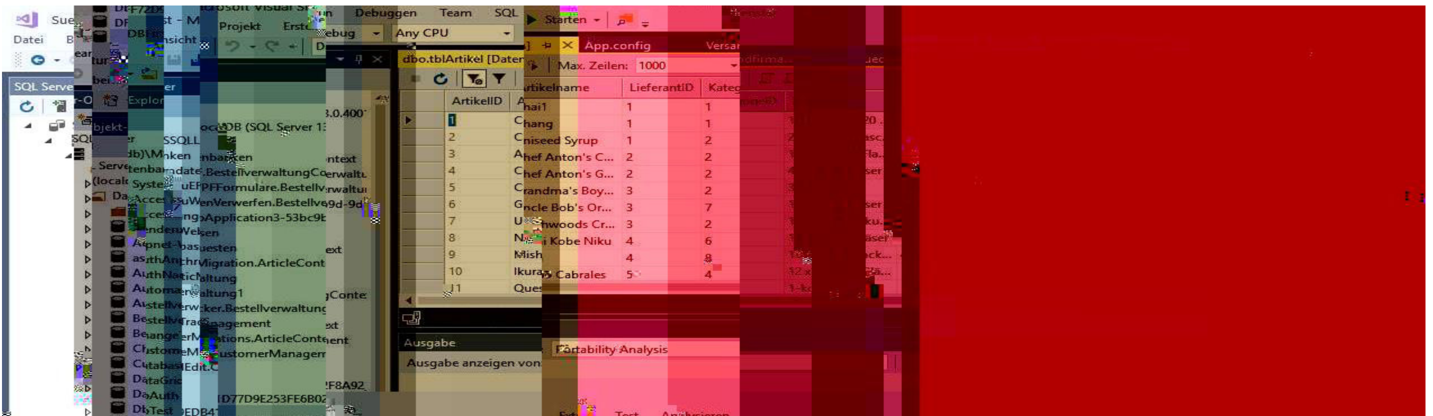
AUSGABE 02-03/2019

DATENBANK ENTWICKLER

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



TOP-THEMEN:

STEUERELEMENTE	Mehrspaltige Kombinationsfelder	SEITE 14
USERINTERFACE	Entity Framework: Bilder in WPF	SEITE 23
EF	Blättern in Datensätzen	SEITE 53
EF	Änderungen erkennen und verwerfen	SEITE 88
ACCESS ZU .NET	Detailformulare mit Combo, Checkbox und Button	SEITE 96

André Minhorst Verlag

datenbankentwickler.net

VB-GRUNDLAGEN	Bytes im Griff mit der Stream-Klasse: FileStream	3
WPF-GRUNDLAGEN	WPF: DataTrigger und MultiDataTrigger	6
BENUTZEROBERFLÄCHE MIT WPF	Mehrspaltige Kombinationsfelder	14
	Entity Framework: Bilder in WPF	23
ENTITY FRAMEWORK	EDM für bestehende Datenbank mit Code First	39
	EDM: Blättern in Datensätzen	53
	Validieren mit VB und EDM	63
	Entity Framework: Der ChangeTracker	77
BENUTZEROBERFLÄCHE MIT WPF	Änderungen erkennen und verwerfen	88
ACCESS ZU WPF	Detailformulare mit Combo, Checkbox und Button	96
	Access zu WPF: Validierung und Navigation	116
SERVICE	Impressum	2

DOWNLOAD

Die Downloads zu dieser Ausgabe finden Sie unter folgendem Link:

http://www.amvshop.de

Klicken Sie dort auf **Mein Konto**, loggen Sie sich ein und wählen dann **Meine Sofortdownloads**.

Impressum

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

Redaktion: Dipl.-Ing. André Minhorst

Das Magazin und alle darin enthaltenen Beiträge und Abbildungen sind urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmung und für die Einspeicherung in elektronische Systeme.
Wir weisen darauf hin, dass die verwendeten Bezeichnungen und Markennamen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen. Die im Werk gemachten Angaben erfolgen nach bestem Wissen, jedoch ohne Gewähr. Für mögliche Schäden, die im Zusammenhang mit den Angaben im Werk stehen könnten, wird keine Gewährleistung übernommen.

Bytes im Griff mit der Stream-Klasse: FileStream

Die Stream-Klasse ist die Basisklasse für verschiedene Stream-Klassen wie FileStream oder MemoryStream. Streams bieten die Möglichkeit, Byte-genau schreibend wie lesend auf Daten aus Dateien, dem Arbeitsspeicher oder dem Netzwerk zuzugreifen. Dieser Artikel zeigt, welche Stream-Klassen es gibt und geht genauer auf die FileStream-Klasse ein. Dabei zeigen wir, wie Sie mit dieser Klasse lesend und schreibend auf den Inhalt von Dateien zugreifen und den Inhalt mit Byte-Arrays austauschen.

