AUSGABE 01/2019

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

André Minhorst Verlag

Impressum

DATENBANKENTWICKLER
© 2019 André Minhorst Verlag
Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

PowerApps: Kamera steuern

PowerApps sind die Lösung von Microsoft, um schnell Anwendungen für Smartphones und Co zu realisieren. Und da es kein Smartphone ohne Kamera gibt, stellt sich natürlich die Frage, ob wir PowerApps auch so programmieren können, dass wir damit Fotos aufnehmen und diese weiterverarbeiten können – etwa, indem wir diese in einer Datenbank oder in der Cloud speichern oder sogar in den Bildern des Smartphones. Der vorliegende Artikel zeigt, wie Sie eine Funktion zum Fotografieren zu Ihrer PowerApp hinzufügen und was Sie mit den aufgenommenen Bildern alles tun können.

Neue PowerApp anlegen

Die Funktion zum Aufnehmen von Fotos und zum Weiterverarbeiten der Aufnahmen wollen wir in einer neuen PowerApp ausprobieren, daher legen Sie unter **web.powerapps.com** nach der Anmeldung mit Ihren Kontodaten mit einem Klick auf Erstellen und der nachfolgenden Auswahl von **Canvas-App ohne Vorlage** eine neue App an (siehe Bild 1).

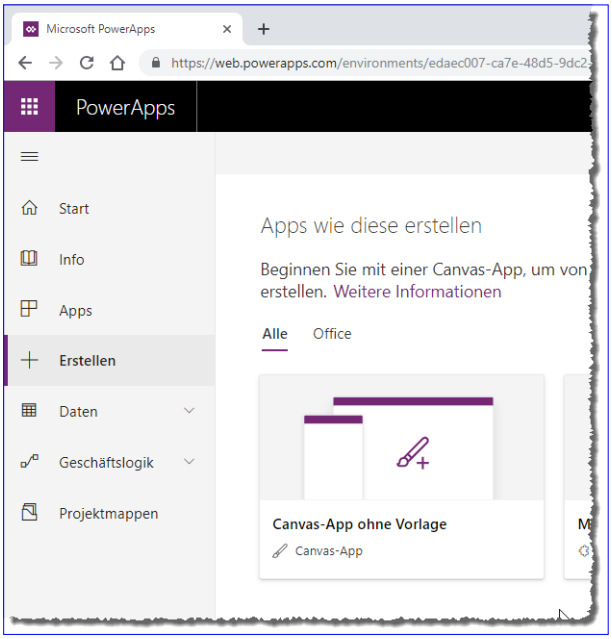


Bild 1: Anlegen der neuen PowerApp

Im folgenden Dialog geben wir als Namen für die App schlicht **Fotos** an und legen fest, dass wir eine App für das Telefon erstellen wollen (siehe Bild 2).

Kamera-Element hinzufügen

Nun empfängt uns die App mit einer neuen, leeren Seite. Hier wollen wir als Erstes ein **Kamera**-Element einfügen. Dazu wählen Sie ganz oben Einfügen aus und aus der nun erscheinenden unteren Menüleiste den Eintrag **MedienlKamera** (siehe Bild 3).

Diese erscheint dann als schwarzer Bereich mit dem Text **Ihre Kamera ist nicht eingerichtet, oder Sie verwenden sie bereits**. Diese Meldung erscheint beispielsweise dann, wenn Sie an einem Rechner mit einem Monitor arbeiten, an dem keine Kamera zur Verfügung steht. Das macht das Testen natürlich etwas komplizierter, aber Sie können dann auch direkt am Smartphone testen. Dazu speichern Sie die Anwendung einfach mit dem Menübefehl **DateilSpeichern** und betätigen danach

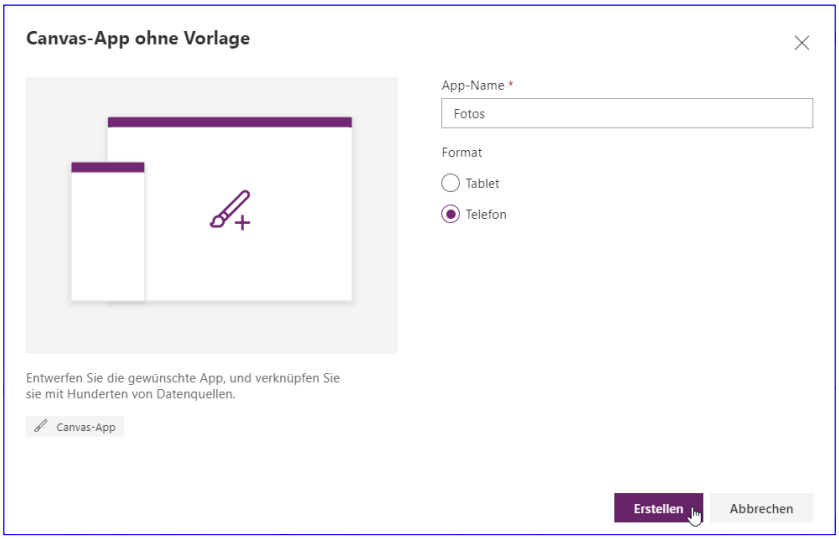


Bild 2: Einstellen der Basiseigenschaften

gegebenenfalls noch die **Veröffentlichen**-Schaltfläche.

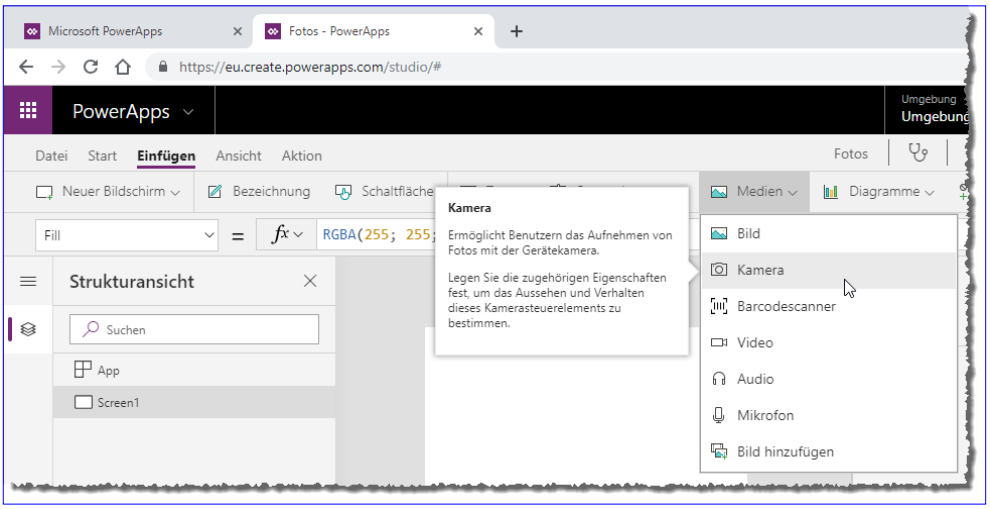


Bild 3: Hinzufügen des Kamera-Elements

Nachdem wir die App dann mangels Smartphone in der PowerApps-App gestartet haben, erscheint immerhin ein Bildschirm, der die Vorschau der Kamera anzeigt. Allerdings finden wir hier keinerlei weitere Bedienelemente wie etwa einen Auslöser oder zum Zoomen. Und auch das Zoomen durch größer- oder kleiner ziehen des Motivs funktioniert nicht. Also benötigen wir wohl noch weitere Elemente, um unsere erste Aufnahme zu schießen.

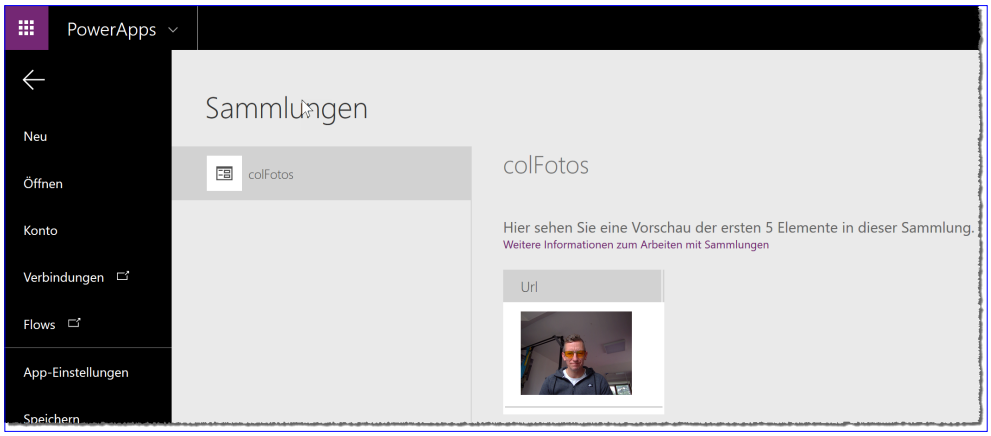


Bild 4: Anzeige eines Bildes aus einer Collection

Testen mit Kamera

Vorab der Hinweis, dass das Entwickeln einer PowerApp mit Kamerafunktion auf einem Rechner ohne angeschlossene Kamera keinen Spaß macht. Sie können die vielen Features, die zur Entwurfszeit zur Verfügung stehen, schlicht nicht nutzen – unter anderem die direkte Anzeige der geschossenen Fotos in der zum Ablegen verwendeten Collections.

Auslösen per Klick auf das Bild

Die einfachste Möglichkeit, ein Foto aufzunehmen, ist ein Klick direkt auf das Kamera-Steuerelement. Dazu fügen Sie der Eigenschaft **OnSelect** des Kamera-Steuerelements im Bereich **Erweitert** der Eigenschaften den folgenden Wert hinzu:

```
ClearCollect(colFotos;Camera1.Photo)
```

Dabei ist **colFotos** eine Sammlung, die direkt nach dem Aufnehmen des ersten Fotos erstellt wird und **Camera1** ist der Name unseres Kamera-Steuerelements. Während des Entwurfs können Sie die Sammlung einsehen, indem Sie oben im Menü den Eintrag **Ansicht** auswählen und dann auf **Sammlungen** klicken. Nach dem Aufnehmen eines ersten Fotos erscheint dieses dann in der Sammlung (siehe Bild 4). Allerdings finden wir dort immer nur das zuletzt aufgenommene Foto vor. Wie können wir mehrere Fotos anzeigen beziehungsweise diese überhaupt erst der Sammlung hinzufügen? Wir haben schlicht die falsche Me-

thode verwendet. **ClearCollect** leert die Sammlung und fügt dann das aktuelle Foto ein. Mit der folgenden Anweisung fügen wir weitere Bilder zur Sammlung hinzu:

```
Collect(colFotos;Camera1.Photo)
```

Wenn wir nun ein paar Fotos schießen, erscheinen diese auch in der Sammlung.

ClearCollect hat dennoch seinen Sinn – Sie können damit immer genau ein Bild aufnehmen und in der Sammlung speichern und es dann wie gewünscht weiterverarbeiten, indem Sie es etwa an einem anderen Ort speichern.

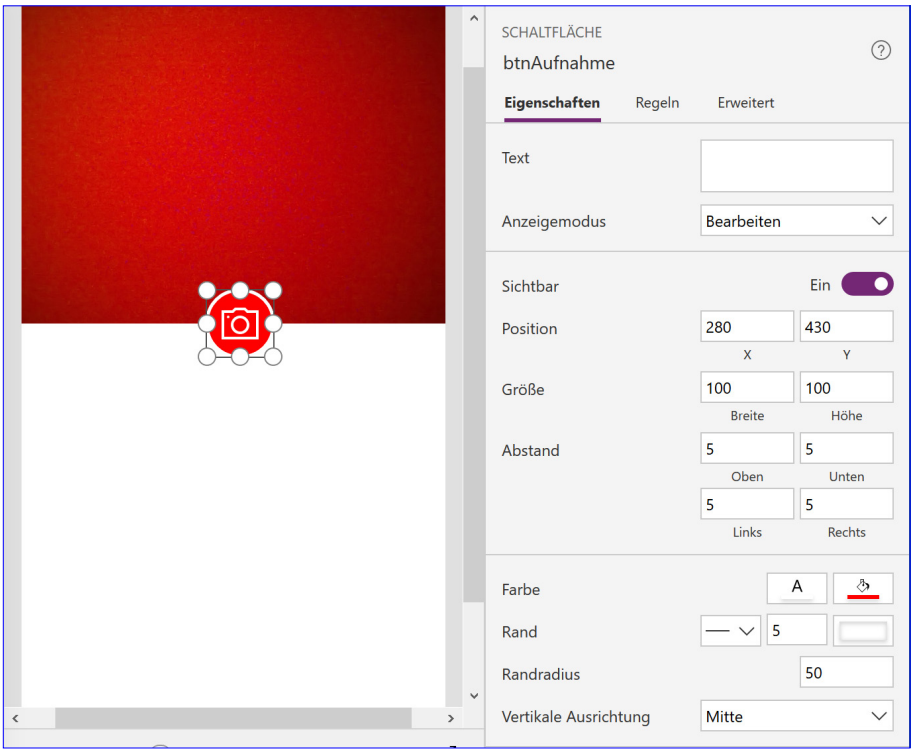


Bild 5: Neue Schaltfläche zum Aufnehmen von Fotos

Fotos per Button

Wir wollen das Aufnehmen von Fotos etwas intuitiver gestalten – zum Beispiel durch einen Button, den wir ähnlich wie bei anderen Apps auf Smartphones im unteren Bereich der Vorschau platzieren. Der Button soll rund sein und die Füllfarbe **Rot** sowie die Rahmenfarbe **Weiß** erhalten. Außerdem fügen wir ihm noch das Symbol einer Kamera hinzu. Um den Button rund zu machen, stellen wir die Größe auf **100x100** sowie den **Randradius** auf **50** ein. Danach sehen das Kamera-Steuerelement und der Button wie in Bild 5 aus. Der Schaltfläche **btnAufnahme** weisen wir für die Eigenschaft **OnSelect** den gleichen Ausdruck zu wie weiter oben dem Kamera-Steuerelement:

```
Collect(colFotos;Camera1.Photo)
```

Interessanterweise funktioniert es so allerdings nicht. Wir müssen in diesem Fall einen kleinen Workaround anwenden:

- Stellen Sie die Eigenschaft **StreamRate** im Bereich **Erweitert** der Eigenschaften des Kamera-Steuerelements auf **100** ein.
- Legen Sie für die **OnSelect**-Eigenschaft der Schaltfläche des Buttons die folgende Funktion fest: **Collect(colFotos;Camera1.Stream)**

Damit lösen Sie die Kamera über die Schaltfläche aus.

PowerApps: Bilder in Datenbank speichern

Im Artikel »PowerApps: Kamera steuern« zeigen wir, wie Sie das Kamera-Steuerelement in PowerApps nutzen, um Fotos aufzunehmen und diese entweder direkt in einem Image-Steuerelement anzuzeigen oder in einer Sammlung abzulegen, deren Elemente dann in einem Katalog-Steuerelement angezeigt werden können. Beide Methoden sind nicht als dauerhafter Speicherort zu verstehen, sodass wir uns eine Alternative überlegen müssen. Für Datenbank-Entwickler liegt es nahe, die erfassten Bilddateien in einer SQL Server-Datenbank zu speichern. Wie das gelingt, zeigt der vorliegende Beitrag.

Fotos speichern

Wenn wir mit den Lösungen aus dem Artikel **PowerApps: Kamera steuern** Fotos aufnehmen, landen diese Fotos zwar in einem Bildsteuerelement oder in einer Sammlung, allerdings nur temporär: Wenn wir die PowerApp speichern und veröffentlichen und diese in der PowerApps-App auf unserem Smartphone testen, finden wir die Bilder nach einem Neustart der App nicht mehr in der Liste vor. Also benötigen wir eine alternative Möglichkeit zum Speichern der Bilder.

