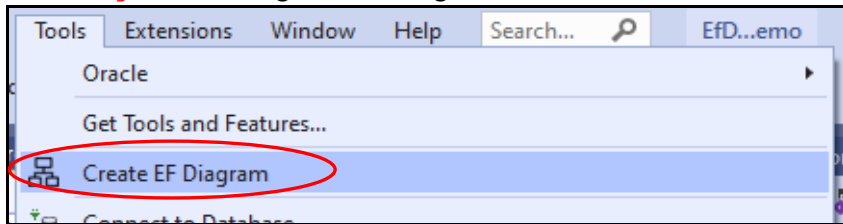
Leider kann man in der aktuellen Version aus den Entity-Klassen das Model-Diagramm nicht erstellen zu können. Aus diesem Grund habe ich ein Visual Studio Plugin erstellt, mit dem das dann möglich ist.



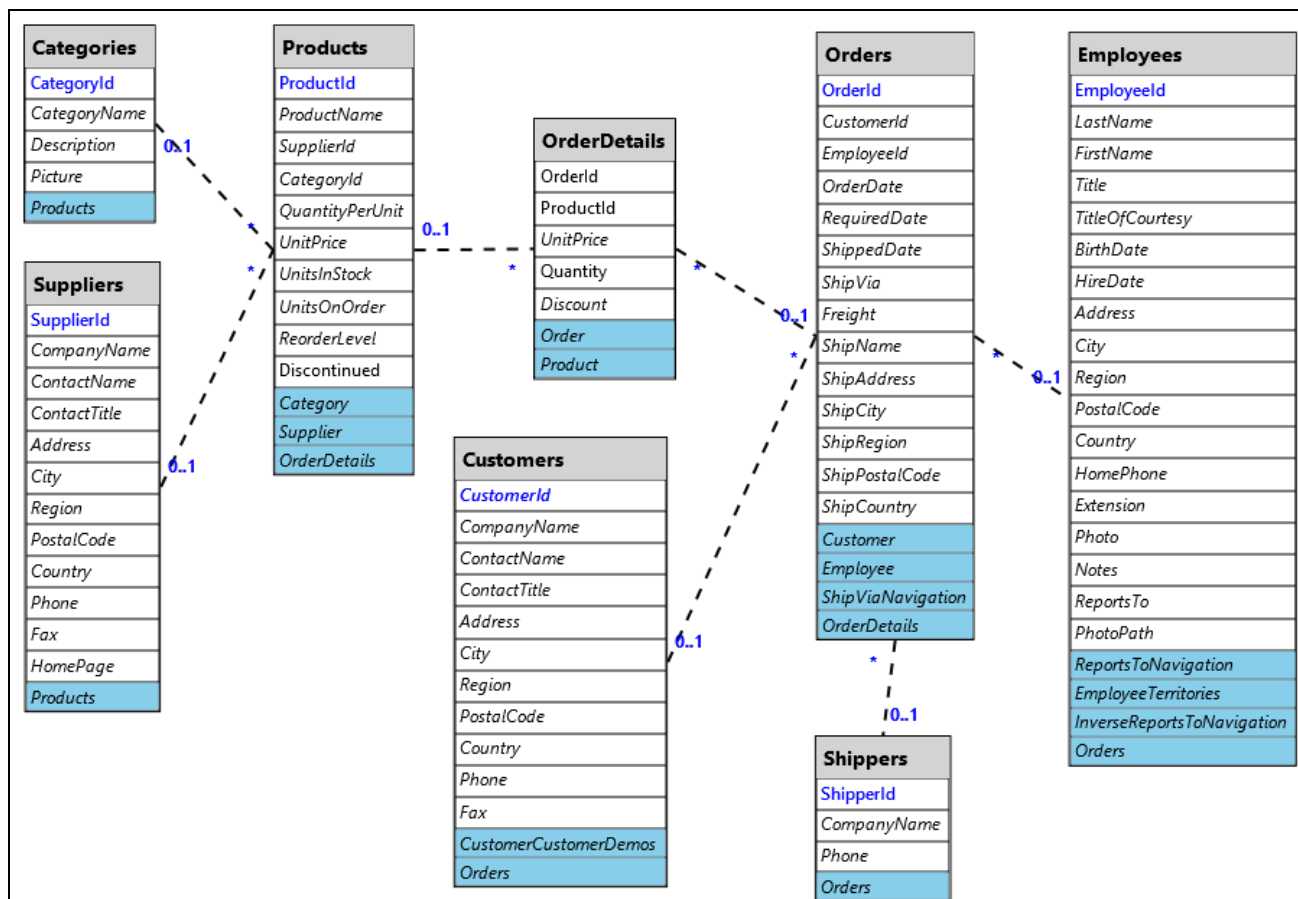
Suchen nach **EfDbDiagramPlugin**:



Man muss nur eine Datei im Projekt mit dem Datenbank-Context selektieren und dann mit **Tools** → **Create EF Diagram** das Diagramm erzeugen



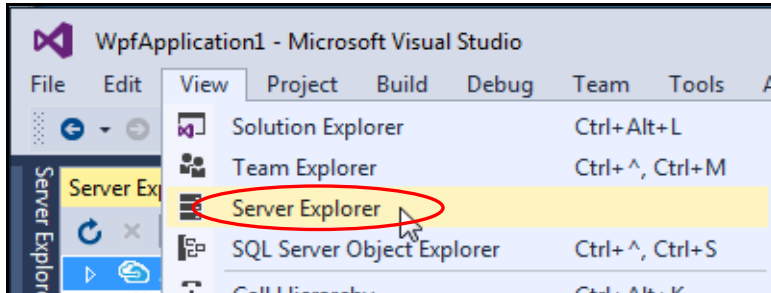
Dieses erzeugt dann ein Datenbank-Diagramm. Das Diagramm kann man anpassen (Tabellen löschen und verschieben, sowie mit <Strg>+Mausrad zoomen). Das sollte man vor allem bei umfangreicheren Datenbanken tun, um einen vernünftigen Überblick über die Abhängigkeiten zu bekommen:



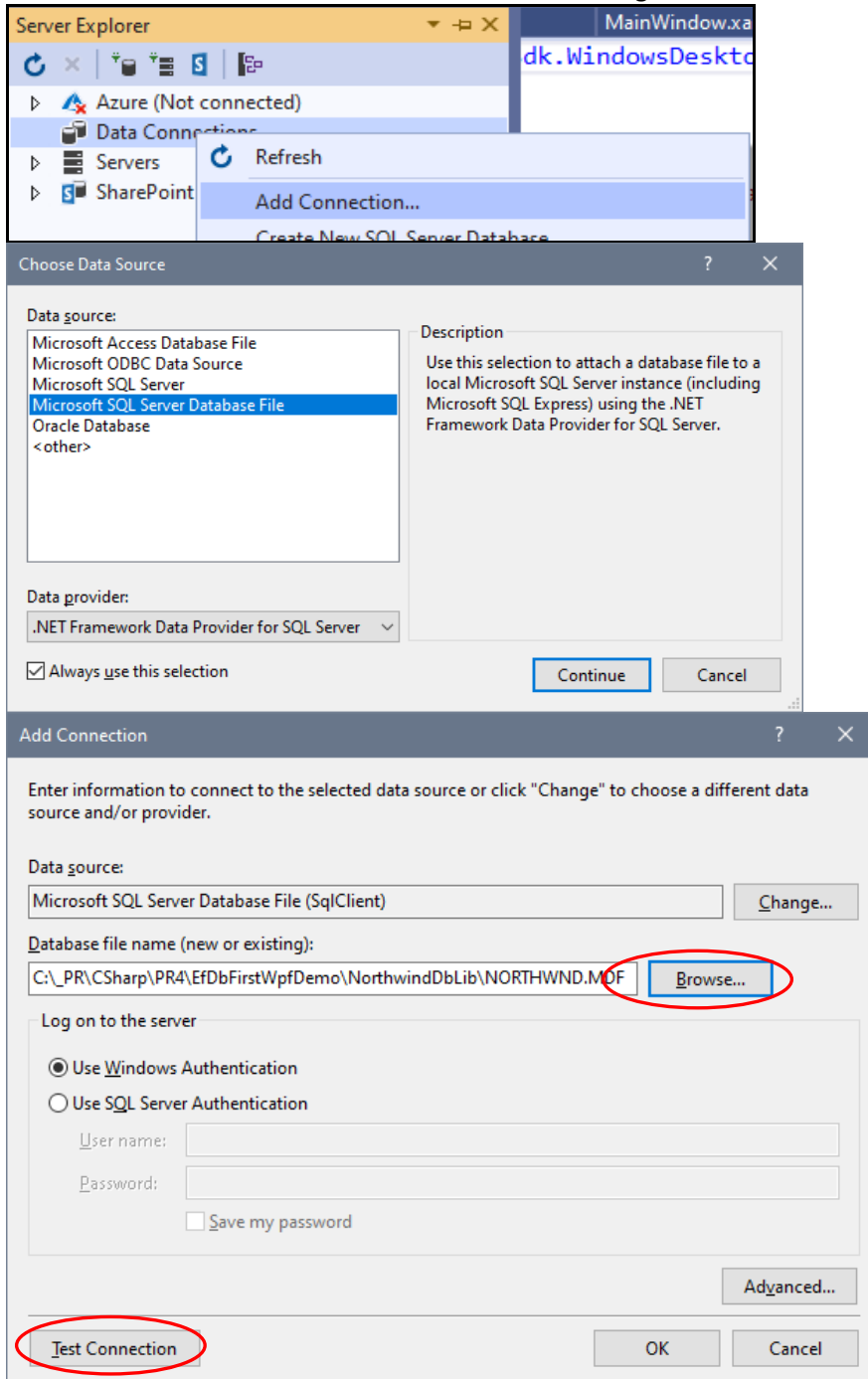
4 Daten prüfen

4.1 Server Explorer

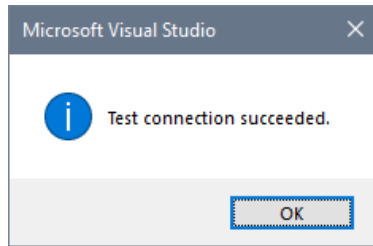
Man kann sich die Daten in der Datenbank direkt im Visual Studio anschauen. Dazu muss **Server Explorer** eingeblendet werden:



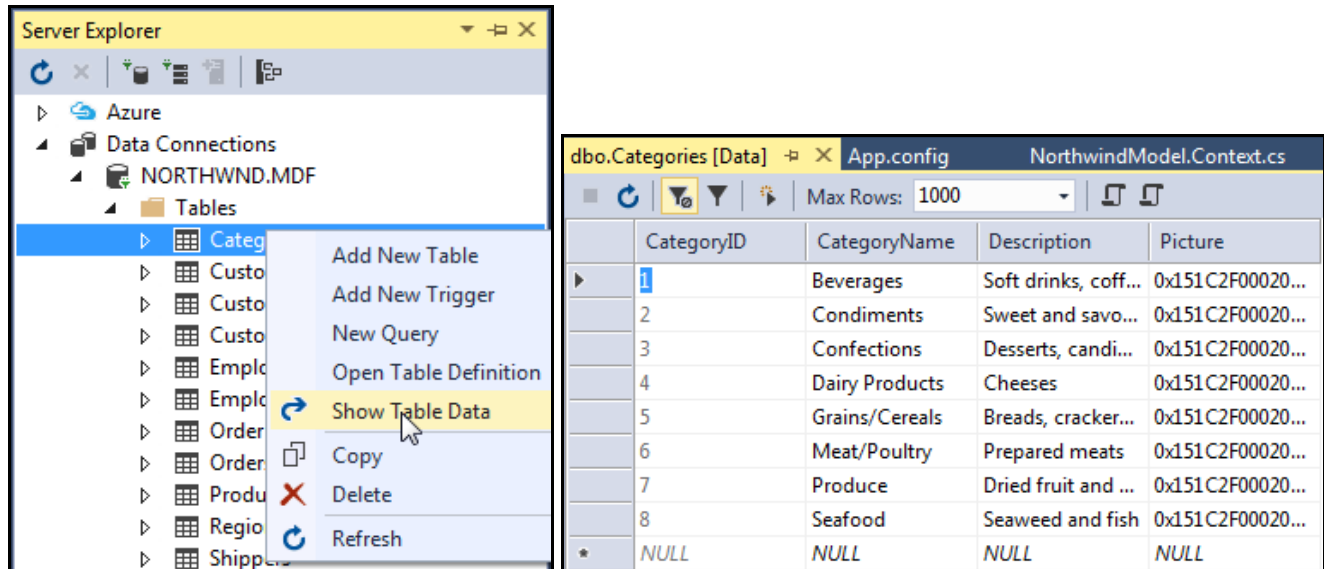
Dort muss man bei Data Connections eine Verbindung zum MDF-File herstellen:



Eventuell mit Test Connection die Verbindung testen.