Verschiedene Stream-Klassen

Es gibt folgende von der **Stream**-Klasse abgeleitete Klassen:

- **BufferedStream**: Puffert Daten aus E/A-Datenströmen im Arbeitsspeicher.
- **CryptoStream**: Verschlüsselt Daten.
- **FileStream**: Schreibt Daten in Dateien im Dateisystem.
- **GZipStream**: Komprimiert und dekomprimiert Streams.
- **MemoryStream**: Schreibt einen Stream in den Hauptspeicher statt in eine temporäre Datei.
- **NetworkStream**: Erlaubt den Zugriff auf Netzwerkressourcen.

Wir wollen uns in diesem Artikel auf die **FileStream**- und die **MemoryStream**-Klasse konzentrieren.

Die FileStream-Klasse

Mit der **FileStream**-Klasse können Sie byteweise aus Dateien lesen und in Dateien schreiben sowie einen Positionszeiger in einen Stream setzen. Der FileStream puffert dabei die Daten (bis zu acht Kilobyte). Wenn Sie ein Objekt auf Basis der **FileStream**-Klasse erzeugen wollen, können Sie per Konstruktor verschiedene Parameter übergeben:

- **path**: Pfad der zu lesenden oder schreibenden Datei
- **mode**: Modus für den Dateizugriff. Die Enumeration **FileMode** bietet folgende Werte: **Append**, **Create**, **CreateNew**, **Open**, **OpenOrCreate**, **Truncate**
- **access**: Art des Dateizugriffs mit folgenden Werten der Enumeration **FileAccess**: **Read**, **ReadWrite**, **Write**
- **share**: Gibt an, ob gemeinsamer Zugriff auf die Datei möglich ist. Die Enumeration **FileShare** liefert dazu die folgenden möglichen Werte: **Delete**, **Inheritable**, **None**, **Read**, **ReadWrite**, **Write**

- **bufferSize**: Ein **int32**-Wert größer, der die Buffer-Größe angibt. Standardwert ist 4.096.

- **isAsync**: **Boolean**-Wert, der angibt, ob asynchrone Zugriffe möglich sein sollen.

Byte-Array in eine Datei schreiben

Die folgenden Anweisungen schreiben ein Byte-Array in eine neue Datei. Zuerst deklarieren wir die nötigen Elemente, legen den Pfad fest und erstellen das Array mit Zahlenwerten:

```
Dim objFileStream As FileStream
Dim strPath As String
Dim byt As Byte()
strPath = "C:\...\Stream.txt"
byt = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Dann erstellen wir das **FileStream**-Objekt mit dem Pfad und mit dem Modus **Create**, was bedeutet, dass die Datei neu erstellt wird, auch wenn sie bereits existiert:

```
objFileStream = New FileStream(strPath, FileMode.Create)
```

Schließlich schreiben wir mit der **Write**-Methode den Inhalt des Arrays ab der Position **0** der Datei bis zur Position, die durch die Länge des Arrays angegeben wird, und schließen die Datei wieder:

```
With objFileStream
    .Write(byt, 0, byt.Length)
    .Close
End With
```

Byte-Array aus einer Datei lesen

Die folgenden Codezeilen erlauben das Lesen eines Byte-Arrays aus einer Datei. Auch hier deklarieren wir wieder einige Variablen, wobei wir hier direkt die Größe des Byte-Arrays angeben:

```
Dim objFileStream As FileStream
Dim strPath As String
Dim byt(9) As Byte
strPath = "C:\Users\User\Dropbox\Daten\Fachmagazine\VisualStudioDatenbankentwicklung\2019\02\Streams\Stream.txt"
Dim l As Long
```

Das **FileStream**-Objekt erstellen wir diesmal mit dem Wert **FileMode.Open** als zweiten Parameter.

Damit öffnen wir die Datei zum Lesen. Dann greifen wir mit der **Read**-Methode auf die Datei zu und lesen in das Array **byt** von der ersten Position mit der Länge des **FileStream**-Objekts ein:

WPF: DataTrigger und MultiDataTrigger

Um die Eigenschaftswerte von Steuerelementen abhängig von den Werten anderer Steuerelemente oder auch von Eigenschaften einzustellen, verwenden Sie DataTrigger oder MultiDataTrigger. Wir schauen uns in diesem Artikel die Grundlagen zu diesen Elementen an und zeigen anhand einiger Beispiele, was beim Praxiseinsatz zu beachten ist. Dabei betrachten wir auch, wie Sie DataTrigger auf Basis von Eigenschaften im Code behind-Modul des WPF-Objekts verwenden können.

Eigenschaften per VB einstellen

Wer von Access kommt, kennt nur zwei Methoden, um die Eigenschaften eines Steuerelements einzustellen: über das Eigenschaftsfenster oder per VBA. Beide Methoden sind beliebig miteinander kombinierbar. Sie können einen initialen Wert über das Eigenschaftsfenster einstellen oder dies per VBA erledigen, etwa in der Ereignisprozedur **Form_Load** des Formulars. Nachträgliche Einstellungen sind allerdings nur per VBA möglich.

Unter WPF/VB gibt es umfangreichere Möglichkeiten. Sie können eine Eigenschaft natürlich explizit im WPF-Code eines Elements festlegen – um etwa ein Steuerelement gleich beim Anzeigen des Fensters zu deaktivieren, stellen Sie die Eigenschaft einfach wie folgt ein (siehe Fenster **EigenschaftPerWPF.xaml**):