Es gibt die Möglichkeit, Bilder unter SharePoint zu speichern, in OneDrive oder auch in einer SQL Server-Datenbank. Da wir uns hier mit Datenbanken beschäftigen, wollen wir die Bilder in einer SQL Server-Datenbank speichern.

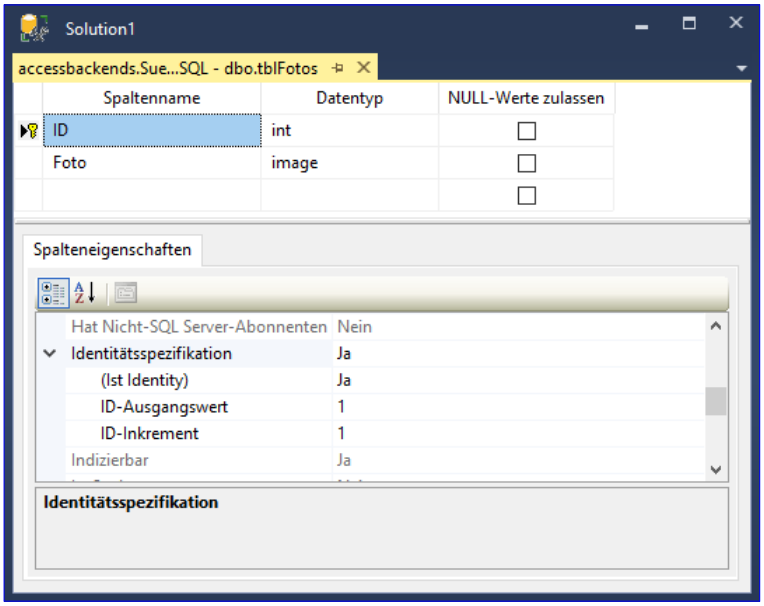


Bild 1: Anlegen der neuen PowerApp

Dazu benötigen wir zunächst eine Beispieldatenbank mit einer Tabelle, die ein Feld des Typs **image** enthält. Außerdem muss die Datenbank natürlich über das Internet verfügbar sein, damit wir auch vom Smartphone aus darauf zugreifen können. Der Einfachheit halber fügen wir unserer Beispieldatenbank **Südsturm**, die auf einem Azure-Server liegt, einfach eine Tabelle namens **tblFotos** hinzu. Diese Tabelle soll neben dem Primärschlüsselfeld **ID** noch ein Feld namens **Foto** erhalten (siehe Bild 1). Die Tabelle erstellen wir beispielsweise im SQL Server Management Studio, mit dem wir uns leicht mit dem Azure-Datenbankserver verbinden können. Wenn Sie noch keine Azure-Datenbank angelegt haben, finden Sie im Artikel **SQL Server-Datenbank ins Web mit SQL Azure** aus Ausgabe 6/2018 alle notwendigen Informationen.

Datenquelle verknüpfen

Nachdem Sie die SQL Server-Datenbank mit der Tabelle **tblFotos** erstellt haben, legen wir eine Verknüpfung zu dieser Tabelle als Datenquelle in der PowerApp an. Dazu wählen Sie im oberen Menü den Eintrag **Ansicht** und im unteren Menü den Eintrag **Datenquellen** aus. Daraufhin erscheint der Bereich **Daten**, der uns den Eintrag **Datenquelle hinzufügen** anbietet (siehe Bild 2).

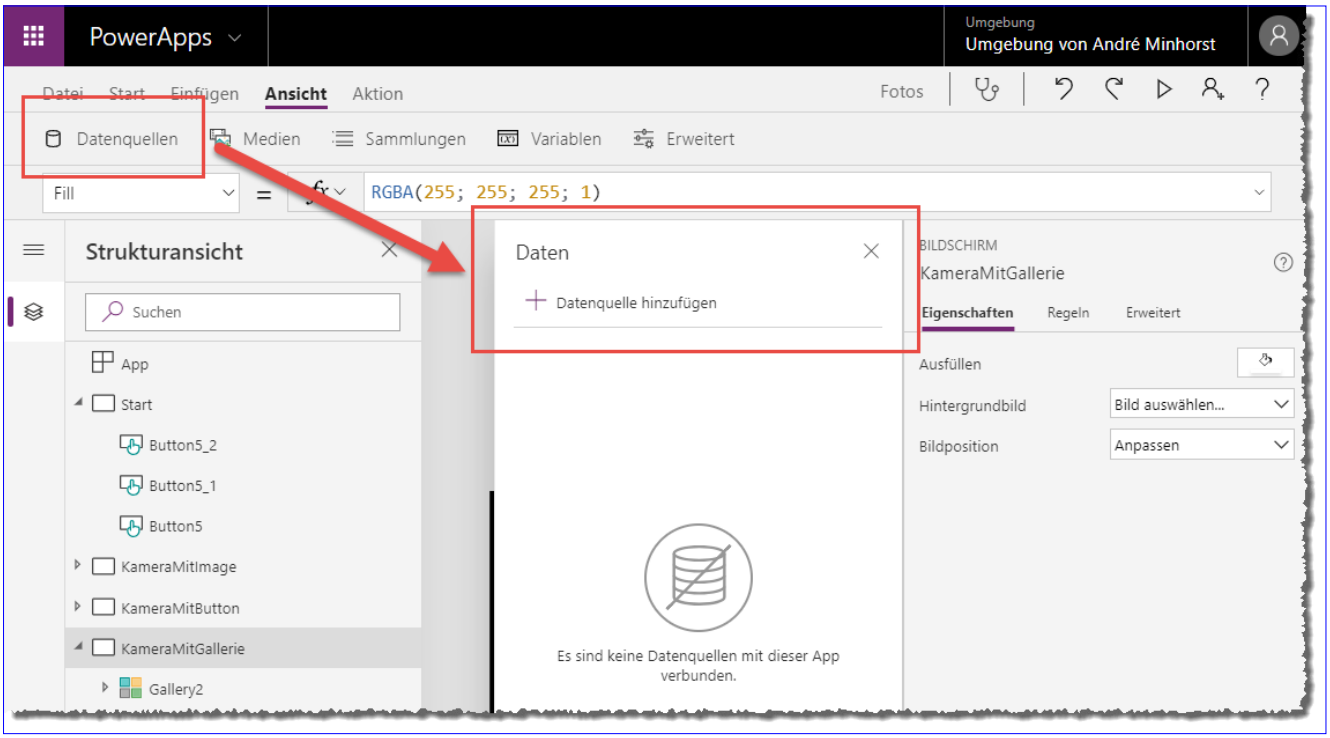


Bild 2: Hinzufügen einer neuen Datenquelle

Die Liste im Bereich **Daten** zeigt nun alle bereits zuvor verwendeten Datenquellen an. Finden Sie die gewünschte Verbindung hier nicht vor, klicken Sie auf **Neue Verbindung**. Es erscheint eine Liste aller möglichen Verbindungen, von denen Sie den Eintrag **SQL Server** auswählen. Nach dem Anklicken dieses Eintrags erscheint ein weiterer Bereich, der Sie zur Eingabe der Verknüpfungsdaten wie **Name des SQL Servers**, **Name der SQL-Datenbank**, **Benutzername** und **Kennwort** auffordert.

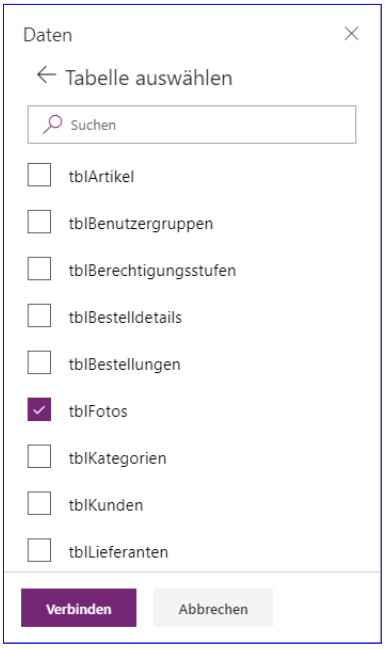


Bild 3: Auswählen der Tabelle zum Speichern der Fotos

Nach der korrekten Eingabe dieser erscheint eine Liste der Tabellen der SQL Server-Datenbank. Hier wählen Sie die Tabelle **tblFotos** aus und betätigen die Schaltfläche **Verbinden** (siehe Bild 3). Die so verknüpfte Tabelle steht dann als Datenquelle bereit.

Bildschirm mit Kamera erstellen

Danach erstellen wir einen neuen Bildschirm mit einem Kamera-Steuerelement namens **cam1** im oberen Bereich. Darunter fügen wir zwei Schaltflächen namens **btnFotoSpeichern** und **btnZurueck** ein. Ganz unten legen wir ein Bildsteuerelement namens **imgFotoDb** an (siehe Bild 4).

Wir belassen es in diesem Beispiel bei der Möglichkeit, Fotos durch Anklicken oder Antippen des Kamera-Steuerelements aufzunehmen und die Fotos dann im Bildsteuerelement **imgFotoDb** anzuzeigen, indem wir dessen Eigenschaft **Image** auf den Wert **cam1.Photo** einstellen.

PowerApps: Bilder aus der Datenbank anzeigen

Im Artikel »PowerApps: Bilder in der Datenbank speichern« zeigen wir, wie Sie Fotos, die Sie mit dem Kamera-Steuerelement einer PowerApp aufgenommen haben, in der Tabelle einer Datenbank speichern. Im vorliegenden Artikel fügen wir der PowerApp eine weitere Bildschirmseite hinzu, auf der wir die Bilder dieser Datenbank anzeigen und verwalten.

Wir setzen auf der Anwendung aus dem Artikel **PowerApps: Bilder in der Datenbank speichern** auf, da wir die dortige Tabelle und die darin über die PowerApp gespeicherten Bilder nutzen wollen.

Der PowerApp aus dieser Anwendung fügen wir eine neue Bildschirmseite hinzu, der wir den Namen Fotos aus der Datenbank geben. Wir verwenden wieder die Vorlage für eine leere Bildschirmseite.

Das Herstellen der Verbindung mit der Datenquelle haben wir bereits im oben genannten Artikel beschrieben, sodass wir nun direkt auf die Tabelle **dbo.tblFotos** als Datenquelle zugreifen können.

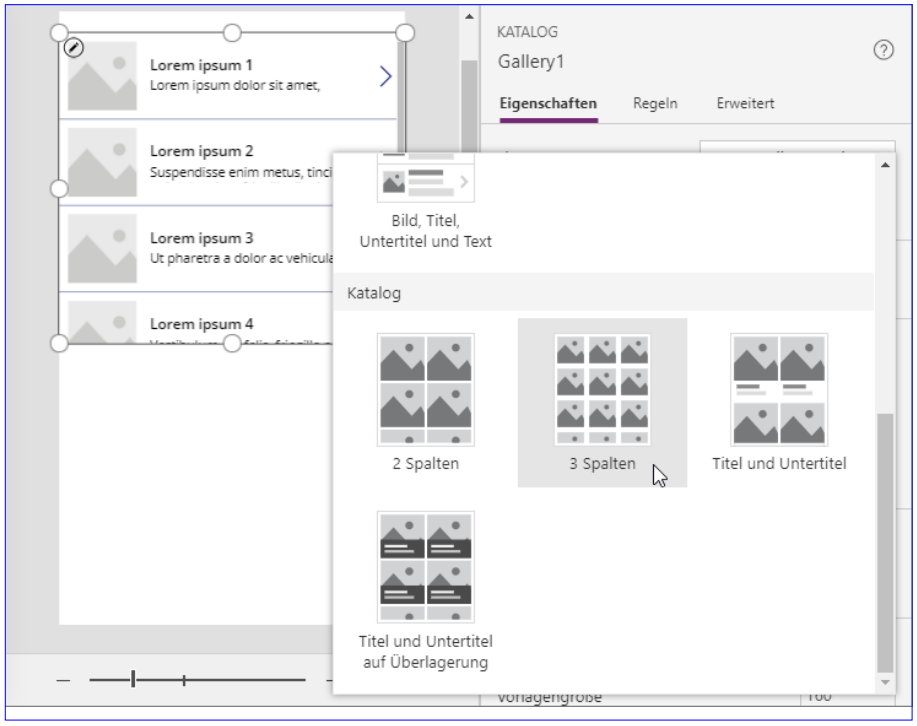


Bild 1: Einstellen des Formats des Katalogs zur Bildanzeige

Anschließend fügen wir der Bildschirmseite einen neuen Katalog hinzu, indem Sie im oberen Menü den Eintrag **Einfügen** und im unteren Menü den Eintrag **KataloglVertikal** auswählen. Dieses Format wollen wir nicht beibehalten, deshalb öffnen wir die Auswahlliste für die Eigenschaft **Layout** im Bereich **Eigenschaften** und wählen dort den Eintrag **3 Spalten** aus (siehe Bild 1).

Im gleichen Bereich wählen wir nun die Tabelle **[dbo].[tblFotos]** für die Eigenschaft **Elemente** aus (siehe Bild 2).

Danach erscheinen die in der Datenbank gespeicherten Bilder bereits in der Katalog-Ansicht (siehe Bild 3).

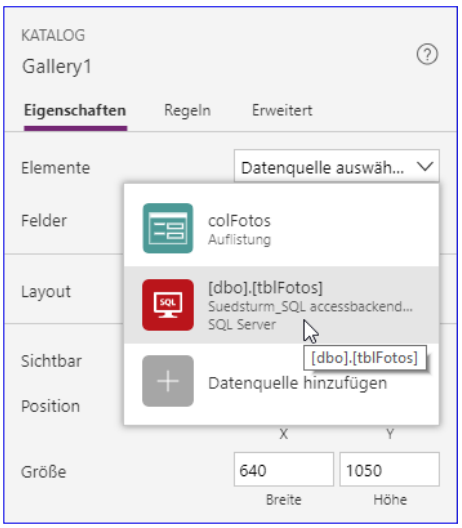


Bild 2: Einstellen des der Datenquelle für den Katalog zur Bildanzeige

Bild im Detail anzeigen

Nun fügen wir eine weitere Bildschirmseite hinzu, welche eines der Bilder nach dem Anklicken in der vollen Ansicht anzeigt. Diese neue Seite nennen wir **FotoAusDerDatenbankDetail**.

Der Seite fügen wir im oberen Bereich eine Schaltfläche namens **btnZurueck** hinzu. Diese statten wir über die Eigenschaft **OnSelect** mit dem Befehl **Back()** aus. Mit dieser Schaltfläche wollen wir nach dem Anzeigen des Bildes in der Detailansicht wieder zur Übersicht zurückkehren.

Dann fügen wir der Bildschirmseite über den Menübefehl **FormularelAnzeigen** ein Formular hinzu, das später ein Bildsteuerelement zum Anzeigen des gewünschten Bildes aufnehmen soll. Nachdem wir das Formular an die Größe des Bildschirms angepasst haben, sieht dieser wie in Bild 4 aus.