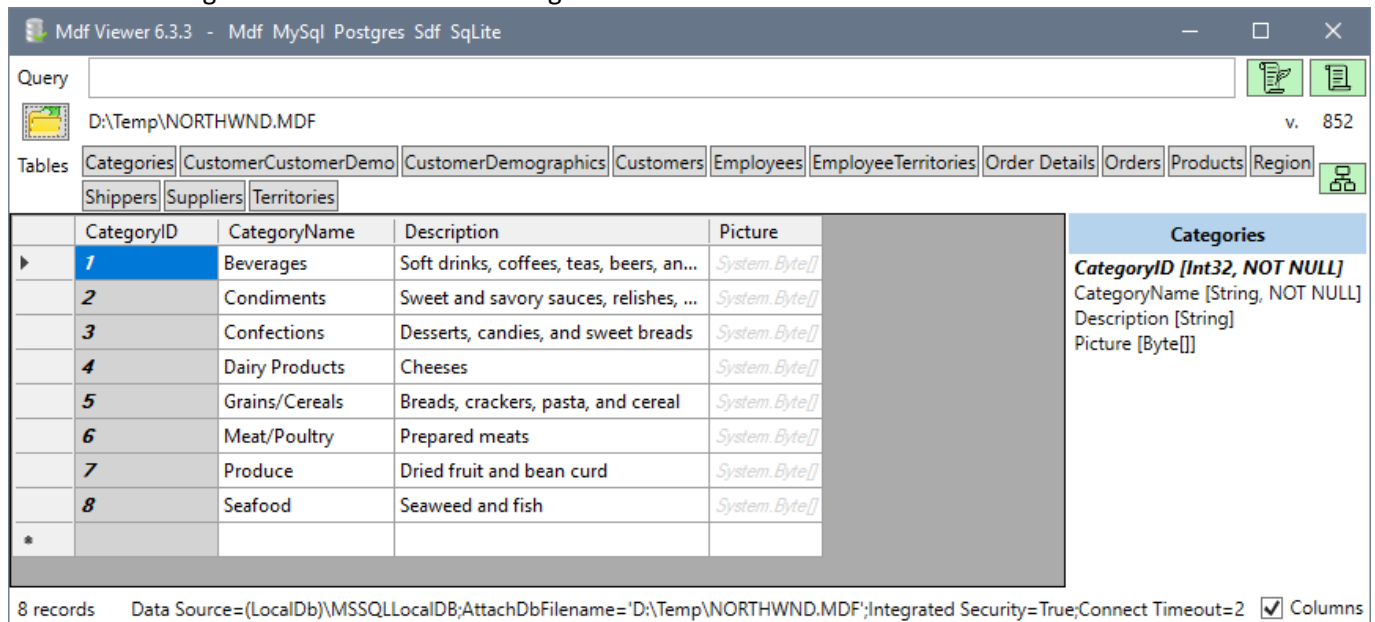
Dort erscheint dann die Datenbank und man kann sich mit dem Kontextmenü „**Show Table Data**“ die Daten ansehen.



4.2 MdfViewer

Da die Anzeige der Daten im Visual Studio etwas umständlich ist, habe ich ein kleines Programm gemacht, mit dem man sich die Tabellen etwas einfacher anschauen kann (zumindest meiner Meinung nach).

Die Beschreibung dazu findet sich in einem eigenen Dokument.

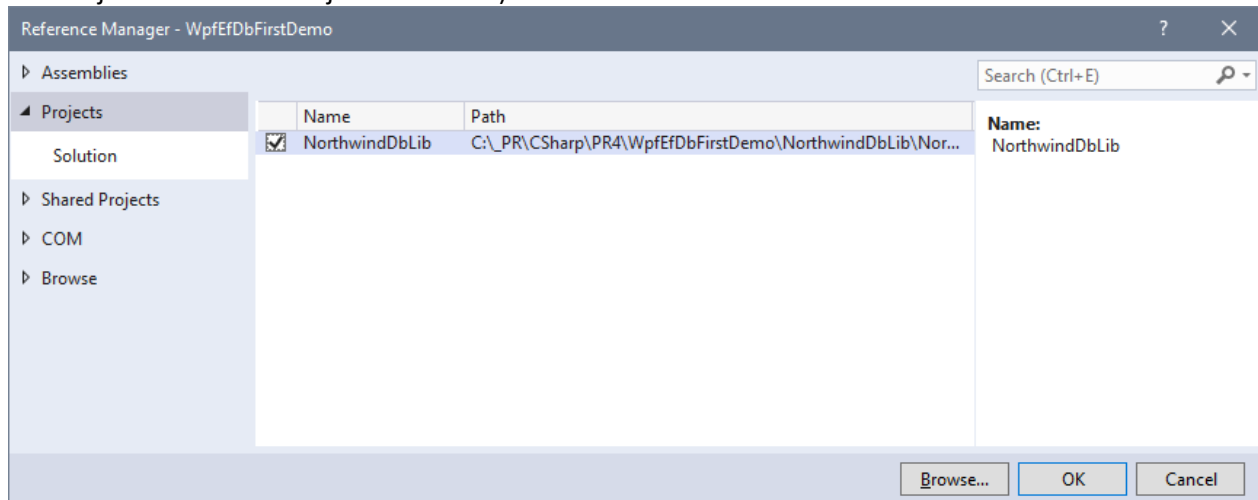


5 Verwenden

Bei der Verwendung muss man praktisch nicht an eine Datenbank denken – man arbeitet so wie gewohnt mit LINQ.

5.1 Library zu Projekt hinzufügen

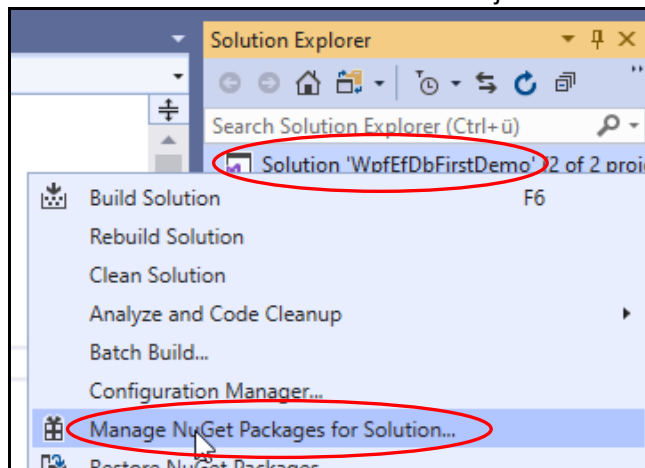
Wie auch bei anderen Libraries gewohnt, muss das verwendende Projekt die Library referenzieren (Kontextmenü des Projekts → Add → Project Reference).