```
<Button x:Name="btnBeimAnzeigenDeaktiviert" ... IsEnabled="False" />
```

Sie können dem Fenster dann zwei Schaltflächen hinzufügen, mit deren Ereignismethoden für das Attribut **Click** die Eigenschaft **IsEnabled** auf **True** oder **False** eingestellt wird:

```
Private Sub btnAktivieren_Click(sender As Object, e As RoutedEventArgs)
    btnBeimAnzeigenDeaktiviert.IsEnabled = True
End Sub

Private Sub btnDeaktivieren_Click(sender As Object, e As RoutedEventArgs)
    btnBeimAnzeigenDeaktiviert.IsEnabled = False
End Sub
```

Das Ergebnis sieht dann wie in Bild 1 aus.

Die nächste Möglichkeit ist, diese Eigenschaft an eine öffentliche Eigenschaft im Code behind-Modul zu koppeln.

Eigenschaft an Code behind-Eigenschaft koppeln

Sie können die Eigenschaft **IsEnabled** auch an eine öffentlich deklarierte Eigenschaft im Code behind-Modul koppeln (siehe

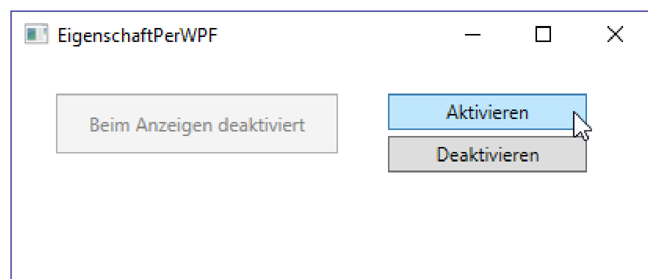


Bild 1: Aktivieren und deaktivieren per Schaltfläche und VB-Prozedur

Fenster **EigenschaftPerCodeBehind.xaml**). Man könnte naiverweise meinen, dass folgender Code im Code behind-Modul ausreicht:

```
Public Property Aktiviert As Boolean

Private Sub btnAktivieren_Click(sender As Object, e As RoutedEventArgs)
    Aktiviert = True
End Sub

Private Sub btnDeaktivieren_Click(sender As Object, e As RoutedEventArgs)
    Aktiviert = False
End Sub
```

Dabei wollen wir voraussetzen, dass die Schaltfläche über die Eigenschaft **IsEnabled** an die Eigenschaft **Aktiviert** gebunden ist:

```
<Button x:Name="btnBeimAnzeigenDeaktiviert" Content="Beim Anzeigen deaktiviert" IsEnabled="{Binding Aktiviert}" ... />
```

Das reicht allerdings nicht aus, denn beim Anklicken der beiden Schaltflächen **btnAktivieren** und **btnDeaktivieren** geschieht nichts. Das ändert sich auch nicht, wenn wir die Schnittstelle **INotifyPropertyChanged** implementieren, die notwendig ist, um Aktualisierungen von Eigenschaften über die Bindung in die Benutzeroberfläche zu übertragen. Zusätzlich benötigen wir nämlich auch noch eine Konstruktor-Methode, die beim Erstellen des Fensters aufgerufen wird und dem WPF-Fenster mitteilt, dass es an die Eigenschaften des Code behind-Moduls gebunden werden soll. Insgesamt sieht der Code für das Code behind-Modul nun wie folgt aus. Als Erstes fügen wir einen Verweis auf den Namespace **System.ComponentModel** hinzu, den wir für die **INotifyPropertyChanged**-Schnittstelle benötigen:

```
Imports System.ComponentModel
```

Dann fügen wir die **Implements**-Anweisung zur Implementierung der Schnittstelle hinzu sowie die Eigenschaft **Aktiviert**, in der wir die Schnittstelle nutzen wollen:

```
Public Class EigenschaftPerCodeBehind
    Implements INotifyPropertyChanged
    Private _Aktiviert As Boolean
    Public Event PropertyChanged As PropertyChangedEventHandler Implements INotifyPropertyChanged.PropertyChanged

    Public Property Aktiviert As Boolean
        Get
            Return _Aktiviert
        End Get
        Set
```

```
            _Aktiviert = Value
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("Aktiviert"))
        End Set
    End Property
```

Fehlt noch die Konstruktor-Anweisung, in der wir mit **DataContext = Me** angeben, dass die öffentlichen Eigenschaften des Code behind-Moduls vom XAML-Code aus gebunden werden können:

```
    Public Sub New()
        InitializeComponent()
        DataContext = Me
        Aktiviert = False
    End Sub
End Class
```

Das allein reicht allerdings noch nicht aus. Wir müssen auch noch die Eigenschaft **IsEnabled="{Binding Aktiviert}"** ersetzen, denn für die Bindung an eine Eigenschaft einer anderen Klasse ist ein DataTrigger nötig. Diesen hinterlegen wir wie folgt für die Schaltfläche **btnAktivierenDeaktivieren**:

```
<Button x:Name="btnAktivierenDeaktivieren" Content="Aktivieren/Deaktivieren" HorizontalAlignment="Left" Height="38"
        Margin="28,22,0,0" VerticalAlignment="Top" Width="180">
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <Setter Property="IsEnabled" Value="False"></Setter>
            <Style.Triggers>
                <DataTrigger Binding="{Binding Aktiviert}" Value="True">
                    <Setter Property="IsEnabled" Value="True" />
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </Button.Style>
</Button>
```

Hier sind gleich mehrere wichtige Informationen untergebracht:

- Mit dem ersten **Setter**-Element legen wir für die Eigenschaft **IsEnabled** den Wert **False** als initialen Wert fest. Das ist die Methode, den Standardwert zu definieren, wenn Sie einen DataTrigger verwenden.

- Im **DataTrigger**-Element definieren wir, welchen Wert wir untersuchen wollen, hier die gebundene Eigenschaft **Aktiviert** für das Attribut **Property**, und mit welchem Wert wir den gelieferten Wert vergleichen wollen – hier **True** für das Attribut **Value**.

Ist diese Bedingung erfüllt, also ist der Wert der Eigenschaft **Aktiviert** gleich **True**, wird die im untergeordneten Setter-Element angegebene Eigenschaft mit dem dort angegebenen Wert gefüllt.

Achtung: Wenn Sie den Standardwert bei Verwendung eines DataTriggers mit WPF-Mitteln einstellen wollen, gelingt dies nur mit dem erwähnten **Setter**-Element – das Einstellen der Eigenschaft als Attribut des hier verwendeten **Button**-Elements ist nicht möglich (also etwa mit **IsEnabled = False**). Wenn Sie das dennoch tun, kann der Wert durch den DataTrigger nicht mehr geändert werden!

Mehrere Bedingungen verwenden

Wie gehen wir nun vor, wenn wir mehrere Bedingungen verwenden wollen? Wir starten mit zwei Oder-verknüpften Bedingungen. Dazu fügen wir zu einem neuen Fenster (siehe **EigenschaftMitOder.xaml**) die folgenden beiden Kontrollkästchen hinzu:

```
<CheckBox x:Name="chkBedingung1" Content="Bedingung 
1" HorizontalAlignment="Left" Margin="239,23,0,0" 
VerticalAlignment="Top"/>
<CheckBox x:Name="chkBedingung2" Content="Bedingung 
2" HorizontalAlignment="Left" Margin="239,43,0,0" 
VerticalAlignment="Top"/>
```

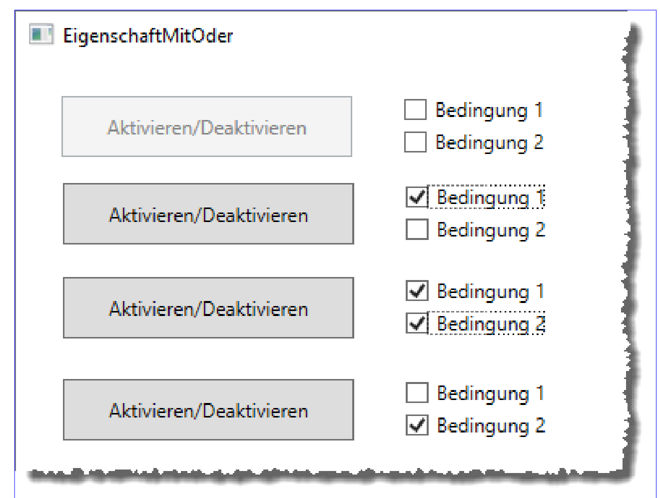


Bild 2: Bedingung für das Aktivieren mit Oder

Diese sollen nun Oder-verknüpft dafür sorgen, dass bei mindestems einem aktivierten Kontrollkästchen die Schaltfläche **btnAktivierenDeaktivieren** aktiviert wird – so wie in Bild 2.

Statt eines **DataTrigger**-Elements verwenden wir nun gleich zwei – einen für jede Bedingung. Wir stellen den Standardwert für die Eigenschaft **IsEnabled** wieder auf **False** ein. Die beiden **DataTrigger**-Elemente prüfen jeweils den Wert der Eigenschaft **IsChecked** eines der Kontrollkästchen und stellen dann den Wert des Attributes **IsEnabled** auf **True** ein. Dadurch, dass die **DataTrigger** nacheinander durchlaufen werden, reicht es aus, wenn einer der **DataTrigger** ausgelöst wird. Somit arbeiten die Trigger wie eine Oder-Verknüpfung:

```
<Button x:Name="btnAktivierenDeaktivieren" Content="Aktivieren/Deaktivieren" ...>
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <Setter Property="IsEnabled" Value="False"/>
            <Style.Triggers IsEnabled="Test">
                <DataTrigger Binding="{Binding ElementName=chkBedingung1, Path=IsChecked}" Value="True">
                    <Setter Property="IsEnabled" Value="True" />
                </DataTrigger>
                <DataTrigger Binding="{Binding ElementName=chkBedingung2, Path=IsChecked}" Value="True">
                    <Setter Property="IsEnabled" Value="True" />
```

Mehrspaltige Kombinationsfelder

Manchmal möchte man mit einem Kombinationsfeld nicht nur den Wert eines Feldes anzeigen, sondern gleich mehrere. Unter Access war das zumindest für die Auswahlliste schnell erledigt – Sie brauchten einfach nur die Eigenschaften Spaltenanzahl und Spaltenbreiten anzupassen. Wenn der Inhalt des Kombinationsfeldes auch die Daten mehrerer Felder liefern sollte, mussten Sie diese per Zeichenkettenfunktion zusammenführen. Wie aber sieht das im WPF-Steuerelement »ComboBox« aus? Dieser Artikel zeigt, welche Möglichkeiten dieses Steuerelement für mehrspaltige Einträge bietet.

Vorbereitung

Für das Beispiel verwenden wir die Basis vieler anderer Artikel: Wir verwenden die aus einer Access-Datenbank erzeugten Entitätsklassen plus den von dort exportierten Daten. Sie brauchen, wenn Sie das Beispielprojekt verwenden wollen, einfach nur die Paket-Manager-Konsole anzuzeigen und den folgenden Befehl einzugeben:

```
Update-Database
```

Das fügt dann eine neue Datenbank entsprechend der Verbindungszeichenfolge aus der Datei **App.config**, die Sie gegebenenfalls noch anpassen müssen, zum SQL Server beziehungsweise zu **LocalDb** hinzu.

Im Code behind-Modul fügen wir folgenden Code hinzu:

```
Imports System.Collections.ObjectModel

Class MainWindow
    Private m_Kunden As List(Of Kunde)
    Private dbContext As BestellverwaltungContext
    Public Sub New()
        InitializeComponent()
        dbContext = New BestellverwaltungContext
        Kunden = New List(Of Kunde)(dbContext.Kunden)
        DataContext = Me
    End Sub

    Public Property Kunden As List(Of Kunde)
        Get
            Return m_Kunden
        End Get
        Set(value As List(Of Kunde))
            m_Kunden = value
```

```
        End Set
    End Property
End Class
```

Danach fügen wir das folgende **ComboBox**-Element zum Startfenster **MainWindow.xaml** hinzu (siehe **cboKundeID_EinFeld**):

```
<ComboBox x:Name="cboKundeID_EinFeld" Padding="0" Height="21"
Margin="118,111,0,0" Width="78"
    ItemsSource = "{Binding Kunden}">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <TextBlock>
                <TextBlock.Text>
                    <MultiBinding StringFormat="{}{0}">
                        <Binding Path="Nachname" />
                    </MultiBinding>
                </TextBlock.Text>
            </TextBlock>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

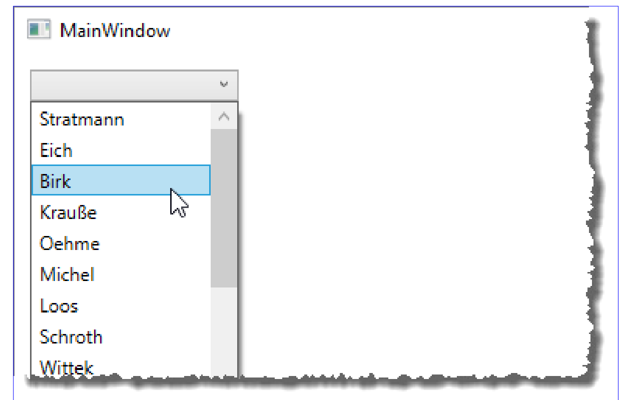


Bild 1: Kombinationsfeld mit einer Spalte

Damit erhalten wir das Kombinationsfeld aus Bild 1. Um den Inhalt eines zweiten Feldes als ausgewählten Eintrag anzuzeigen wie etwa **Vorname**, passen Sie das **MultiBinding**-Element einfach wie folgt an (siehe):

```
<MultiBinding StringFormat="{}{0},{1}">
    <Binding Path="Nachname" />
    <Binding Path="Vorname" />
</MultiBinding>
```

Das Ergebnis sieht dann wie in Bild 2 aus. Sowohl der ausgewählte Eintrag als auch die Einträge der Liste werden dann im gewünschten Format **<Nachname>,<Vorname>** angezeigt.