Nun legen wir die Datenquelle für das Formular fest. Sie könnten nun auf den Link **Mit Daten verbinden** klicken, erneut die SQL Server-Datenbank auswählen und die Tabelle **tblFotos** festlegen. Dadurch würde aber eine neue Verknüpfung namens **[dbo].[tblFotos_1]** in der App angelegt, was unnötig ist. Also geben Sie schlicht **'[dbo].[tblFotos]'** für die

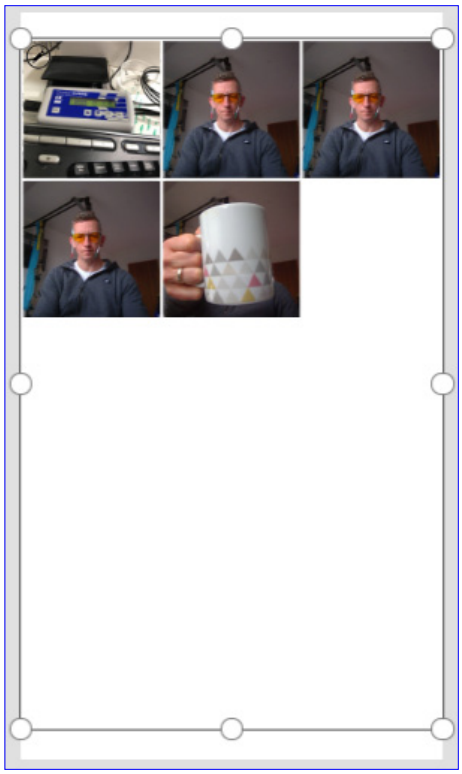


Bild 3: Anzeige von Bildern aus der Datenbank im Katalog-Steuerelement

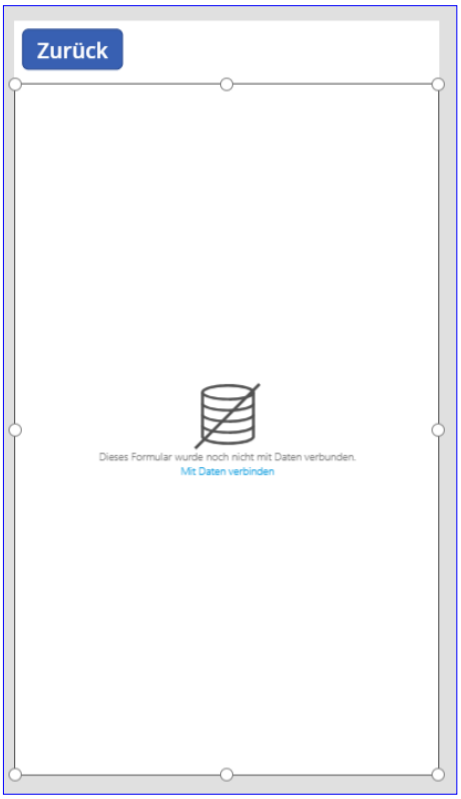


Bild 4: Detailformular für die Anzeige der Fotos

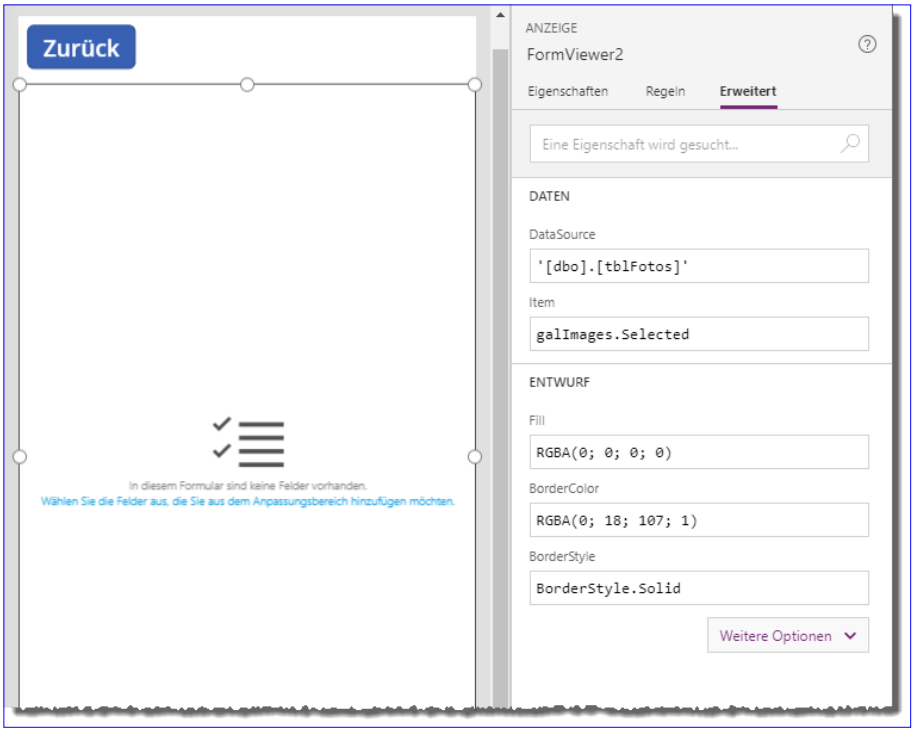


Bild 5: Einstellen der Eigenschaften für die Datenquelle

Von Access zu Entity Framework: Update 1

In Ausgabe 5/2018 haben wir in zwei Artikeln gezeigt, wie Sie das Datenmodell und die Daten einer Access-Datenbank in ein Entity Data Model und darüber in eine SQL Server-Datenbank migrieren. Im vorliegenden Artikel finden Sie eine Optimierung der dort beschriebenen Prozeduren. Im Detail geht es darum, dass in Access-Tabellen manchmal Namen in Tabellen verwendet werden, die gleichzeitig Plural und Singular der enthaltenen Entität sind – wie zum Beispiel bei tblArtikel. Das führt bei unserer automatisierten Migration früher oder später zu Problemen. Daher erweitern wir unsere Prozeduren um die notwendigen Unterscheidungen.

Im Artikel **Access zu WPF: Detailformulare mit Textfeldern** aus der aktuellen Ausgabe beispielsweise erstellen wir das Code behind-Modul für eine WPF-Detailansicht der Einträge der Tabelle **tblKunden** beziehungsweise der Collection **Kunden** und der Klasse **Kunde**. Hier gelingt alles reibungslos, da wir beim Migrieren der Tabellen der Access-Tabelle in die **DbSet**-Definition und die Klassendefinition mit **Kunde** und **Kunden** entsprechende unterschiedliche Bezeichnungen verwenden konnten. Wenn wir die dort erarbeiteten Prozeduren nutzen wollen, um eine Detailansicht auf Basis eines Formulars zu erstellen, dass die Tabelle **tblArtikel** als Datensatzquelle verwendet, gelingt das nicht problemlos, da beim Migrieren des Datenmodells in das Data Entity Model aus **tblArtikel** sowohl das **DbSet**-Element **Artikel** als auch die Klasse **Artikel** generiert wurden. Das führt noch nicht zu Fehlern, aber wenn wir dann mit der Prozedur **FormularNachWPF_CodeBehind** die Code behind-Klasse erzeugen und dort einfügen, erhalten wir eine Reihe von Fehlermeldungen (siehe Bild 1).

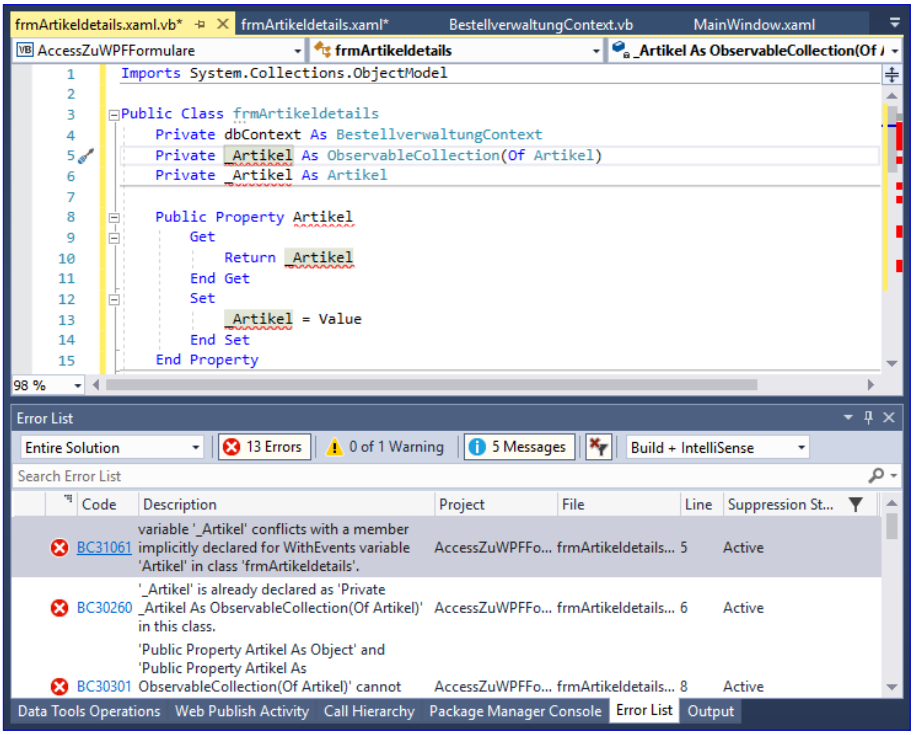


Bild 1: Fehler bei Verwendung gleicher Variablennamen für Objekte und Collections

Wie können wir das Problem lösen? Eine Möglichkeit wäre, die Bezeichnungen im Code etwa der Code behind-Klasse so anzupassen, dass die verschiedenen mehrfach vorkommenden Variablen und Eigenschaftsbezeichnungen eindeutige Namen erhalten. Noch schöner wäre es allerdings, wenn wir direkt, wie auch beim Beispiel der Tabelle **tblKunden**, eindeutige Namen für die Klasse und die **DbSet**-Elemente hätten – in diesem Fall auch der besseren Lesbarkeit halber als Singular und Plural.

Erstellung des Entity Data Models anpassen

Und hier wollen wir ansetzen, indem wir die Prozedur **EDMErstellen** des Moduls **mdlEDM** aus dem Artikel **Von Access zu Entity Framework: Datenmodell** entsprechend anpassen. Das sieht dann ausschnittsweise wie folgt aus:

```
Public Sub EDMErstellen()
    ...
    For Each tdf In db.TableDefs
        If Not Left(tdf.Name, 4) = "MSys" Then
            strPK = ""
            If Not GetPrimaryKey(tdf.Name, strPK) Then
               ...
            Else
                strEntity = Replace(strPK, "ID", "")
                strEntities = Replace(tdf.Name, "tbl", "")
                If strEntity = strEntities Then
                    Do While strEntity = strEntities
                        MsgBox "Plural und Singular der Bezeichnung der Entitäten (hier """ & strEntity _
                            & """) im Tabellennamen scheinen gleich zu sein." & vbCrLf & vbCrLf & "Geben Sie " _
                            & "nachfolgend eine Bezeichnung für den Singular der Entität ein und eine für den Plural."
                        strEntity = InputBox("Bezeichnung für den Singular/den Klassennamen der Entität """ _
                            & strEntity & """:", "Singular bestimmen", strEntity)
                        strEntities = InputBox("Bezeichnung für den Plural/den Namen der Auflistung der Entität """ _
                            & strEntities & """:", "Plural bestimmen", strEntities)
                    Loop
                End If
                ...
            End If
        End If
    Next tdf
    ...
End Sub
```

Nachdem wir die Bezeichnungen für die einzelne Entität und die Mehrzahl ermittelt und in **strEntity** und **strEntities** gespeichert haben, vergleichen wir diese in einer **If... Then**-Bedingung. Sind beide gleich, steigen wir in eine **Do While**-Schleifen ein. Diese zeigt zunächst eine Meldung wie in Bild 2 an.

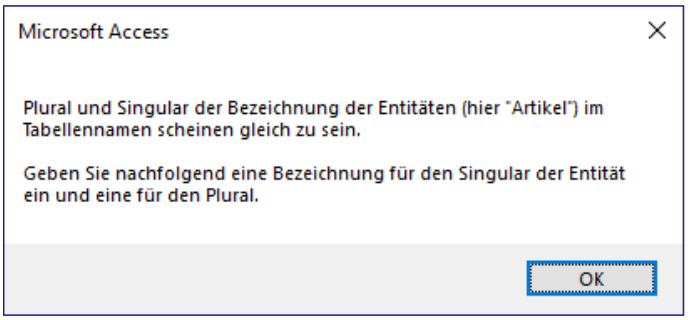


Bild 2: Meldung bei Gleichheit von Singular und Plural

Nach dieser Erklärung folgen zwei **InputBox**-Anweisungen, mit denen die Prozedur die Singular- und die

Plural-Version für die zu erstellenden Klassen und **DbSet**-Elemente abfragt. Sind die beiden Werte von **strEntity** und **strEntities** nach dem Durchlauf der **Do While**-Schleife nicht mehr gleich, wird diese verlassen und die Prozedur erstellt die Klasse und die **DbSet**-Definition mit den neuen Namen.

Wenn wir die Prozedur nun aufrufen und die erstellten und in die Zwischenablage kopierten Code-Strukturen in das Projekt einfügen, finden wir allerdings noch einen kleinen Syntaxfehler vor: Das Fremdschlüsselfeld **ArtikelID** der Tabelle **tblArtikel** wird noch nicht entsprechend der neuen Benennung der **Artikel**-Entitäten umbenannt. Das Feld sollte nun **ProduktID** statt **ArtikelID** heißen und die Eigenschaft, die das **Artikel**-Objekt aufnimmt, sollte das **Produkt**-Objekt aufnehmen (siehe Bild 3).

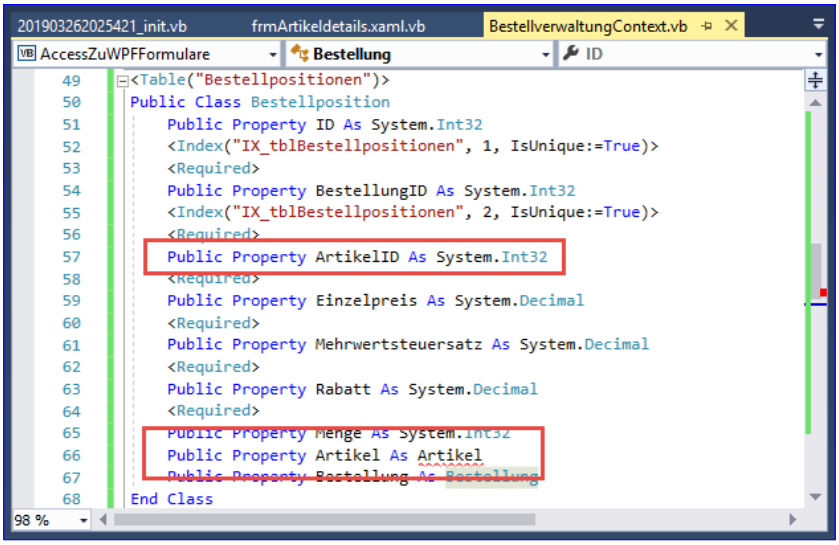


Bild 3: Artikel muss noch durch Produkt ersetzt werden.

Da wir in der Prozedur alle Tabellen in willkürlicher Reihenfolge durchlaufen, kann es sein, dass wir die Tabelle **tblArtikel** erst nach der Tabelle **tblBestelldetails** durchlaufen. Wir erfahren aber erst beim Durchlaufen der Tabelle **tblArtikel**, dass hier eine Umbenennung notwendig ist, die sich auch schon auf Felder der Tabelle **tblBestelldetails** auswirkt. Wir müssen also eine Schleife einbauen, die schon vorher prüft, ob Ersetzungen bei Tabellennamen notwendig sind, weil dieser gleichzeitig Plural und Singular ist und dann später beim zweiten Durchlauf und beim Zusammenstellen des Codes für die Entitäten und **DbSet**-Elemente auf die hier gewonnenen Informationen zurückgreifen.