5.2 NuGet

Es müsste zwar auch ohne zusätzliche Referenzen funktionieren, im Zweifelsfall aber auch im WPF-Projekt die EntityFrameworkCore-Pakete installieren.

Dazu kann man entweder bei beiden Projekten das Paket installieren oder auch auf einmal für beide Projekte:



5.3 DbContext

Man muss sich zuerst eine **Referenz auf die Datenbank** besorgen. Das erfolgt über den oben erwähnten DbContext:

```
var db = new NorthwindContext();
```

Falls man nicht weiß, wie man den DbContext genannt hat – es ist jene Klasse, die von DbContext abgeleitet ist und die man bei Erzeugen angegeben hat – üblicherweise als **DatabaseNameContext.cs**, also im Beispiel **NorthwindContext**:

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

5.4 Testen

Um zu testen, ob alles so weit passt, empfehle ich, eine ganz einfache Abfrage zu erstellen und das Ergebnis im Window-Title anzuzeigen.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    var db = new NorthwindContext();
    try
    {
        int nrCategories = db.Categories.Count();
        Title = $"{nrCategories} Categories in Db";
    }
    catch (Exception exc)
    {
        Title = exc.Message;
    }
}
```

5.5 Abfrage

Die LINQ-Abfragen funktionieren genauso, wie bei den letzten Übungen, wo wir mit IEnumerable- bzw. List-Objekten gearbeitet haben.

5.5.1 Navigation Properties

Anstelle der JOINS sollte man bei LINQ auf eine Datenbank **wann immer möglich** (und das ist es praktisch immer) aber die erzeugten Navigation Properties verwenden. Am besten sieht man das an einem Beispiel:

```
var query = db.Order_Details
    .Where(x => x.Order.Customer.CustomerID == "BOTTM" && x.Product.UnitPrice > 30)
    .Select(x => x.Product.ProductName);
```

5.5.2 Mit Joins

So funktioniert eine Abfrage ohne Verwendung von Navigation Properties, die aus mehreren Tabellen Daten liest. Man müsste dazu so ähnlich wie bei SQL Joins verwenden. Das funktioniert zwar, wir werden das aber **nie so machen**.

Dabei wäre ausnahmsweise die Query Syntax sogar lesbarer:

```
var query3 = from o in db.Orders
              join od in db.OrderDetails on o.OrderId equals od.OrderId
              join p in db.Products on od.ProductId equals p.ProductId
              where o.CustomerId == "BOTTM" && p.UnitPrice > 30
              select p.ProductName;
```

Für Interessierte – mit Lambda-Syntax wäre dies ausnahmsweise umständlicher:

```
var query2 = db.Orders
    .Join(db.OrderDetails,
        x => x.OrderId,
        x => x.OrderId,
        (order, detail) => new { Order = order, Detail = detail })
    .Join(db.Products,
        x => x.Detail.ProductId,
        x => x.ProductId,
        (detail, product) => new { Detail = detail, Product = product })
    .Where(x => x.Detail.Order.CustomerId == "BOTTM" && x.Product.UnitPrice > 30)
    .Select(x => x.Product.ProductName);
```

Das dabei erzeugte SQL-Statement ist dabei praktisch das gleiche.

5.6 SQL-Statement

In Wirklichkeit wird aus dem LINQ-Statement natürlich ein SQL-String erzeugt – etwas anderes versteht eine Datenbank ja nicht. Diesen kann man sich mit **ToQueryString()** anschauen:

```
string sql = query.ToQueryString();
Console.WriteLine(sql);
```

Die Ausgabe für obige Query sieht so aus:

```
SELECT [p].[ProductName]
FROM [Order Details] AS [o]
INNER JOIN [Orders] AS [o0] ON [o].[OrderID] = [o0].[OrderID]
LEFT JOIN [Customers] AS [c] ON [o0].[CustomerID] = [c].[CustomerID]
INNER JOIN [Products] AS [p] ON [o].[ProductID] = [p].[ProductID]
WHERE ([c].[CustomerID] = N'BOTTM') AND ([p].[UnitPrice] > 30.0)
```

Man sieht: die Verwendung von Navigation Properties vereinfacht die Abfrage enorm – es wird aber von EntityFramework in JOINS übersetzt.

5.1 Log SQL

Man kann sich ganz einfach die SQL-Statements ansehen, die tatsächlich an die Datenbank geschickt werden. Dazu übergibt man dem Konstruktor des DbContext ein Options-Objekt. Dieses muss über einen **DbContextOptionsBuilder<>** erzeugt werden. Die wesentliche Property dabei ist **LogTo**, der man eine Action<string> zum Verarbeiten der Logs sowie optional einen **LogLevel** übergibt.