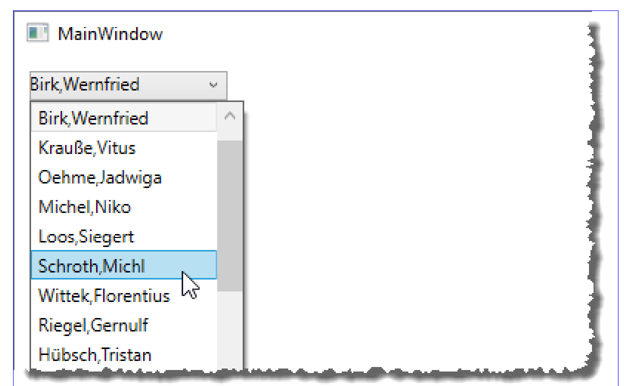


Bild 2: Kombinationsfeld mit einer Spalte und zwei kombinierten Feldern

Mehrspaltige Anzeige

Um die Anzeige nun auch noch auf mehrere Spalten aufzuteilen, verwenden Sie ein etwas aufwendigeres Konstrukt. Dabei weisen wir dem Unterelement **ComboBox.ItemTemplate** das **DataTemplate**-Element zu, das ein **StackPanel** mit den drei **TextBlock**-Elementen für unsere drei Spalten aufnimmt (siehe **cboKunden_MehrereSpaltenNichtBuendig**):

```
<ComboBox x:Name="cboKunden_MehrereSpaltenNichtBuendig" ItemsSource="{Binding Kunden}" Margin="157,10,311,381" >
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Margin="5" Grid.Column="0" Text="{Binding Vorname}"></TextBlock>
                <TextBlock Margin="5" Grid.Column="1" Text="{Binding Nachname}"></TextBlock>
                <TextBlock Margin="5" Grid.Column="2" Text="{Binding EMail}"></TextBlock>
            </StackPanel>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

Wenn wir das Beispielprojekt nun starten, zeigt die Liste bereits die gewünschten Daten an (siehe Bild 3).

Allerdings bleibt das Kombinationsfeld nach der Auswahl eines der Einträge leer. Außerdem werden die Elemente der drei Spalten nicht bündig angezeigt.

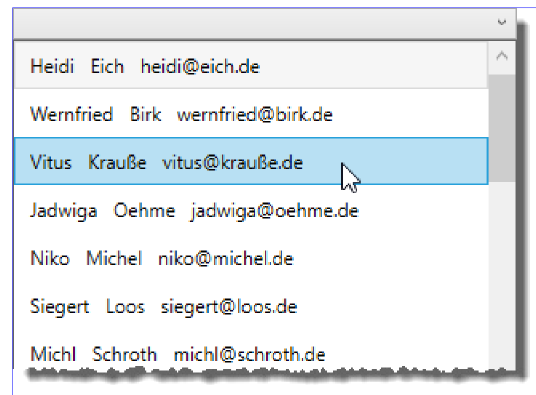


Bild 3: Kombinationsfeld mit drei Spalten, die allerdings nicht bündig sind

ComboBox-Spalten per Grid

Also versuchen wir eine Alternative. Diese verwendet innerhalb des **DataTemplate**-Elements ein **Grid** mit den **ColumnDefinition**-Elementen für die drei Spalten, in denen wir feste Spaltenbreiten definieren (siehe **cboKunden_MehrereSpaltenFixeBreite**):

```
<ComboBox x:Name="cboKunden_MehrereSpaltenFixeBreite" ItemsSource="{Binding Kunden}" Margin="157,10,311,381">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="100"></ColumnDefinition>
                    <ColumnDefinition Width="100"></ColumnDefinition>
                    <ColumnDefinition Width="150"></ColumnDefinition>
                </Grid.ColumnDefinitions>
                <TextBlock Margin="5" Grid.Column="0" Text="{Binding Vorname}"></TextBlock>
                <TextBlock Margin="5" Grid.Column="1" Text="{Binding Nachname}"></TextBlock>
                <TextBlock Margin="5" Grid.Column="2" Text="{Binding EMail}"></TextBlock>
            </Grid>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

Das Ergebnis liefert nun immerhin feste Spaltenbreiten für alle Listeneinträge. Allerdings sind diese nun natürlich fix vorgegeben und können sich nicht an die Inhalte der drei Spalten anpassen (siehe Bild 4).

Stellen wir die Spaltenbreiten testweise auf dynamische Spaltenbreiten um, bei der die Spaltenbreiten an die Inhalte angepasst werden:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"></ColumnDefinition>
    <ColumnDefinition Width="*"></ColumnDefinition>
    <ColumnDefinition Width="*"></ColumnDefinition>
</Grid.ColumnDefinitions>
```

Dies liefert allerdings das Ergebnis aus Bild 5 – soweit waren wir schon.

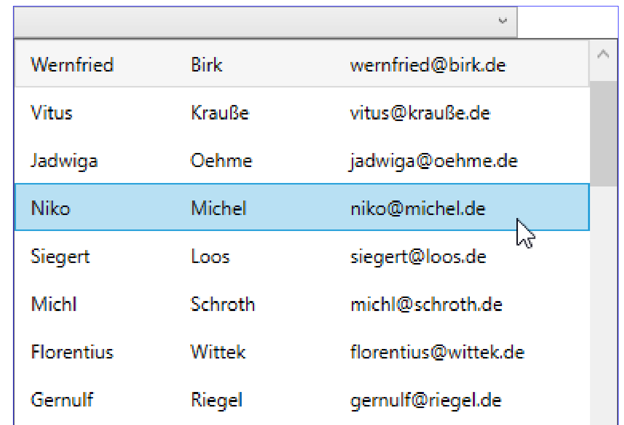


Bild 4: Kombinationsfeld mit drei Spalten mit fixen Spaltenbreiten

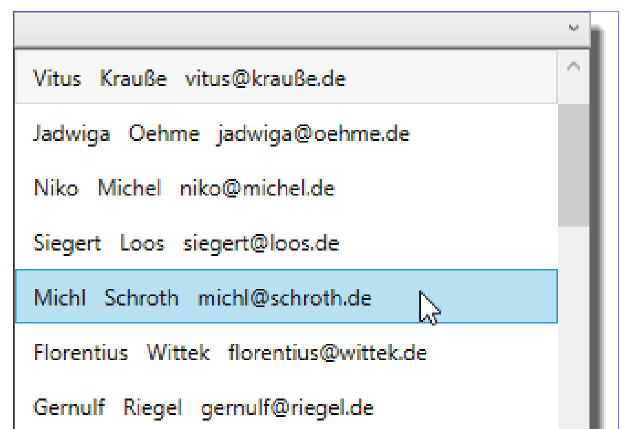


Bild 5: Kombinationsfeld mit drei Spalten mit dynamischen Spaltenbreiten

Spalten einheitlich mit SharedSizeScope

Es gibt jedoch eine Möglichkeit, die Spalten dynamisch, aber dennoch einheitlich zu gestalten. Dabei legen wir für das **ComboBox**-Element die Eigenschaft **IsSharedSizeScope** mit dem Wert **True** fest. Außerdem müssen wir noch für die drei **ColumnDefinition**-Elemente das Attribut **SharedSizeGroup** auf jeweils unterschiedliche Werte einstellen