Dazu sind einige umfangreichere Umstellungen nötig. Diese beruhen im Wesentlichen darauf, dass wir die **TableDef**-Objekte nicht mehr einfach in einer Schleife durchlaufen und dabei die Definitionen aller Entitäten zusammenstellen können – aus dem obigen Grund müssen wir erst einmal alle Tabellen dahingehend analysieren, ob sie Entitätsnamen hervorbringen würden, die gleichzeitig Plural und Singular sind. Also durchlaufen wir zunächst in einer Schleife alle **TableDef**-Objekte und tragen die relevanten Informationen jeweils in eine Klasse ein, die entsprechende Eigenschaften aufnimmt. Die neue Klasse legen Sie im VBA-Editor mit dem Menübefehl **Einfügen|Klassenmodul** an und speichern diese unter dem Namen **clsMapping**. Die folgenden privaten Variablen speichern die Werte:

```
Private m_Tabelle As String
Private m_PK As String
Private m_Entity As String
Private m_Entities As String
Private m_ID As String
Private m_Entity_Original As String
Private m_Entities_Original As String
```

Für jeden dieser Werte gibt es je eine **Property Get**- und eine **Property Let**-Methode, die wie folgt aussehen – hier stellvertretend für die Eigenschaft **Tabelle**:

```
Public Property Get Tabelle() As String
    Tabelle = m_Tabelle
End Property

Public Property Let Tabelle(str As String)
    m_Tabelle = str
End Property
```

Wir erstellen für jedes **TableDef**-Objekt eine dieser Klassen und speichern diese in einer **Collection** namens **colMappings**, die wir wie folgt deklarieren und mit dem Schlüsselwort **New** erstellen:

```
Public Sub EDMErstellen()
    Dim db As DAO.Database, tdf As DAO.TableDef, fld As DAO.Field
    Dim strDbSets As String, strPK As String, strDatatype As String
    Dim bolArray As Boolean, bolDatatypeExists As Boolean
    Dim strArray As String, strFieldname As String, strUntertyp As String
    Dim strIndex As String, strCode As String, colMappings As Collection
    Dim objMapping As clsMapping
    Dim objMappingFK As clsMapping
    Set db = CurrentDb
    Set colMappings = New Collection
```

Dann beginnen wir, die **TableDefs** zu durchlaufen. Dabei prüfen wir wieder, ob der Tabellenname nicht mit **MSys** oder **~** beginnt. Dann prüfen wir, ob die Tabelle einen Primärschlüssel aufweist. Falls nicht, landet ein entsprechender Kommentar in unserem in **strCode** zusammengestellten Code für das VB-Projekt:

```
    For Each tdf In db.TableDefs
        If Not Left(tdf.Name, 4) = "MSys" And Not Left(tdf.Name, 1) = "~" Then
            If Not GetPrimaryKey(tdf.Name, strPK) Then
                strCode = strCode & "    '****" & vbCrLf
                strCode = strCode & "    'Mehrere PKs in " & tdf.Name & ". Die Klasse wird nicht erstellt." & vbCrLf
```

Falls die Tabelle ein Primärschlüsselfeld besitzt, tragen wir einige Werte in die Eigenschaften des Objekts **objMapping** ein, das wir dazu neu erstellen. **Entity** wird mit dem Primärschlüsselwert ohne **ID** gefüllt, **Entity_Original** mit dem gleichen Wert. Gleiches gilt für **Entities** und **Entities_Original**, die mit dem Tabellennamen ohne **tbl** gefüllt werden. Sind **.Entity** und **.Entities** gleich, durchlaufen wir die schon oben beschriebene Schleife, in welcher der Benutzer solange neue Werte für **.Entity** und **.Entities** eingibt, bis sich beide unterschieden (etwa **Produkt** und **Produkte** statt **Artikel**). Ist das erledigt, schreiben wir noch **ID** in die Eigenschaft **.ID**, den Primärschüsselnamen in **.PK** und den Tabellennamen in **.Tabelle**:

```
        Else
            Set objMapping = New clsMapping
            With objMapping
                .Entity = Replace(strPK, "ID", "")
                .Entity_Original = .Entity
                .Entities = Replace(tdf.Name, "tbl", "")
                .Entities_Original = .Entities
                If .Entity = .Entities Then
                    Do While .Entity = .Entities
                        MsgBox "Plural und Singular der Bezeichnung der Entitäten (hier """ & .Entity _
                            & """) im Tabellennamen scheinen gleich zu sein." & vbCrLf & vbCrLf & "Geben Sie " _
                            & "nachfolgend eine Bezeichnung für den Singular der Entität ein und eine für den Plural."
                        .Entity = InputBox("Bezeichnung für den Singular/den Klassennamen der Entität """ & .Entity _
                            & """:", "Singular bestimmen", .Entity)
                        .Entities = InputBox("Bezeichnung für den Plural/den Namen der Auflistung der Entität """ _
                            & .Entities & """:", "Plural bestimmen", .Entities)
                    Loop
                End If
                .ID = "ID"
                .PK = strPK
                .Tabelle = tdf.Name
            End With
        End If
```

Schließlich fügen wir das Objekt **objMapping** zu **colMappings** hinzu, damit wir es später nutzen können:

```
            colMappings.Add objMapping
        End If
    Next tdf
```

Und das geschieht recht schnell, denn gleich nach dem Einlesen der relevanten Informationen durchlaufen wir alle Elemente der **colMappings**-Collection in einer **For Each**-Schleife, wobei wir das jeweilige Objekt mit **objMapping** referenzieren:

```
    For Each objMapping In colMappings
```

Innerhalb dieser Schleife ersetzen wir alle Referenzen auf **strEntity**, **strEntities** und so weiter durch Referenzen auf die entsprechenden Elemente des Objekts **objMapping**. Daraus holen wir das **TableDef**-Objekt mit dem Namen aus **.Tabelle**, um dessen Felder über die **Fields**-Auflistung zu durchlaufen. Vorher stellen wir noch die Zeile für das **DbSet**-Element sowie die Kopfzeile der Entität zusammen:

```
        With objMapping
```

Access zu WPF: Detailformulare mit Textfeldern

Wir haben uns bereits in einigen Artikel angesehen, wie Sie Übersichtsformulare, Detailformulare und so weiter unter WPF anlegen. Was aber, wenn Sie keine Lust haben, die Formulare unter WPF alle neu zu programmieren, obwohl Sie das schon unter Access erledigt haben? In diesem Artikel schauen wir uns an, wie die programmgesteuerten Möglichkeiten aussehen, um Formulare automatisch als WPF-Fenster oder -Seiten abzubilden. Es wird ein wenig Handarbeit übrig bleiben, aber einen großen Teil der Schritte können Sie sich damit deutlich erleichtern.

Voraussetzungen

Wir gehen an dieser Stelle davon aus, dass Sie bereits ein Entity Data Model auf Basis des Access-Datenmodells erstellt und eine entsprechende SQL Server-Datenbank auf Basis des Entity Data Modells erstellt haben. Wie das gelingt, zeigen die Artikel **Von Access zu Entity Framework: Datenmodell** und **Von Access zu Entity Framework: Daten** aus Ausgabe 5/2018.

Nun wollen wir versuchen, die Migration einfacher Formulare von Access aus in funktionsfähige WPF-Fenster zu erreichen.

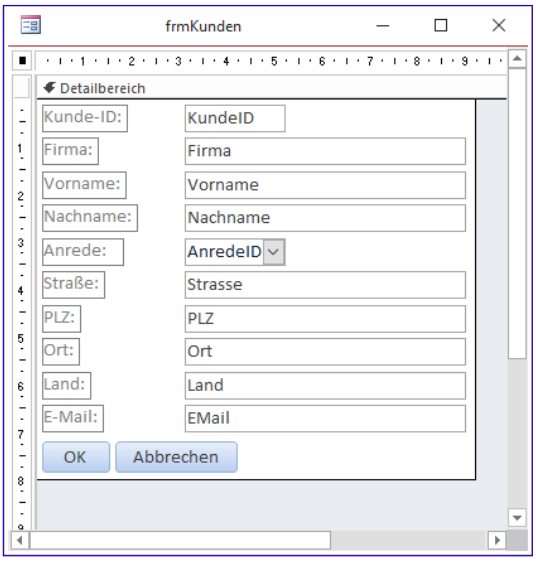


Bild 1: Das Formular **frmKunden** in der Formularansicht

Beispiel: Kundenformular

Unser Beispiel finden Sie in der Beispieldatenbank **Bestellverwaltung.accdb** im Download zu diesem Artikel. Wir wollen das Formular **frmKundendetails** als WPF-Fenster nachbauen. Das Formular sehen Sie in Bild 1 in der Entwurfsansicht. Es verwendet die Tabelle **tblKunden** als Datensatzquelle. Die Textfelder sind jeweils an entsprechende Felder der Datensatzquelle des Formulars gebunden. Das Kombinationsfeld **cboAnredeID** verwendet die Abfrage **SELECT [tblAnreden].[AnredeID], [tblAnreden].[Anrede] FROM tblAnreden;** als Datensatzherkunft und ist an das Feld **AnredeID** gebunden. Für das erste Feld der Datensatzherkunft haben wir die Spaltenbreite auf **0cm** eingestellt, damit dieses nicht angezeigt wird. Es erscheint also nur das zweite Feld der Datensatzherkunft.

Projekt erstellen

Wir erstellen ein neues Projekt namens **AccessZuWPFFormulare** und führen die im Artikel **Von Access zu EF: Step by step** aus Ausgabe 5/2018 beschriebenen Schritte aus, um die Tabellenentwürfe und Daten der Tabellen unserer Beispieldatenbank als Entity Data Model hinzuzufügen. Das gelingt innerhalb von ungefähr fünf Minuten, wenn Sie die Beispieldatenbank dazu verwenden. Für andere Datenbanken haben wir dies noch nicht getestet. Wenn Sie versuchen, andere Datenbanken zu migrieren und auf Fehler stoßen, teilen Sie uns das gern mit. Wir benötigen dann allerdings eine Kopie des Datenmodells im Access-Format, damit wir damit experimentieren und unsere Tools verbessern können (per E-Mail an **andre@minhorst.com**).

Zum Testen der migrierten Datenbank verwenden Sie eine neue Schaltfläche auf der Seite **MainWindow.xaml**, welche die folgende Methode aufruft:

```
Private Sub btnTestDatabase_Click(sender As Object, e As RoutedEventArgs)
    Dim dbContext As BestellverwaltungContext
    Dim Kunden As ObservableCollection(Of Kunde)
    dbContext = New BestellverwaltungContext
    Kunden = New ObservableCollection(Of Kunde)(dbContext.Kunden)
    MessageBox.Show("Anzahl Kunden: " + Kunden.Count().ToString())
End Sub
```

Danach sollte beim Starten des Projekts und Anklicken der Schaltfläche ein Meldungsfenster mit der Anzahl der Datensätze der Tabelle **Kunden** erscheinen.

Vom Formular zu WPF

Nun schauen wir uns an, was für eine Aufgabe überhaupt vor uns liegt. Wir wollen das Formular **frmKundendetails** aus der Access-Datenbank als Fenster der WPF-Anwendung abbilden. Was benötigen wir dazu? Zunächst einmal ist soll das Fenster die Größe erhalten, die auch unser Access-Formular aufweist. Dann wollen wir die Steuerelemente auslesen und im WPF-Fenster anlegen. Diese sollen, genau wie das Fenster, mit den Eigenschaften zum Binden des Fensters und der Steuerelemente an die Daten versehen werden. Schließlich fehlen noch die Ereignisprozeduren, die durch die Steuerelemente und das Formular selbst ausgelöst werden.

Wir wollen zunächst von Access aus starten und hier Code schreiben, mit dem wir die notwendigen Informationen aus dem Formularentwurf auslesen und daraus automatisiert die XAML-Definition für das WPF-Fenster sowie den Inhalt der Code behind-Daten für das Fenster erstellen. Diesen kopieren wir dann zunächst von Hand in ein neues Fenster-Element in unserem Projekt.

Fenstergröße

Wir beginnen einmal mit einer vermeintlich einfachen Aufgabe: dem Ermitteln der Fenstergröße und dem Erstellen des Code-Elements, welches diese beiden Eigenschaften enthält. Wenn wir die Eigenschaften im Eigenschaftenblatt von Access ansehen, erhalten wir Angaben in der Einheit Zentimeter, also beispielsweise **9,198cm** für die Formularbreite und **9,198cm** für die Höhe des Detailbereichs. Geben wir die Eigenschaften **Width** des Form-Elements und **Me.Height** des Detailbereichs (**Me.Section(0).Height**) aus, erhalten wir Werte wie **4535** und **5215**. Wenn wir die Eigenschaften in der XAML-Definition betrachten, finden wir Werte wie **450** für die Eigenschaft **Height** und **800** für die Eigenschaft **Width**. Wir müssen die Zahlen also noch umrechnen. Die Größe und Breite unter Visual Studio wird in Pixel angegeben. Unter Access werden diese Werte in Twips angegeben. Zur Umrechnung von Twips in Pixel gibt es ausreichend Funktionen, von denen wir zwei im Modul **mdlUmrechnungen** der Beispieldatenbank eingefügt haben. Sie heißen **TwipsToPixelHeight** und **TwipsToPixelWidth**. Wir benötigen zwei Funktionen, weil die Faktoren für die Umrechnungen unterschiedlich sein können.

Basisprozedur

Damit legen wir nun die Basis für das Erstellen des WPF-Codes. Die benötigte VBA-Prozedur sieht wie folgt aus, wobei wir zunächst einige Variablen deklarieren:

```
Public Sub FormularNachWPF()
    Dim frm As Form
    Dim strForm As String
    Dim strXAML As String
    Dim lngHeight As Long
    Dim lngWidth As Long
    Dim strTitle As String
```

Dann öffnen wir das Formular in der Entwurfsansicht und weisen dieses der Variablen **frm** zu:

```
    strForm = "frmKundendetails"
    DoCmd.OpenForm strForm, acDesign
    Set frm = Forms(strForm)
```

Dann ermitteln wir mit den Umrechnungsfunktionen die Höhe und Breite:

```
    lngHeight = TwipsToPixelsHeight(frm.Section(0).Height)
    lngWidth = TwipsToPixelswidth(frm.Width)
```

Außerdem prüfen wir, ob ein Titel für das Formular angegeben wurde und schreiben diesen oder alternativ den Formularnamen in die Variable **strTitle**:

```
    If Not Len(frm.Caption) = 0 Then
        strTitle = frm.Caption
    Else
        strTitle = frm.Name
    End If
```

Schließlich bauen wir den grundlegenden XAML-Code für das Fenster zusammen, wobei wir den standardmäßig verwendeten Code nutzen und unsere ermittelten Werte für Fenstername, Titel, Höhe und Breite an den entsprechenden Stellen einfügen:

```
    strXAML = strXAML & "<Window x:Class=""" & frm.Name & """" & vbCrLf
    strXAML = strXAML & "        xmlns=""http://schemas.microsoft.com/winfx/2006/xaml/presentation""" & vbCrLf
    strXAML = strXAML & "        xmlns:x=""http://schemas.microsoft.com/winfx/2006/xaml""" & vbCrLf
    strXAML = strXAML & "        xmlns:d=""http://schemas.microsoft.com/expression/blend/2008""" & vbCrLf
    strXAML = strXAML & "        xmlns:mc=""http://schemas.openxmlformats.org/markup-compatibility/2006""" & vbCrLf
    strXAML = strXAML & "        xmlns:local=""clr-namespace:AccessZuWPFFormulare""" & vbCrLf
    strXAML = strXAML & "        mc:Ignorable=""d""" & vbCrLf
    strXAML = strXAML & "        Title=""" & strTitle & """ Height=""" & lngHeight & """ Width=""" & lngWidth & """>" & vbCrLf
    strXAML = strXAML & "    <Grid>" & vbCrLf
```

Dazwischen platzieren wir den Aufruf der Prozedur **SteuerelementeHinzufuegen**, die wir im Anschluss erstellen – für den ersten Testlauf bleibt diese auskommentiert:

```
    'SteuerelementeHinzufuegen frm, strXAML
    strXAML = strXAML & "    </Grid>" & vbCrLf
    strXAML = strXAML & "</Window>" & vbCrLf
```

Schließlich nutzen wir die Prozedur **InZwischenablage**, um den Inhalt der String-Variablen **strXAML** in die Zwischenablage zu kopieren (siehe **mdlZwischenablage**) und schließen das Formular wieder:

```
    Inzwischenablage strXAML
    DoCmd.Close acForm, strForm
End Sub
```

Wenn Sie diese Prozedur ausführen und den Inhalt der Zwischenablage als XAML-Code eines neuen Fensters einsetzen, erhalten Sie die Ansicht aus Bild 2.