```
var options = new DbContextOptionsBuilder<NorthwindContext>()
    .LogTo(x => Console.WriteLine(x), LogLevel.Information)
    .Options;
var db = new NorthwindContext(options);
```

```
info: 03.11.2021 17:19:30.767 CoreEventId.ContextInitialized[10403] (Microsoft.EntityFrameworkCore.Infrastructure)
Entity Framework Core 5.0.11 initialized 'NorthwindContext' using provider 'Microsoft.EntityFrameworkCore.SqlServ
info: 03.11.2021 17:19:31.144 RelationalEventId.CommandExecuted[20101] (Microsoft.EntityFrameworkCore.Database.Command)
Executed DbCommand (32ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT COUNT(*)
FROM [Categories] AS [c]
info: 03.11.2021 17:19:31.279 RelationalEventId.CommandExecuted[20101] (Microsoft.EntityFrameworkCore.Database.Command)
Executed DbCommand (35ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT [p].[ProductName]
FROM [Order Details] AS [o]
INNER JOIN [Orders] AS [o0] ON [o].[OrderID] = [o0].[OrderID]
LEFT JOIN [Customers] AS [c] ON [o0].[CustomerID] = [c].[CustomerID]
INNER JOIN [Products] AS [p] ON [o].[ProductID] = [p].[ProductID]
WHERE ([c].[CustomerID] = N'BOTTM') AND ([p].[UnitPrice] > 30.0)
```

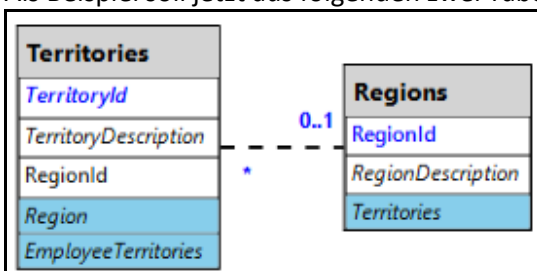
Oder auch Loggen in eine ListBox:

```
var builder = new DbContextOptionsBuilder<NorthwindContext>();
//builder.LogTo(x => Debug.WriteLine(x), LogLevel.Information);
builder.LogTo(x => lstLogs.Items.Add(x), LogLevel.Information);
var db = new NorthwindContext(builder.Options);
```

5.2 Include = Force Join

Mit LINQ erzeugt man keine Datenlisten, sondern SQL Queries. Aufgrund der verzögerten Ausführung wird der im Hintergrund generierte SQL String erst bei Iteration, ToList,... and die Datenbank geschickt. Die Datenbank liefert dann genau das, was im SQL-Statement steht. Das kann zu Problemen führen.

Als Beispiel soll jetzt aus folgenden zwei Tabellen gelesen werden:



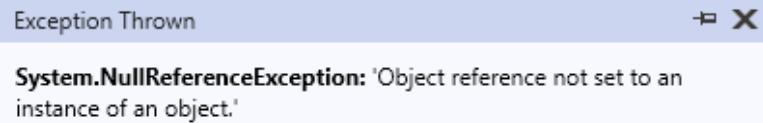
5.2.1 Falsch

Was ist hier falsch?

```
var query = db.Territories;
foreach (var territory in query)
{
    string territoryName = territory.TerritoryDescription;
    string regionName = territory.Region.RegionDescription;
    Debug.WriteLine($"{territoryName} --> {regionName}");
}
```

Man bekommt eine Exception:

```
string territoryName = territory.TerritoryDescription;
string regionName = territory.Region.RegionDescription;
Debug.WriteLine($"
```



Ein Blick auf das SQL statement (`Console.WriteLine(query.ToQueryString())`) sollte den Grund des Problems erkennen lassen.

```
SELECT [t].[TerritoryID], [t].[RegionID], [t].[TerritoryDescription]
FROM [Territories] AS [t]
```

Damit sollte klar sein, dass Region null ist. Warum sollte die Datenbank über einen Join Daten aus einer anderen Tabelle lesen? Die Datenbank kann ja nicht wissen, dass später Informationen aus der Tabelle Region gebraucht werden.

5.2.2 Richtig

Man muss also der Datenbank mitteilen, dass auch Daten aus anderen Tabellen gelesen werden müssen. Man muss im LINQ-Statement anstossen, dass im SQL ein Join erzeugt wird. Das geht über die LINQ-Methode **Include**. Diese gibt an, welche Navigation Property im SQL-String als Join berücksichtigt werden soll.

Man ändert die LINQ-Anweisung als folgendermaßen:

```
var query = db.Territories
    .Include(x => x.Region);
Debug.WriteLine(query.ToQueryString());
foreach (var territory in query)
```

Dadurch wird im erzeugten SQL ein entsprechender Join generiert:

```
SELECT [t].[TerritoryID], [t].[RegionID], [t].[TerritoryDescription], [r].[RegionID], [r].[RegionDescription]
FROM [Territories] AS [t]
INNER JOIN [Region] AS [r] ON [t].[RegionID] = [r].[RegionID]
```

Und der Code funktioniert:

```
Westboro          --> Eastern
Bedford           --> Eastern
Georgetow         --> Eastern
Boston            --> Eastern
```

5.2.3 AutoJoin

Werden Navigation Properties im LINQ verwendet, werden (wie im Eingangsbeispiel) die Joins automatisch erkannt und beim SQL berücksichtigt.

```
var query = db.Territories
    .Where(x => x.Region.RegionDescription=="Western");
Debug.WriteLine(query.ToQueryString());
```

```
SELECT [t].[TerritoryID], [t].[RegionID], [t].[TerritoryDescription]
FROM [Territories] AS [t]
INNER JOIN [Region] AS [r] ON [t].[RegionID] = [r].[RegionID]
WHERE [r].[RegionDescription] = N'Western'
```


5.3 Verzögerte Ausführung

Immer daran denken: **Eine Abfrage wird erst ausgeführt, wenn ToList(), Count(),... aufgerufen oder iteriert wird!**

Und weiters: Nicht alles was in C# möglich (und somit compilierbar) ist, kann in SQL übersetzt werden.

```
var employees = db.Employees
    .Where(x => int.Parse(x.PostalCode) > 90400)
    .OrderBy(x => x.LastName);
// A lot of other code comes here...
// ...
// ...
foreach (var item in employees)
{
    Debug.WriteLine(item);
}
```

Exception Unhandled
System.InvalidOperationException: 'The LINQ expression 'DbSet<Employee>().Where(e => int.Parse(e.PostalCode) > 90400)' could not be translated. Additional information: Translation of method 'int.Parse' failed. If this method can be mapped to your custom

Hinweis: es würde keine Exception geworfen, wenn man die foreach-Schleife auskommentiert.

5.3.1 Clientside evaluation

Einer der C#-Befehle, die nicht in SQL übersetzt werden können, ist die Umwandlung von DateTime in String.

```
var xxx = db.Employees
    .Where(x => $"{x.BirthDate:dd.MM.yyyy}".StartsWith("10"))
    .OrderBy(x => x.LastName)
    .Select(x => $"{x.LastName} {x.FirstName} / {x.BirthDate:dd.MM.yyyy}");
foreach (var item in xxx)
{
    Debug.WriteLine(item);
}
```

Man bekommt also die erwartete Exception:

System.InvalidOperationException: 'The LINQ expression 'DbSet<Employee>().Where(e => string.Format(format: "{0:dd.MM.yyyy}", arg0: (object)e.BirthDate) != null && "10" != null && string.Format(

ABER: kommentiert man den WHERE-Teil aus, funktioniert es überraschenderweise!

```
var xxx = db.Employees
    // .Where(x => $"{x.BirthDate:dd.MM.yyyy}".StartsWith("10"))
    .OrderBy(x => x.LastName)
    .Select(x => $"{x.LastName} {x.FirstName} / {x.BirthDate:dd.MM.yyyy}");
```

Man versteht das, wenn man sich die DB-Logs ansieht (siehe weiter oben):

```
info: 03.11.2021 17:35:11.221 RelationalEventId.CommandExecuted[20101] (Microsoft.EntityFrameworkCore.Database.Command)
      Executed DbCommand (13ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [e].[LastName], [e].[FirstName], [e].[BirthDate]
      FROM [Employees] AS [e]
      ORDER BY [e].[LastName]
```

Es wird also das „Date to String“ nicht ausgeführt, und man erhält trotzdem die richtige Ausgabe:

```
Buchanan Steven / 04.03.1955
Callahan Laura / 09.01.1958
Davolio Nancy / 08.12.1948
Dodsworth Anna / 27.01.1966
```

Was widersprüchlich erscheint, folgt einer genauen Vorgangsweise:

- Wenn ein LINQ-Ausdruck nicht in SQL übersetzt werden kann, wird prinzipiell eine Exception geworfen
- Ausnahme: wenn dieser Ausdruck im abschließendem SELECT vorkommt, wird
 - der LINQ-Ausdruck davor in SQL übersetzt
 - dieser Ausdruck an die Datenbank geschickt (was faktisch einem ToList() entspricht)

- das Ergebnis wird clientseitig (also in C#, nicht SQL) weiterverarbeitet
- dort ist dann das Select kein Problem mehr

Achtung: es gilt aber weiterhin, dass die Query erst bei foreach, ToList(),... an die Datenbank geschickt wird.

Also:

- im abschließendem Select() kann immer jeder compilierbare Code notiert werden
- braucht man derartigen Code schon vorher, muss mit einem zusätzlichen ToList() die Abfrage ausgeführt werden. Man muss sich aber bewusst sein, dass dann aber auch diese unter Umständen sehr vielen Datensätze aus der Datenbank gelesen werden.

5.4 ConnectionString

Momentan steht der Connection string noch im generierten DbContext-File und wäre somit vom Endbenutzer nicht veränderbar. Um das zu beheben schreibt man den Connectionstring in eine Konfigurationsdatei und verwendet diesen dann bei den Options im Konstruktor.

5.4.1 appsettings.json

Die Datei muss appsettings.json heißen und man kann sie z.B. mit Add → Json File erzeugen. Diese Datei muss den Key ConnectionStrings haben, darin kann man dann beliebig viele Connection Strings angeben. Der Connection string selbst unterscheidet sich nicht von jenem, der in NorthwindContext.cs angeführt ist.

```
{
  "ConnectionStrings": {
    "NorthwindDb": "Server=(LocalDB)\\mssqllocaldb;attachdbfilename=C:\\_PR\\CSharp\\PR
  }
}
```

Nicht vergessen für die Datei in den Properties „Copy if Newer“ anzugeben!

5.4.2 Nuget

Zur Verwendung braucht man ein zusätzliches Paket, nämlich

Microsoft.Extensions.Configuration.Json. Dieses daher mit Nuget (oder auch dem Package Manager installieren). Darin befindet sich der unten benötigte ConfigurationBuilder

5.4.3 ConfigurationBuilder

Diese Konfiguration kann man dann mit dem erwähnten **ConfigurationBuilder** auslesen. Der wesentliche Punkt ist dann, dass der ConnectionString mit **builder.UseSqlServer()** gesetzt wird.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    var config = new ConfigurationBuilder()
        .SetBasePath(AppContext.BaseDirectory)
        .AddJsonFile("appsettings.json")
        .Build();
    string connectionnString = config.GetConnectionString("NorthwindDb");

    var options = new DbContextOptionsBuilder<NorthwindContext>()
        .LogTo(x =>
        {
            Console.WriteLine(x);
            lstLogs.Items.Add(x);
        }, LogLevel.Information)
        .UseSqlServer(connectionnString)
        .Options;
    var db = new NorthwindContext(options);
}
```

Der Name „NorthwindDb“ entspricht dabei dem Eintrag in appsettings.json.

Hinweis: Wie der name **UseSqlServer** vermuten lässt könnte man auch auf andere Datenbanksysteme wie z.B. Sqlite zugreifen.

5.4.4 Relativer Pfad - |DataDirectory|

Etwas lästig ist noch, dass der Pfad der Datenbank als absoluter Pfad eingetragen ist. Besser wäre, einen relativen Pfad mit **|DataDirectory|** angeben zu können.

```
{
  "ConnectionStrings": {
    "NorthwindDb": "Server=(LocalDB)\\mssqllocaldb;attachdbfilename=|DataDirectory|Northwnd.mdf;...",
    "NorthwindDb_": "Server=(LocalDB)\\mssqllocaldb;attachdbfilename=C:\\_PR\\CSharp\\PR4\\191_Ef..."
  }
}
```

Auch hier darf man dann nicht vergessen, dass man für Northwnd.mdf „Copy if newer“ angibt.

5.4.5 Sqlite

Um Sqlite verwenden zu können, muss das entsprechende Paket installiert werden:

Microsoft.EntityFrameworkCore.Sqlite

Dann muss die Datenbankdatei in appsettings.json angegeben werden:

```
{
  "ConnectionStrings": {
    "NorthwindDb": "Server=(LocalDB)\\mssqllocaldb; attachdbfilename=C:\\_PR\\CSharp\\PR4\\EfDbFirstWpfDemo\\No...",
    "NorthwindDbSqlite": "data source=C:\\_PR\\CSharp\\PR4\\EfDbFirstWpfDemo\\NorthwindDbLib\\Northwnd.sqlite"
  }
}
```

Dieser Connectionstring muss dann anstelle von UseSqlServer mit UseSqlite verwendet werden:

```
var config = new ConfigurationBuilder()
    .SetBasePath(AppContext.BaseDirectory)
    .AddJsonFile("appsettings.json")
    .Build();
//string connectionnString = config.GetConnectionString("NorthwindDb");
string connectionnString = config.GetConnectionString("NorthwindDbSqlite");

var options = new DbContextOptionsBuilder<NorthwindContext>()
    .LogTo(x =>
    {
        Console.WriteLine(x);
        lstLogs.Items.Add(x);
    }, LogLevel.Information)
    //UseSqlServer(connectionnString)
    .UseSqlite(connectionnString)
    .Options;
var db = new NorthwindContext(options);
```