```
<ComboBox x:Name="cboKunden_MehrereSpaltenBuendigFlexibleBreite" ItemsSource="{Binding Kunden}" Margin="157,10,311,381"
Grid.IsSharedSizeScope="True">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" SharedSizeGroup="Column1"></ColumnDefinition>
                    <ColumnDefinition Width="Auto" SharedSizeGroup="Column2"></ColumnDefinition>
                    <ColumnDefinition Width="Auto" SharedSizeGroup="Column3"></ColumnDefinition>
                </Grid.ColumnDefinitions>
                <TextBlock Margin="5" Grid.Column="0" Text="{Binding Vorname}"></TextBlock>
                <TextBlock Margin="5" Grid.Column="1" Text="{Binding Nachname}"></TextBlock>
                <TextBlock Margin="5" Grid.Column="2" Text="{Binding EMail}"></TextBlock>
            </Grid>
        </DataTemplate>
```

Entity Framework: Bilder in WPF

Im Artikel »PowerApps: Bilder in Datenbank speichern« haben wir gezeigt, wie Sie mit einer PowerApp über ein Smartphone oder ein Tablet Bilder aufnehmen und diese dann in einer Datenbank speichern. Nun wollen wir uns ansehen, wie Sie diese Bilder aus der Datenbank in einer WPF-Anwendung anzeigen, die über das Entity Framework auf die Tabellen der Datenbank zugreift.

Voraussetzung

Um die Beispiele dieses Artikels nachzuvollziehen, benötigen Sie eine Datenbank, die eine Tabelle namens **Fotos** enthält – genau wie die, die wir im oben genannten Artikel erstellt haben. Diese enthält ein Primärschlüsselfeld namens **ID** und mit dem Datentyp **int** sowie ein Feld namens **Foto** und mit dem Datentyp **image**. Unsere Beispieldatenbank liegt auf einem Azure-Server. Sie können aber natürlich auch unsere Beispieldatenbank mit ein paar Beispielfotos aus dem SQL-Skript aus dem Download auf einem lokalen SQL Server anlegen und mit der folgenden Anwendung auf diese Datenbank zugreifen.

Vorbereitung

Wir erstellen in Visual Studio ein neues Projekt auf Basis der Vorlage **Visual Basic|Windows Desktop|WPF-App**. Da wir diesmal nicht, wie so oft in den vorherigen Ausgaben, ein Code First-Model erstellen, sondern mit einer bestehenden Datenbank starten, noch einmal eine kurze Beschreibung der nötigen Schritte. Dem neuen Projekt fügen wir als Erstes ein neues Element hinzu und wählen den Typ **ADO.NET Entity Data Model** aus. Nachdem wir als Name **FotoverwaltungContext** eingestellt haben, klicken wir auf **Hinzufügen** (siehe Bild 1).