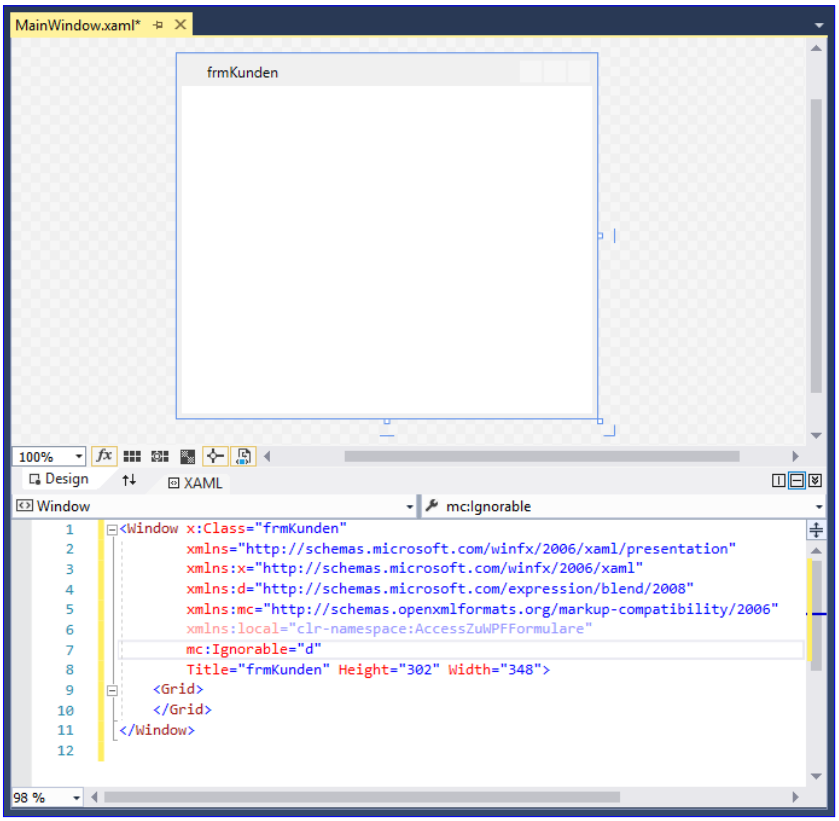


Bild 2: Höhe, Breite, Titel und Name wurden vom Access-Formular übernommen

Ob dieses Fenster unseren Ansprüchen genügt, müssen wir allerdings noch herausfinden – dazu benötigen wir im Fenster **MainWindows.xaml** noch eine Schaltfläche, mit der wir das neu erstellte Fenster öffnen können. Diese soll den folgenden Code ausführen:

```
Private Sub btnKundendetails_Click(sender As Object, e As RoutedEventArgs)
    Dim Kundendetails As frmKundendetails
    Kundendetails = New frmKundendetails()
    Kundendetails.Show()
End Sub
```

Steuerelemente migrieren

Als nächstes wollen wir uns um die Steuerelemente kümmern. Den folgenden Aufruf haben wir ja schon im Code platziert – Sie müssen nur noch die Markierung als Kommentar entfernen:

```
   SteuerelementeHinzufuegen frm, strXAML
```

Die Prozedur selbst erwartet die beiden übergebenen Parameter, nämlich einen Verweis auf das Formular (**frm**) und die Variable zum Zusammenstellen des XAML-Codes (**strXAML**). Damit starten wir in die erste Rohfassung der Prozedur:

```
Public Function SteuerelementeHinzufuegen(frm As Form, strXAML As String)
    Dim ctl As Control
    Dim lngWidth As Long
    Dim lngHeight As Long
    Dim lngTop As Long
    Dim lngLeft As Long
    Dim strCaption As String
```

In dieser durchlaufen wir alle Steuerelemente des Formulars über die **Controls**-Auflistung und referenzieren das aktuelle Steuerelement jeweils mit der Variablen **ctl**. Hier tragen wir zunächst die Größe und die Position in die entsprechenden Variablen ein:

```
    For Each ctl In frm.Controls
        lngWidth = TwipsToPixelsWidth(ctl.Width)
        lngHeight = TwipsToPixelsHeight(ctl.Height)
        lngTop = TwipsToPixelsHeight(ctl.Top)
        lngLeft = TwipsToPixelsWidth(ctl.Left)
```

Dann prüfen wir in einer **Select Case**-Schleife, welchen Typ das aktuell mit **ctl** referenzierte Steuerelement hat. In diesem Fall haben wir zunächst die Typen **acTextBox** und **acLabel** untersucht und fügen für jedes Bezeichnungsfeld und jedes Textfeld eine Zeile zu **strXAML** hinzu, welche das Aussehen des jeweiligen Steuerelements beschreibt:

```
        Select Case ctl.ControlType
            Case acTextBox
                strXAML = strXAML & "<TextBox x:Name=""" & ctl.Name & """ FontFamily=""Calibri"" FontSize=""" _
                    & ctl.FontSize & "pt"" HorizontalAlignment=""Left"" Height=""" & lngHeight & """ Margin=""" _
                    & lngLeft & "," & lngTop & ",0,0"" TextWrapping=""Wrap"" Text=""TextBox"" " _
                    & "VerticalAlignment=""Top"" Width=""" & lngWidth & """/>" & vbCrLf
            Case acLabel
                strCaption = ctl.Caption
                strXAML = strXAML & "<Label x:Name=""" & ctl.Name & """ Content=""" & strCaption _
                    & """ FontFamily=""Calibri"" FontSize=""11pt"" Padding=""0"" " _
                    & "VerticalContentAlignment=""Center"" HorizontalAlignment=""Left"" Height=""" & lngHeight _
                    & """ Margin=""" & lngLeft & "," & lngTop & ",0,0"" VerticalAlignment=""Top"" Width=""" & lngWidth _
                    & """/>" & vbCrLf
            Case acCommandButton
            Case acComboBox
            Case Else
```

```
                Debug.Print "Steuerelementtyp nicht behandelt: " & ctl.Name, ctl.ControlType
        End Select
    Next ctl
End Function
```

Die übrigen Steuerelemente schauen wir uns später an. Wenn wir die Prozedur **FormularNachWPF** nun erneut ausführen, erhalten wir Code, der das Fenster aus Bild 3 erstellt.

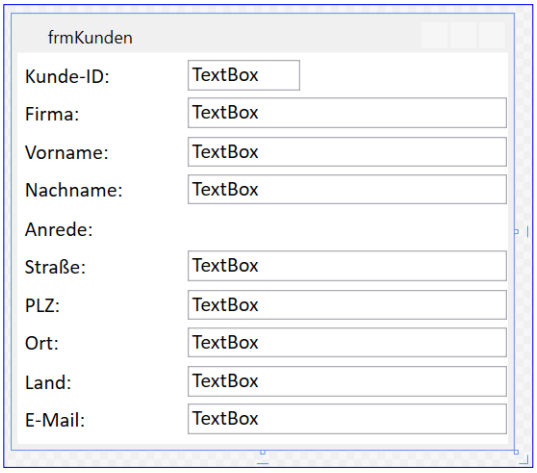


Bild 3: Textfelder und Bezeichnungsfelder

Zwischenergebnis prüfen

Nachdem wir nun das Fenster und die enthaltenen Steuerelemente definiert haben, können wir die Anwendung einmal starten und das Fenster **frmKundendetails.xaml** einmal über das geöffnete Access-Formular **frmKundendetails** legen. Das Ergebnis sieht wie in Bild 4 aus. Hier stellen wir fest, dass etwas mit der Berechnung von Höhe und Breite des Fensters nicht funktioniert hat. Das WPF-Fenster ist insgesamt kleiner als der Innenraum des Formulars. Berechnet haben wir eine Breite von 348 Pixeln und Höhe von 302 Pixeln. Erstellen wir einen Screenshot und schauen uns die Dimensionen im Screenshotprogramm an, finden wir Maße von 334 x 295 Pixeln vor. Schauen wir uns das Verhältnis für das Fenster **MainWindow.xaml** an, finden wir den gleichen absoluten Fehler – es fehlen 14 Pixel in der Breite und 7 in der Höhe. Legen wir das Fenster nun erneut über das Formular, erkennen wir, dass es den Innenraum komplett abdeckt. Allerdings müssen wir noch einige Pixel für die Titelleiste des WPF-Fensters hinzurechnen – dann passen auch die Schaltflächen noch unter die gebundenen Steuerelemente. Wir messen das grob im Screenshotprogramm und erhalten 23 Pixel, die wir noch zur Höhe hinzuaddieren.

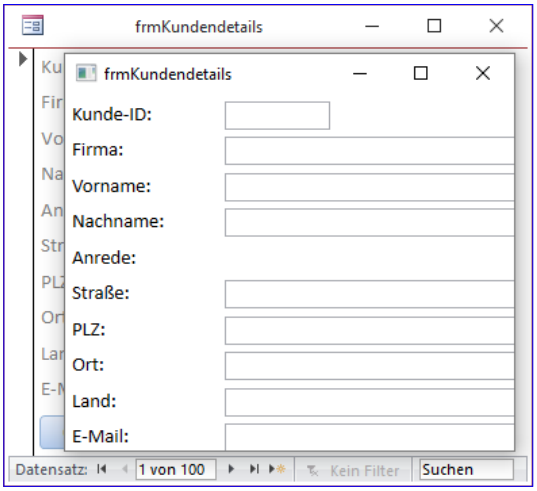


Bild 4: Die Größe von Original und Kopie stimmt nicht überein.

Standardeigenschaften fensterweit definieren

Wenn wir uns die beiden folgenden Elemente ansehen, erkennen wir einige Eigenschaften, die wir für die gleichen Steuerelementtypen immer wieder verwenden werden – zum Beispiel die Schriftart oder die Ausrichtung:

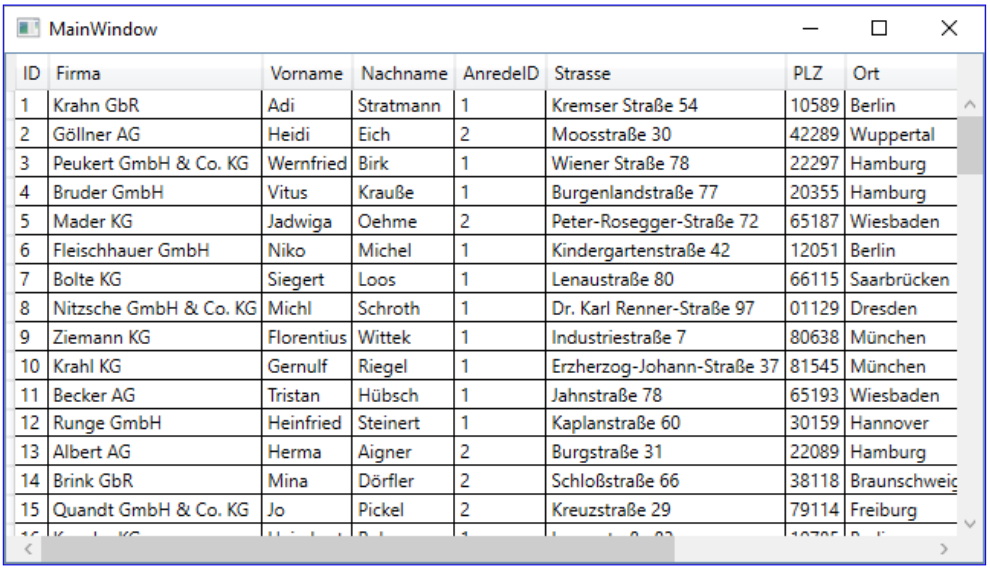
```
<TextBox x:Name="txtKundeID" FontFamily="Calibri" FontSize="11pt" HorizontalAlignment="Left" Height="21" Mar-
gin="118,5,0,0" TextWrapping="Wrap" Text="TextBox" VerticalAlignment="Top" Width="78"/>
<Label x:Name="Bezeichnungsfeld0" Content="Kunde-ID:" FontFamily="Calibri" FontSize="11pt" Padding="0" VerticalContentAli
gnment="Center" HorizontalAlignment="Left" Height="21" Margin="5,5,0,0" VerticalAlignment="Top" Width="67"/>
```

EF: Daten abfragen mit VB und LINQ

Unter Access waren Sie es gewöhnt, auf einfache Weise Abfragen mit der Abfrage-Entwurfsansicht zu erstellen. Ein paar Tabellen hinzufügen, die Felder auswählen, Kriterien, Sortierungen und Gruppierungen hinzufügen – fertig war die Abfrage. Gegebenenfalls haben Sie SQL-Kenntnisse und konnten SQL-Anweisungen für den Einsatz in VBA-Anweisungen von Hand schreiben. Unter VB und Entity Framework sieht das anders aus, weil wir ja nicht mehr auf Tabellen zugreifen, sondern auf Objekte. Und für die gibt es eine andere Abfragesprache, die sich direkt in den VB-Code integrieren lässt. Dieser Artikel stellt die Abfragetechnik LINQ für Visual Basic vor.

Voraussetzungen

Um die Beispiele dieses Artikels nachstellen zu können, benötigen das Entity Data Model aus dem Beispielprojekt aus dem Download zu diesem Artikel. Für die Beispielabfragen fügen wir dem Fenster **MainWindow.xaml** unserer Beispielanwendung ein **DataGrid**-Steuerelement hinzu. Der Code zum Füllen des **DataGrid**-Elements mit allen Eigenschaften aller **Kunden**-Elemente wie in Bild 1 sieht so aus:



ID	Firma	Vorname	Nachname	AnredeID	Strasse	PLZ	Ort
1	Krahn GbR	Adi	Stratmann	1	Kremser Straße 54	10589	Berlin
2	Göllner AG	Heidi	Eich	2	Moosstraße 30	42289	Wuppertal
3	Peukert GmbH & Co. KG	Wernfried	Birk	1	Wiener Straße 78	22297	Hamburg
4	Bruder GmbH	Vitus	Krauße	1	Burgenlandstraße 77	20355	Hamburg
5	Mader KG	Jadwiga	Oehme	2	Peter-Rosegger-Straße 72	65187	Wiesbaden
6	Fleischhauer GmbH	Niko	Michel	1	Kindergartenstraße 42	12051	Berlin
7	Bolte KG	Siegert	Loos	1	Lenaustraße 80	66115	Saarbrücken
8	Nitzsche GmbH & Co. KG	Michl	Schroth	1	Dr. Karl Renner-Straße 97	01129	Dresden
9	Ziemann KG	Florentius	Wittek	1	Industriestraße 7	80638	München
10	Krahl KG	Gernulf	Riegel	1	Erzherzog-Johann-Straße 37	81545	München
11	Becker AG	Tristan	Hübsch	1	Jahnstraße 78	65193	Wiesbaden
12	Runge GmbH	Heinfried	Steinert	1	Kaplanstraße 60	30159	Hannover
13	Albert AG	Herma	Aigner	2	Burgstraße 31	22089	Hamburg
14	Brink GbR	Mina	Dörfler	2	Schloßstraße 66	38118	Braunschweig
15	Quandt GmbH & Co. KG	Jo	Pickel	2	Kreuzstraße 29	79114	Freiburg

Bild 1: DataGrid mit allen Kundendaten

```
Imports System.Collections.ObjectModel

Class MainWindow
    Private dbContext As BestellverwaltungContext
    Private _Kunden As ObservableCollection(Of Kunde)

    Public Property Kunden As ObservableCollection(Of Kunde)
        Get
            Return _Kunden
        End Get
        Set(value As ObservableCollection(Of Kunde))
```

```
            _Kunden = value
        End Set
    End Property

    Public Sub New()
        InitializeComponent()
        dbContext = New BestellverwaltungContext
        Kunden = New ObservableCollection(Of Kunde)(dbContext.Kunden)
        DataContext = Me
    End Sub
End Class
```

Den XAML-Code haben wir so einfach wie möglich gehalten:

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="450" Width="800">
    <Grid>
        <DataGrid ItemsSource="{Binding Kunden}"></DataGrid>
    </Grid>
</Window>
```

Die Anweisung der Konstruktor-Methode **New()**, die wir als Ausgangspunkt für die folgenden Beispiele nutzen, ist diese:

```
Kunden = New ObservableCollection(Of Kunde)(dbContext.Kunden)
```

Hier weisen wir der **ObservableCollection** mit Elementen des Typs **Kunde** einfach alle Kunden des Datenbankkontextes zu, also **dbContext.Kunden**. Statt **dbContext.Kunden** können wir auch die nachfolgend vorgestellten Abfrageausdrücke verwenden.

Einfache Auswahlabfrage

Die einfachste Syntax bei einer LINQ-Abfrage sieht wie folgt aus – hier noch in Zusammenhang mit unserer Anweisung:

```
Kunden = New ObservableCollection(Of Kunde)(From Kunde In dbContext.Kunden)
```

Der Einfachheit halber schauen wir uns nun noch den Teil an, den wir in eine **ObservableCollection** umwandeln, hier also diesen Teil:

```
From Kunde In dbContext.Kunden
```

Dies liefert alle Elemente der **Kunden**-Auflistung. **dbContext.Kunden** ist dabei die Quelle der Daten. **Kunde** ist eine Variable, auf die wir mit weiteren Schlüsselwörtern der Abfragesprache LINQ zugreifen können. Diese Abfrage liefert das gleiche Ergebnis wie **dbKontext.Kunden**.

Eine ausführlichere Schreibweise unter Einbeziehung des **Select**-Schlüsselworts sieht wie folgt aus und liefert wiederum das gleiche Ergebnis:

```
From Kunde In dbContext.Kunden Select
Kunde
```

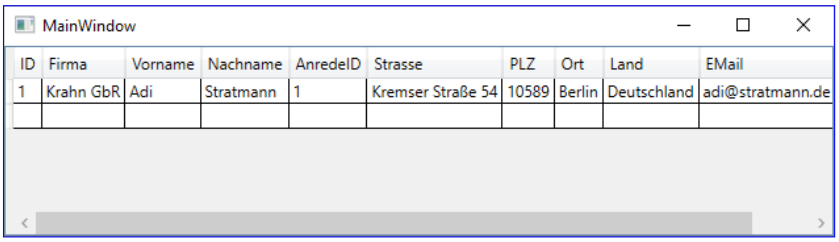


Bild 2: DataGrid mit dem Kunden mit dem Wert **1** im Feld **ID**

Daten filtern mit Where

Wenn Ihnen die bisherigen Schlüsselwörter schon von SQL bekannt vorkamen, gibt es gute Nachrichten – das **Where**-Schlüsselwort dürften Sie auch kennen. Dieses hängen Sie hinter die **From**-Anweisung an und vor der **Select**-Anweisung.

Das folgende Beispiel selektiert beispielsweise nur den Kunden, dessen Feld **ID** den Wert **1** aufweist (Ergebnis siehe Bild 2):

```
From Kunde In dbContext.Kunden Where Kunde.ID = 1 Select Kunde
```

Ersatz für den SQL-LIKE-Operator: Contains, StartsWith und EndsWith

Unter SQL haben Sie Vergleiche mit Platzhaltern wie dem Sternchen mit dem **LIKE**-Operator realisiert, also etwa mit **SELECT * FROM tblKunden WHERE PLZ LIKE "1*"**, um alle Kunden mit einer PLZ, die mit **1** beginnt, zu ermitteln.

Unter LINQ gibt es dazu die drei Operatoren **Contains**, **StartsWith** und **EndsWith**:

- **Contains**: Entspricht der Suche nach einem Ausdruck, der irgendwo im Feldinhalt vorkommen darf, also gleichbedeutend mit **LIKE "*<Suchbegriff>*"**.
- **StartsWith**: Sucht nach Datensätzen, deren zu durchsuchendes Feld mit dem angegebenen Ausdruck beginnt. Unter SQL wäre das **LIKE "<Suchbegriff>*"**.
- **EndsWith**: Sucht nach Datensätzen, deren zu durchsuchendes Feld auf den angegebenen Ausdruck endet. Unter SQL entspricht das **LIKE "*<Suchbegriff>"**.

Die Operatoren hängen wir an das zu durchsuchende Feld an und geben den Suchbegriff in Klammern an. Alle Kunden, deren **PLZ** mit **1** beginnt, ermitteln wir also mit dem folgenden Ausdruck:

```
From Kunde In dbContext.Kunden Where Kunde.PLZ.StartsWith("1") Select Kunde
```

Kriterien verknüpfen

Die Suchkriterien verknüpfen Sie einfach wie von SQL gewohnt mit den Operatoren **Or** und **And**. Kunden, deren Vorname und Nachname mit **A** beginnt, finden Sie beispielsweise so:

```
From Kunde In dbContext.Kunden Where Kunde.Vorname.StartsWith("A") Or Kunde.Nachname.StartsWith("A") Select Kunde
```

Daten sortieren mit Order By

Natürlich können Sie die Daten auch in beliebiger Sortierung in die **ObservableCollection** füllen. Dazu verwenden wir den folgenden Abfrageausdruck, in dem wir das **Order By**-Schlüsselwort gefolgt von dem zu sortierenden Feld angeben:

```
From Kunde In dbContext.Kunden Order By Kunde.Nachname Select Kunde
```

Wenn Sie nach mehreren Feldern sortieren wollen, geben Sie diese durch ein Komma voneinander getrennt hinter dem **Order By**-Schlüsselwort an:

```
From Kunde In dbContext.Kunden Order By Kunde.Nachname, Kunde.Vorname Select Kunde
```

Hier haben wir noch nicht explizit die Sortierreihenfolge angegeben. In diesem Fall sortiert die Abfrage aufsteigend nach den angegebenen Feldern. Sie können die Sortierreihenfolge auch explizit angeben, indem Sie eines der Schlüsselwörter **Ascending** (**Aufsteigend**) oder **Descending** (**Absteigend**) hinter dem Sortierfeld angeben:

```
From Kunde In dbContext.Kunden Order By Kunde.Nachname Ascending, Kunde.Vorname Descending Select Kunde
```

Reihenfolge der Elemente

Im Gegensatz zu SQL, wo die Reihenfolge strikt festgelegt ist (zum Beispiel **SELECT ... FROM ... WHERE ... ORDER BY**), können Sie in LINQ etwa die Reihenfolge der **Where**- und der **Order By**-Klausel vertauschen. Wir empfehlen jedoch, die Reihenfolge wie unter SQL zu gestalten, um die Lesbarkeit zu erhöhen. Die folgende Abfrage, bei der wir die **Order By**-Klausel vor der **Where**-Klausel platziert haben, funktioniert jedoch genauso wie die mit der umgekehrten Reihenfolge:

```
From Kunde In dbContext.Kunden Order By Kunde.Nachname Ascending Where Kunde.Nachname.StartsWith("A") Select Kunde
```

Die »Laufvariable«

Die Bezeichnung zwischen **From** und **In**, in unserem Fall also etwa **Kunde**, können Sie beliebig wählen. Wenn Sie den Abfragecode kürzer darstellen wollen, können Sie auch einfach **k** verwenden:

```
From k In dbContext.Kunden Where k.Nachname.StartsWith("A") Order By k.Nachname Ascending Select k
```

Zu selektierende Daten festlegen mit Select

Die **Select**-Anweisung haben wir in den obigen Beispielen schon ausgiebig genutzt. Allerdings haben wir mit **... Select Kunde** immer einfach die kompletten **Kunde**-Objekte selektiert. Das hätten wir auch erreicht, wenn wir die **Select**-Klausel weggelassen hätten. **Select** bietet allerdings eine Menge mehr als nur die Möglichkeit, genau die Objekte zurückzuliefern, welche die zu untersuchende Auflistung bietet, in diesem Fall die Liste der **Kunde**-Objekte. In allen Fällen, in denen Sie nicht einfach die Objekte zurückliefern, die in der zu untersuchenden Auflistung enthalten sind, handelt es sich um eine sogenannte Projektion. Schauen wir uns an, wie das aussehen kann! Wenn wir beispielsweise nur die ID, den Vornamen und den Nachnamen des Kunden erhalten wollen, formulieren wir den Ausdruck wie folgt:

```
From k In dbContext.Kunden Select k.ID, k.Vorname, k.Nachname
```

Allerdings liefert dies dann kein Objekt vom Typ **Kunde** mehr zurück, was sich in einer entsprechenden Fehlermeldung bemerkbar macht (siehe Bild 3). Also spielen wir erst einmal außerhalb des DataGrids mit dieser Möglichkeit:

```
Dim KundenMitName = From k In dbContext.Kunden Select k.ID, k.Vorname, k.Nachname
For Each k In KundenMitName
    Debug.Print(k.ID.ToString() + " " + k.Vorname + " " + k.Nachname)
Next
```

Wir weisen also das Ergebnis, das die Eigenschaften **ID**, **Vorname** und **Nachname** der Elemente aus **dbContext.Kunden** enthält, der Variablen **KundenMitName** zu. **KundenMitName** wir dann zu einer Variablen des Typs **DbQuery**. Auf die enthaltenen Eigenschaften können wir wie auf die Eigenschaften einer Klasse zugreifen. Im Beispiel oben durchlaufen wir diese Elemente in einer **For Each**-Schleife, wobei wir das aktuelle Element der Variablen **k** zuweisen. Diese hält dann die Eigenschaften **ID**, **Vorname** und **Nachname** bereit. Diese geben wir innerhalb der Schleife mit der **Debug.Print**-Anweisung im **Output**-Fenster aus.

DbQuery-Ergebnis in DataGrid anzeigen

Wie aber zeigen wir dieses Ergebnis im DataGrid an? Dieses ist ja an die Eigenschaft **Kunden** der Code behind-Klasse gebunden. Wenn wir dies nicht ändern wollen, müssen wir die Informationen **ID**, **Vorname** und **Nachname** anderweitig in Elemente des Typs **Kunde** schreiben und diese dann der Auflistung zuweisen. Aber gelingt uns das? Der theoretische Ansatz sieht wie folgt aus:

```
Dim KundenMitName = From k In dbContext.Kunden Select New Kunde With {.ID = k.ID, .Vorname = k.Vorname, .Nachname =
k.Nachname}
Kunden = New ObservableCollection(Of Kunde)(KundenMitName)
```

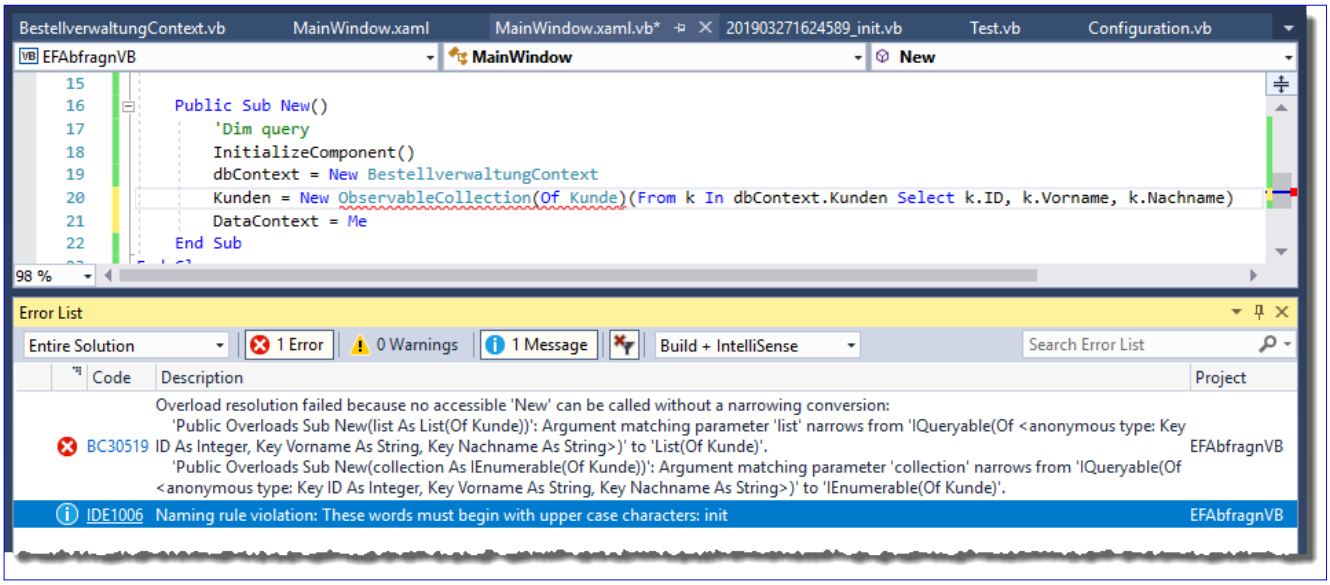


Bild 3: Fehlermeldung, wenn keine Objekte des Typs **Kunde** zurückgeliefert werden

Hier weisen wir einer Variablen namens **KundenMitName** das Ergebnis der Abfrage in Form von Objekten mit den Eigenschaften **ID**, **Vorname** und **Nachname** zu. Danach wollen wir das Ergebnis der öffentlichen **ObservableCollection**-Variablen **Kunden** zuweisen. Das liefert allerdings den Fehler **The entity or complex type 'EFAbfragnVB.Kunde' cannot be constructed in a LINQ to Entities query**. Der Grund für den Fehler ist, dass die Kunde-Entitäten nur teilweise mit den Daten aus der Datenbank gefüllt werden, was in einem nicht aktualisierbaren Zustand resultiert.

Also verwenden wir doch eine neue Klasse, um das neue Ergebnis zu speichern. Diese Klasse sieht wie folgt aus:

```
Public Class KundeMitName
    Public Property ID As System.Int32
    <StringLength(255)>
    <Required>
    Public Property Vorname As System.String
    <StringLength(255)>
    <Required>
    Public Property Nachname As System.String
End Class
```

Den Code der Code behind-Klasse von **KundenMitName.xaml** gestalten wir wie folgt, wobei wir die neue private Variable und die öffentliche Eigenschaft **_KundenMitName** und **KundenMitName** nennen:

```
Public Class KundenMitName
    Private dbContext As BestellverwaltungContext
    Private _KundenMitName As List(Of KundeMitName)

    Public Property KundenMitName As List(Of KundeMitName)
        Get
            Return _KundenMitName
        End Get
        Set(value As List(Of KundeMitName))
            _KundenMitName = value
        End Set
    End Property
```

In der Konstruktor-Methode füllen wir die Variable **KundenMitNamen** dann über die folgende Anweisung, in der wir hinter der **Select**-Klausel jeweils ein neues **KundeMitName**-Objekt erzeugen und seinen drei Eigenschaften **ID**, **Vorname** und **Nachname** die entsprechenden Werte des jeweiligen Datensatzes zuweisen:

```
    Public Sub New()
        InitializeComponent()
        dbContext = New BestellverwaltungContext
```

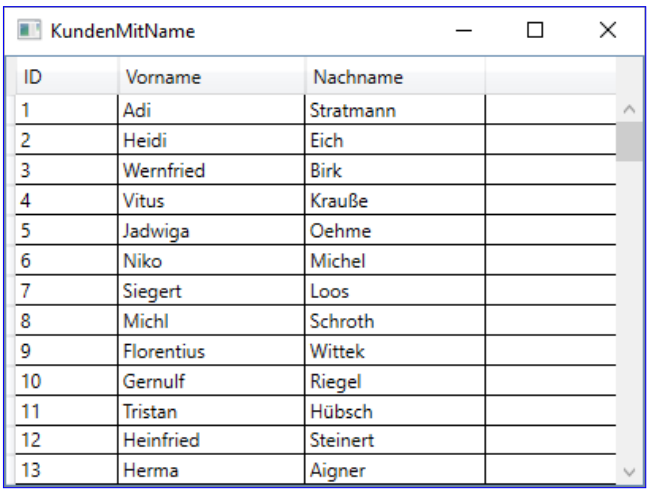
```
        KundenMitName = New ObservableCollection(Of KundeMitName)(From k In dbContext.Kunden Select New KundeMitName _
            With {.ID = k.ID, .Vorname = k.Vorname, .Nachname = k.Nachname})
        DataContext = Me
    End Sub
End Class
```

Das Ergebnis sieht dann wie in Bild 4 aus.

Auf die gleiche Art und Weise können Sie natürlich auch das erzeugen, was sich unter Access »berechnete Felder« nennt – also beispielsweise eine neue Eigenschaft, die Vorname und Nachname etwa in der Form **<Nachname>, <Vorname>** zusammenfasst.

Die ersten x Ergebnisse zurückliefern

Wenn Sie das abbilden wollen, was wir unter SQL mit dem **TOP**-Statement erreichen, nämlich die Ausgabe nur einer bestimmten Anzahl von Datensätzen, verwenden Sie das **Take**-Schlüsselwort. Für die ersten zehn Elemente der **Kunden**-Tabelle verwenden Sie einfach **Take 10** nach der Angabe der Datenquelle:



ID	Vorname	Nachname
1	Adi	Stratmann
2	Heidi	Eich
3	Wernfried	Birk
4	Vitus	Krauße
5	Jadwiga	Oehme
6	Niko	Michel
7	Siegert	Loos
8	Michl	Schroth
9	Florentius	Wittek
10	Gernulf	Riegel
11	Tristan	Hübsch
12	Heinfried	Steinert
13	Herma	Aigner

Bild 4: DataGrid mit bestimmten Werten eines Kunden

```
From k In dbContext.Kunden Take 10
```

Gruppierungen

Auch zum **GROUP BY**-Schlüsselwort zum Gruppieren von Daten zum Zwecke der Ermittlung etwa von Summen, Durchschnittswerten et cetera bietet LINQ ein Pendant. Dabei handelt es sich um **Group By**-Klausel. Diese setzen wir etwa wie folgt ein, wobei wir hier bereits einen berechneten Ausdruck als Grundlage für die Gruppierung verwenden. Dabei ermitteln wir mit der **From**-Klausel eine Variable namens **Bestellung**, die wir mit den Elementen der Tabelle **Bestellungen** füllen. Diese wählen wir mit **Select** aus und gruppieren diese dann mit der **Group By**-Klausel, für die wir in **MonatJahr** einen Ausdruck zusammenstellen, der aus dem Jahr, einem Schrägstrich und dem Monat des Bestelldatums zusammengesetzt wird. Schließlich folgen noch das **Into**-Schlüsselwort und ein Alias namens **Bestellungen** für die Gruppierung **Group**:

```
Dim BestellungenNachMonat = From Bestellung In dbContext.Bestellungen
                            Select Bestellung
                            Group By MonatJahr = Bestellung.Bestelldatum.Year.ToString() + "/" _
                                + Bestellung.Bestelldatum.Month.ToString()
                                Into Bestellungen = Group
```

Hier gibt es gegenüber SQL eine Erleichterung. Wir können nicht nur Aggregatfunktionen auf die Elemente einer Gruppierung anwenden, sondern die Datensätze auch einfach nur über die jeweilige Gruppierung ansprechen. Das heißt, dass wir nicht nur

die Elemente der oben definierten Auflistung **BestellungenNachMonat** durchlaufen können, sondern auch die in der jeweiligen Gruppierung enthaltenen Elemente der ursprünglichen Datenquelle. Die äußere **For Each**-Schleife des folgenden Beispiels durchläuft dabei etwa alle Gruppierungen aus **BestellungenNachMonat**, die mit der Variablen **BestellungMonat** referenziert werden, sondern auch noch die in jeder Gruppierung enthaltenen Elemente der Auflistung **Bestellungen** – die wir in einer inneren **For Each**-Schleife durchlaufen:

```
For Each BestellungMonat In BestellungenNachMonat
    Debug.Print(BestellungMonat.MonatJahr)
    For Each Bestellung In BestellungMonat.Bestellungen
        Debug.Print("  " + Bestellung.Bestelldatum)
    Next
Next
```

Das Ergebnis sieht im Direktbereich von Visual Studio schließlich wie in Bild 6 aus.

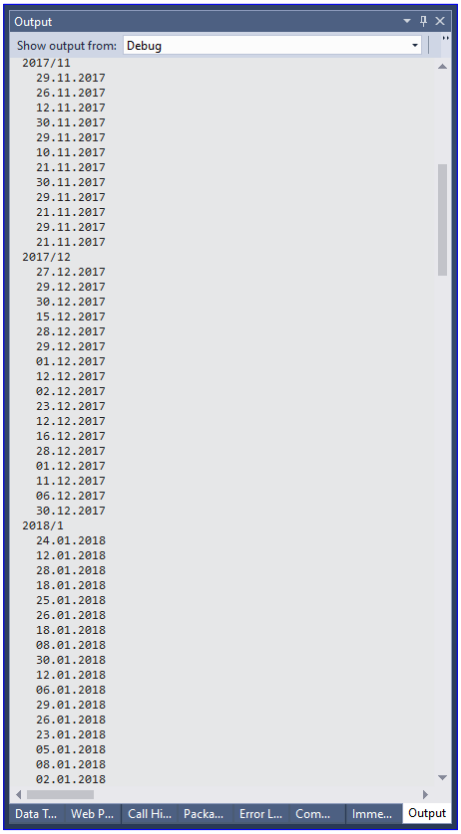


Bild 6: Gruppierte Ausgabe nach Jahr und Monat

Beziehungen abbilden

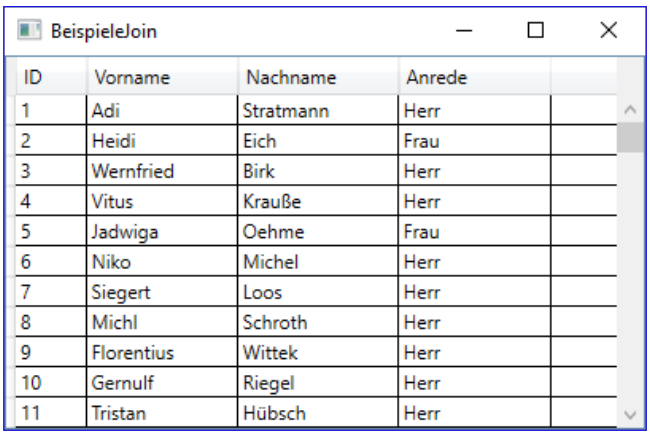
Natürlich werden Sie nicht immer nur auf die Daten einer einzelnen Entität zugreifen wollen, sondern gelegentlich auch auf die Daten aus mehreren Tabellen – oder zumindest werden Sie mehrere verknüpfte Tabellen verwenden, um die Kriterien für die Ermittlung der benötigten Daten zu definieren.

Im ersten Beispiel schauen wir uns an, wie wir eine Collection der Kunden um die Anrede aus der verknüpften Tabelle **Anreden** anreichern. Dazu benötigen wir wieder eine neue Klasse, welche Eigenschaften der **Kunde**-Klasse enthält und die Eigenschaft **Anrede** der **Anreden**-Tabelle:

```
Public Class KundeMitAnrede
    Public Property ID As System.Int32
    Public Property Vorname As System.String
    Public Property Nachname As System.String
    Public Property Anrede As System.String
End Class
```

Das Beispiel haben wir etwas umgeschrieben, sodass der Code der Code behind-Klasse des neuen Fensters namens **BeispieleJoin.xaml** wie folgt aussieht:

```
Public Class BeispieleJoin
    Private dbContext As BestellverwaltungContext
```



ID	Vorname	Nachname	Anrede
1	Adi	Stratmann	Herr
2	Heidi	Eich	Frau
3	Wernfried	Birk	Herr
4	Vitus	Krauße	Herr
5	Jadwiga	Oehme	Frau
6	Niko	Michel	Herr
7	Siegert	Loos	Herr
8	Michl	Schroth	Herr
9	Florentius	Wittek	Herr
10	Gernulf	Riegel	Herr
11	Tristan	Hübsch	Herr

Bild 5: DataGrid mit einem Teil der Kundendaten

Entity Framework: Gespeicherte Prozeduren

Entity Framework bietet die Möglichkeit, mit einer Datenbank zu arbeiten, die lediglich einfache Tabellen enthält. Sie könnten so sämtliche Geschäftslogik in der Anwendung halten. Manch einer mag aber vielleicht Teile der Geschäftslogik in die Datenbank überführen oder, wenn die Datenbank schon existiert, dort belassen, damit diese von verschiedenen Anwendungen aus genutzt werden kann. Ein Beispiel sind Trigger, die bei Datenänderungen automatisch ausgelöst werden, ein anderes sind gespeicherte Prozeduren, über die sie sowohl Daten abfragen als auch anlegen, ändern oder löschen können. Dieser Artikel beschäftigt sich mit den gespeicherten Prozeduren und den Möglichkeiten, die sich unter Entity Framework zu ihrer Nutzung bieten.

Wenn Sie ein Objekt in Entity Framework anlegen wollen, erstellen Sie das Objekt neu und rufen die **Add**-Methode des jeweiligen DbSets auf. Der anschließende Aufruf der **SaveChanges**-Methode sorgt dann für das Übertragen der Änderung in der lokalen Auflistung in die Datenbank. Wenn Sie Eigenschaften eines Objekts ändern, übertragen Sie die Änderungen ebenfalls mit der **SaveChanges**-Methode. Und das geschieht auch, wenn Sie ein Element etwa mit der **Remove**-Methode entfernt haben – erst die **SaveChanges**-Methode löscht den entsprechenden Datensatz aus der zugrundeliegenden Tabelle.

Ausführung gespeicherte Prozeduren unter LINQ beobachten

Im Hintergrund führt die Anwendung entsprechende SQL-Anweisungen aus, wovon wir uns leicht überzeugen können, wenn wir den SQL Server Profiler mitlaufen lassen. Die grundlegende Bedienung erläutern wir im Artikel **SQL Server-Interaktion mit dem Profiler verfolgen** in Ausgabe 4/2016.

In unserem Fall wollen wir für die aktuelle Datenbank, deren ID wir im SQL Server Management Studio mit der folgenden Abfrage für die Zieldatenbank ermitteln, die ausgeführten SQL-Anweisungen anzeigen lassen:

```
SELECT DB_ID()
```

Das Ergebnis lautet beispielsweise **40**. In den Ablaufverfolgungseigenschaften des SQL Server Profilers aktivieren Sie die Option **Alle Spalten anzeigen**. Dann können Sie nach einem Klick auf **Spaltenfilter** im Dialog **Filter bearbeiten** einen Filter für die Eigenschaft **DatabaseID** festlegen, und zwar auf den oben ermitteln Wert, in unserem Fall **40** (siehe Bild 1).

Danach aktivieren Sie noch die Option **Alle Ereignisse anzeigen** und selektieren dann im Bereich **Ereignisauswahl** das Ereignis **Stored ProceduresIRCP: Completed** (siehe Bild 2). Damit legen wir fest, dass alle per Remote ausgeführten Abfragen vom Profiler erfasst und ausgegeben werden.

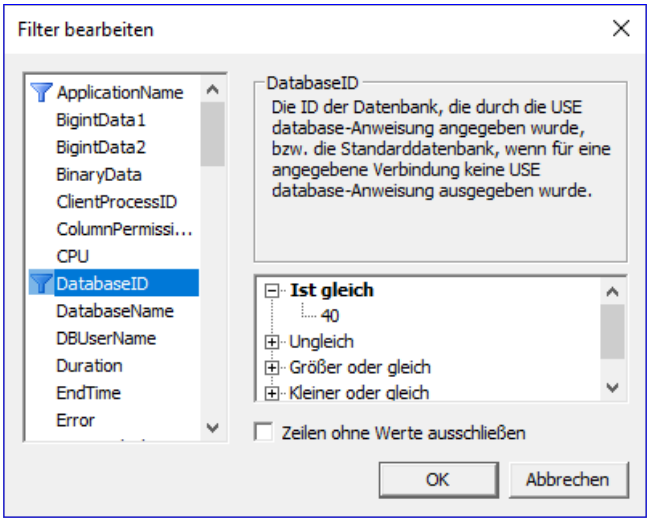


Bild 1: Festlegen eines Spaltenfilters

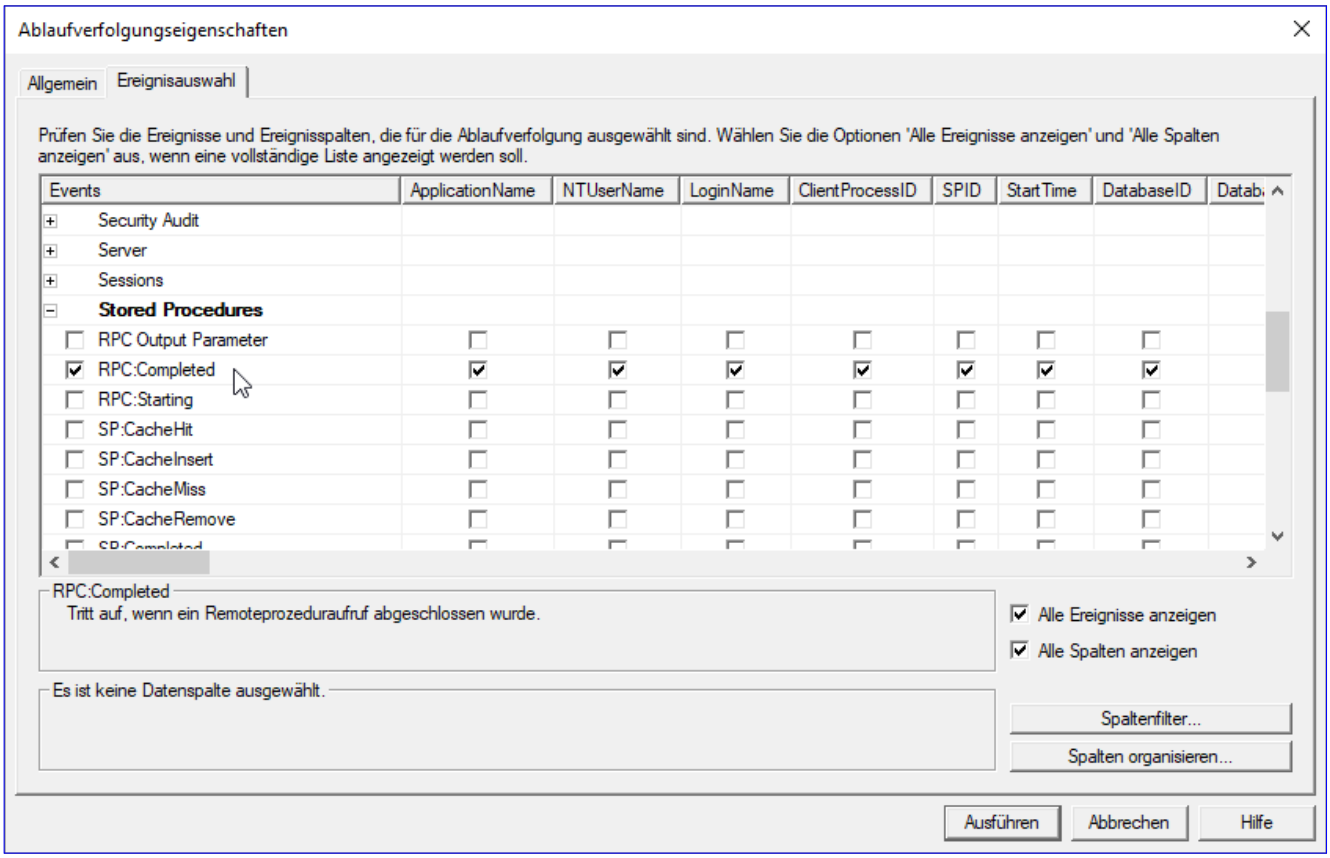


Bild 2: Auswahl der zu filternden SQL-Aktionen

Beispielprojekt vorbereiten

Als Beispielprojekt verwenden wir ein Projekt auf Basis der Vorlage **Visual Basic\Windows Desktop\WPF App**. Dieser fügen wir mit den Methoden aus dem Artikel **Von Access zu EF: Step by step** aus Ausgabe 5/2018, erweitert um die Informationen aus **Von Access zu EF: Update 1** aus der vorliegenden Ausgabe, ein Entity Data Model hinzu, aus dem wir dann eine entsprechende SQL Server-Datenbank generieren.

Automatisch generierte gespeicherte Prozedur ermitteln

Um zu erkennen, welche gespeicherten Prozeduren im Hintergrund ausgeführt werden, wenn wir Daten ändern, löschen oder hinzufügen, legen wir für das Fenster **MainWindow.xaml** einen Konstruktor in der Code behind-Klasse an, den wir wie folgt ausstatten.

Wir deklarieren dort einen Datenbankkontext namens **dbContext** und füllen diesen mit einer neuen Instanz unserer Datenbankkontext-Klasse **BestellverwaltungContext**:

```
Public Sub New()
    Dim dbContext As BestellverwaltungContext
    Dim Kunde As Kunde
    dbContext = New BestellverwaltungContext
```

Dann weisen wir der Objektvariablen **Kunde** den ersten Kunden zu, den wir in der Tabelle **Kunden** finden, bearbeiten die Eigenschaft **Firma** des Kunden und speichern die Änderung durch den Aufruf der Methode **SaveChanges**:

```
Kunde = dbContext.Kunden.First
Kunde.Firma = Kunde.Firma + "1"
dbContext.SaveChanges()
```

Dann entfernen wir diesen Kunden und speichern die Änderungen ebenfalls mit **SaveChanges**:

```
dbContext.Kunden.Remove(Kunde)
dbContext.SaveChanges()
```

Schließlich legen wir noch einen neuen Kunden an, fügen das **Kunde**-Element zur Auflistung der Kunden mit der **Add**-Methode hinzu und speichern die Änderungen mit **SaveChanges**:

```
    Kunde = New Kunde With {.Firma = "Firma", .AnredeID = 1, .Vorname = "Klaus", .Nachname = "Müller", _
        .EMail = "klaus@mueller.de", .Land = "Deutschland", .Ort = "Testhausen", .PLZ = "11111", .Strasse = "Teststr. 1"}
    dbContext.Kunden.Add(Kunde)
    dbContext.SaveChanges()
End Sub
```

Bevor wir die Anwendung öffnen, starten wir den SQL Server Profiler mit den obigen Einstellungen. Setzen Sie einen Haltepunkt in die erste Zeile mit der Anweisung **dbContext.SaveChanges**. Dann starten Sie die Anwendung. Nachdem der Haltepunkt erreicht ist, lassen Sie durch Betätigen der Taste **F11** die **SaveChanges**-Methode ausführen und schauen sich das Ergebnis im SQL Server Profiler an. Hier finden Sie die SQL-Anweisung mit dem **UPDATE**-Schlüsselwort aus Bild 3 vor.

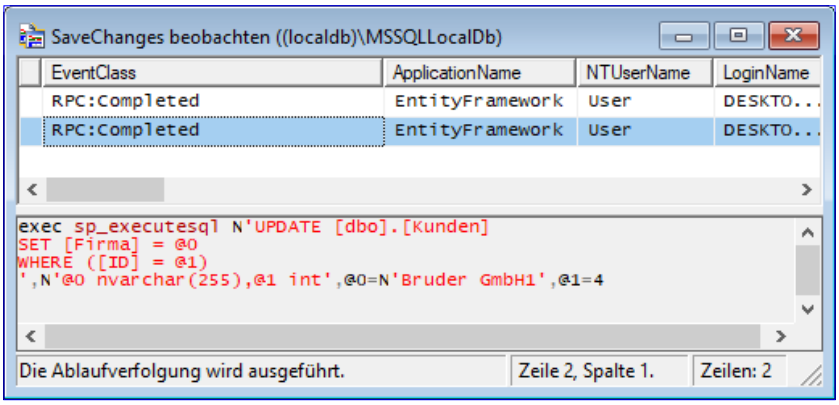


Bild 3: SQL-Anweisung zum Ändern eines Datensatzes

Die gespeicherte Prozedur, die für den zweiten **SaveChanges**-Aufruf nach dem Entfernen eines der Elemente ausgeführt wird, lautet wie folgt und enthält erwartungsgemäß das **DELETE**-Schlüsselwort:

```
exec sp_executesql N'DELETE [dbo].[Kunden]
WHERE ([ID] = @0)',N'@0 int',@0=4
```

Fehlt noch die letzte SQL-Anweisung zum Anlegen eines Datensatzes nach dem Hinzufügen eines neuen **Kunde**-Objekts:

```
exec sp_executesql N'INSERT [dbo].[Kunden]([Firma], [Vorname], [Nachname], [AnredeID], [Strasse], [PLZ], [Ort], [Land],
[EMail])
VALUES (@0, @1, @2, @3, @4, @5, @6, @7, @8)
SELECT [ID]
FROM [dbo].[Kunden]
WHERE @@ROWCOUNT > 0 AND [ID] = scope_identity()',N'@0 nvarchar(255),@1 nvarchar(255),@2 nvarchar(255),@3 int,@4 nvar-
char(255),@5 nvarchar(255),@6 nvarchar(255),@7 nvarchar(255),@8 nvarchar(255)',@0=N'Firma',@1=N'Klaus',@2=N'Müller',@3=1,
@4=N'Teststr. 1',@5=N'11111',@6=N'Testhausen',@7=N'Deutschland',@8=N'klaus@mueller.de'
```

Diese SQL-Anweisung unterscheidet sich nicht nur dadurch von den vorherigen beiden, weil sie die **INSERT INTO**-Anweisung ausführt, sondern weil sie auch noch eine **SELECT**-Abfrage auslöst. Diese dient dazu, den Primärschlüsselwert des neu hinzugefügten Datensatzes zu ermitteln, damit dieser auch dem hinzugefügten Objekt zugewiesen werden kann.

Automatisch generierte SQL-Anweisungen durch gespeicherte Prozeduren ersetzen

Nun kann es sein, dass Sie aus verschiedenen Gründen noch weitere Schritte hinzufügen wollen, wenn ein Datensatz per Entity Framework hinzugefügt, geändert oder gelöscht werden soll. Vielleicht möchten Sie etwa dafür sorgen, dass das Änderungsdatum eingetragen wird oder bei geänderten oder gelöschten Datensätzen eine Kopie der vorherigen Version des Datensatzes in einer speziellen Archivtabelle landet. Dies können Sie per Trigger erledigen, aber auch über gespeicherte Prozeduren, über die alle Aktionen wie das Anlegen, Ändern oder Löschen von Datensätzen laufen sollen.

Die Frage ist nur: Wie können Sie solche gespeicherten Prozeduren in die automatischen Abläufe des Entity Frameworks integrieren? Das ist leichter, als Sie vielleicht denken: Sie müssen lediglich in der Datenbankkontext-Klasse angeben, dass Sie die entsprechende Entität an gespeicherte Prozeduren binden wollen. Das erledigen wir in der Klasse, die in unserem Beispiel **BestellverwaltungContext** heißt, durch das Hinzufügen einer Methode namens **OnModelCreating**:

```
Public Class BestellverwaltungContext
    Inherits DbContext
    ...
    Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
        modelBuilder.Entity(Of Anrede)().MapToStoredProcedures()
        modelBuilder.Entity(Of Produkt)().MapToStoredProcedures()
        modelBuilder.Entity(Of Bestellposition)().MapToStoredProcedures()
        modelBuilder.Entity(Of Bestellung)().MapToStoredProcedures()
        modelBuilder.Entity(Of Kunde)().MapToStoredProcedures()
        modelBuilder.Entity(Of Mehrwertsteuersatz)().MapToStoredProcedures()
    End Sub
    ...
End Class
```

Wenn wir die Anwendung nun zum Debuggen starten, erhalten wir allerdings erst einmal die Fehlermeldung aus Bild 4, nach der sich das Modell seit dem Erstellen der Datenbank geändert hat. Logisch: Wir haben einige wesentliche Elemente zur Klasse

BestellverwaltungContext hinzufügt, die noch nicht in der Datenbank berücksichtigt sind – und auch noch nicht in den Klassen im Ordner **Migrations**.

Also aktivieren wir die Paketmanager-Konsole und geben dort den Befehl **Add-Migration** mit dem Namen der neuen Migration ein, hier **StoredProcedures**:

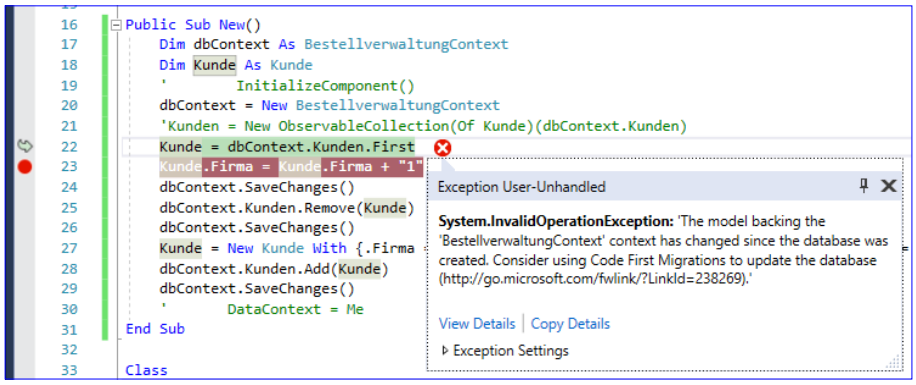


Bild 4: Fehler beim Versuch, die Anwendung zu starten

```
Add-Migration StoredProcedures
```

Dies fügt eine neue Klasse in den Ordner **Migrations** ein, der in der Methode **Up** für jede weiter oben mit der Methode **MapToStoredProcedures** versehene Entität die Anweisungen zum Erstellen von gespeicherten Prozeduren zum Einfügen, Bearbeiten und Löschen hinzufügt. Die Methode **Down** enthält für jede dieser gespeicherten Prozeduren die Anweisung zum Löschen. Für die einfache Entität **Anrede** sehen die Anweisungen zum Erstellen etwa wie folgt aus.

Die erste **CreateStoredProcedure**-Anweisung erstellt die gespeicherte Prozedur **dbo.Anrede_Insert**, die den Parameter **Name** für das gleichnamige Feld der Tabelle erwartet. Der erste Parameter der **CreateStoredProcedure**-Anweisung erwartet den Namen der gespeicherten Prozedur, der zweite einen Ausdruck, der die Parameter definiert und der dritte den SQL-Code, der für die gespeicherte Prozedur angelegt werden soll. Im Fall der gespeicherten Prozedur zum Einfügen eines neuen Datensatzes enthält diese eine **INSERT INTO**-Anweisung sowie zwei **SELECT**-Anweisungen, welche zum Ermitteln der ID des neuen Datensatzes genutzt werden:

```
Public Overrides Sub Up()
    CreateStoredProcedure(
        "dbo.Anrede_Insert",
        Function(p) New With
            { .Name = p.String(maxLength := 255) },
        body :=
            "INSERT [dbo].[Anreden]([Name])" & vbCrLf & _
            "VALUES (@Name)" & vbCrLf & "" & vbCrLf & _
            "DECLARE @ID int" & vbCrLf & _
            "SELECT @ID = [ID]" & vbCrLf & _
            "FROM [dbo].[Anreden]" & vbCrLf & _
            "WHERE @@ROWCOUNT > 0 AND [ID] = scope_identity()" & vbCrLf & "" & vbCrLf & _
            "SELECT t0.[ID]" & vbCrLf & _
            "FROM [dbo].[Anreden] AS t0" & vbCrLf & _
            "WHERE @@ROWCOUNT > 0 AND t0.[ID] = @ID"
    )
```

Der zweite Aufruf der Methode **CreateStoredProcedure** soll die gespeicherte Prozedur **Anrede_Update** anlegen. Dieser Aufruf soll eine gespeicherte Prozedur mit zwei Parametern anlegen (**ID** und **Name**) und schließlich die entsprechende **UPDATE**-Anweisung ausführen:

```
    CreateStoredProcedure(
        "dbo.Anrede_Update",
        Function(p) New With
            {
                .ID = p.Int(),
                .Name = p.String(maxLength := 255)
            },
        body :=
            "UPDATE [dbo].[Anreden]" & vbCrLf & _
            "SET [Name] = @Name" & vbCrLf & _
            "WHERE ([ID] = @ID)"
    )
```

Es fehlt noch der dritte Aufruf von **CreateStoredProcedure**, der die **DELETE**-Prozedur anlegt, die nur die **ID** des zu löschenden Datensatzes als Parameter benötigt:

```
    CreateStoredProcedure(
        "dbo.Anrede_Delete",
        Function(p) New With
            { .ID = p.Int() },
        body :=
            "DELETE [dbo].[Anreden]" & vbCrLf & _
            "WHERE ([ID] = @ID)"
    )
End Sub
```

Gespeicherte Prozeduren anlegen

Nachdem diese Klasse mit den **Up**- und **Down**-Methoden zum Anlegen der gespeicherten Prozeduren erstellt wurde, wollen wir diese auch aufrufen. Dazu geben Sie in der Paketmanager-Konsole den Befehl **Update-Database** ein.

Danach können Sie den SQL Server Objekt Explorer aktivieren, indem Sie den entsprechenden Eintrag im Ansicht-Menü anklicken. Wenn Sie in der Zieldatenbank zum Element **Programmability|Stored Procedures** (in der deutschen Version

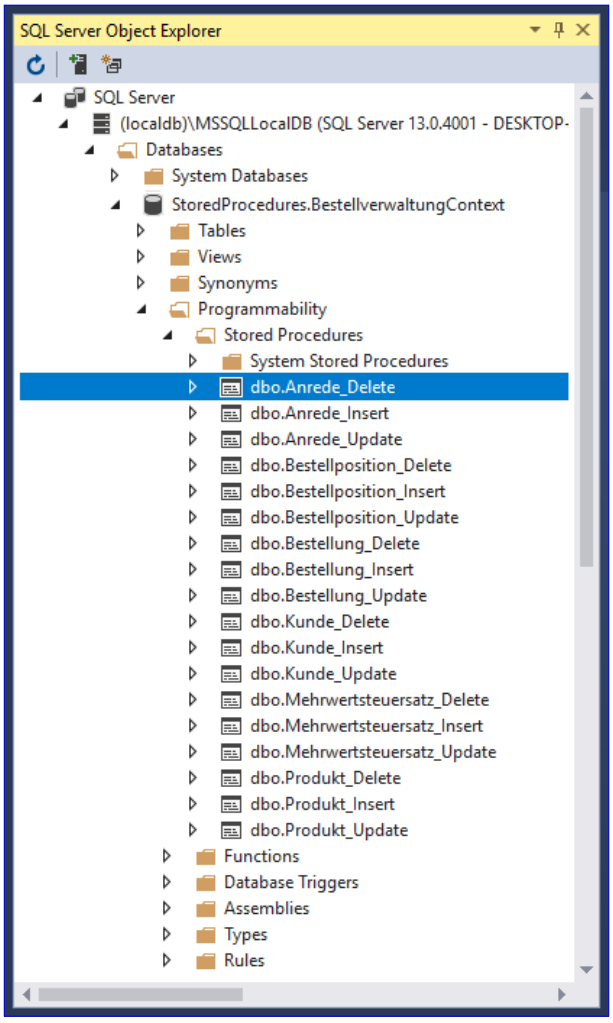


Bild 5: Die neu hinzugefügten gespeicherten Prozeduren