Danach wählen wir als Modellinhalt **Code First aus**

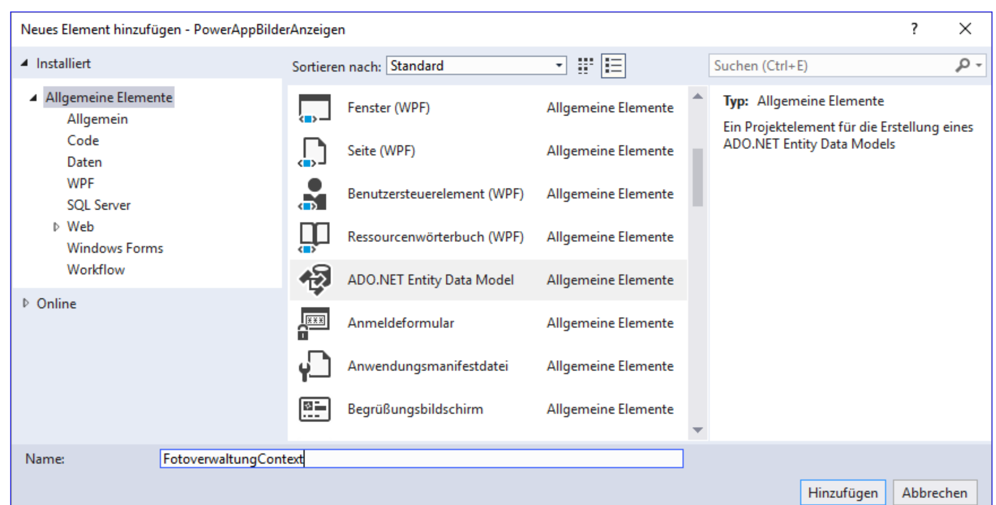


Bild 1: Einfügen des Entity Data Models

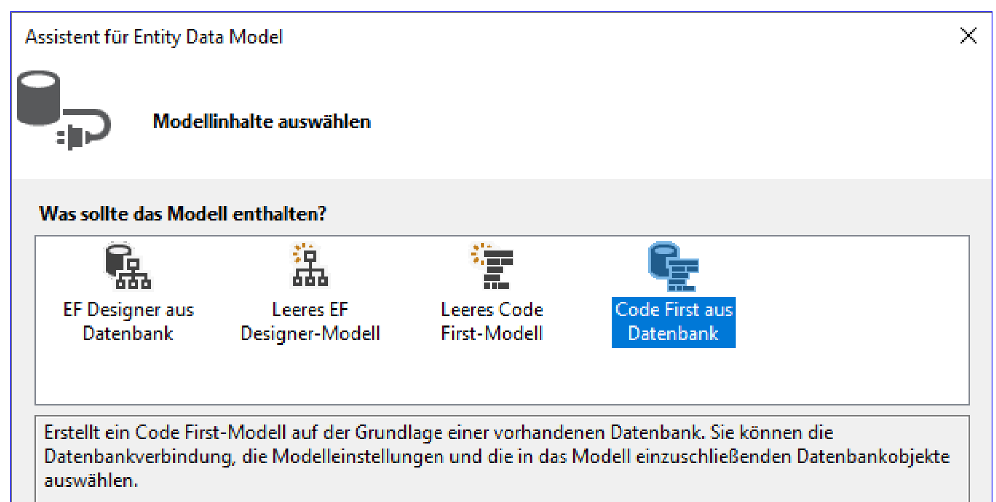


Bild 2: Auswahl des Typs **Code First aus Datenbank**

Datenbank aus (siehe Bild 2). Der folgende Dialog bietet die Möglichkeit zur Auswahl der Datenverbindung. Hier klicken wir auf die Schaltfläche **Neue Verbindung**. Im nun erscheinenden Dialog namens **Verbindungseigenschaften** geben Sie unter **Servername** den Namen des gewünschten Servers ein. Danach wählen Sie die Authentifizierungsmethode aus – beim Zugriff auf eine Azure-Datenbank immer die SQL Server-Authentifizierung – und geben die Zugangsdaten ein.

Schließlich wählen Sie die Datenbank aus und bestätigen nach einem erfolgreichen Test mit einem Klick auf die Schaltfläche **Testverbindung** die Eingaben mit **OK** (siehe Bild 3).

Danach kehren wir zum Dialog Assistent für Entity Data Model zurück, der Sie fragt, ob Sie Benutzername und Kennwort wirklich in der Verbindungszeichenfolge speichern wollen (siehe Bild 4). Für dieses Beispiel lassen wir diese Daten in der Verbindungszeichenfolge, aber im Artikel **Zugangsdaten für SQL Server abfragen** zeigen wir, wie Sie diese jeweils beim Start der Anwendung abfragen können.

Schließlich wählen wir im Schritt **Wählen Sie Ihre Datenbankobjekte und Einstellungen** noch die Tabelle aus, deren Daten wir in der Anwendung anzeigen wollen. Wir haben die Tabelle **tblFotos** einfach in eine bestehende Datenbank eingefügt, daher müssen wir diese einzeln auswählen (siehe Bild 5).

Dieser Dialog bietet auch noch die Option **Generierte Objektnamen in den Singular oder Plural setzen** an. Da die Tabelle das Präfix **tbl** im Namen **tblFotos** enthält, das wir in unserem Entity Data Model nicht als Bezeichnung der Entität sehen wollen, müssen wir hier ohnehin noch ein Mapping vornehmen.

Bild 3: Auswahl der Datenquelle

Bild 4: Einstellen weiterer Eigenschaften

Nach einem Klick auf die Schaltfläche **Fertigstellen** finden Sie zwei neue Elemente im Projektmappen-Explorer vor: die Klasse **FotoverwaltungContext.vb**, die von der Klasse **DbContext** erbt und unter anderem das **DbSet** namens **tblFotos** für den Zugriff auf die Liste der Entitäten auf Basis der in der Tabelle gespeicherten Daten bereitstellt sowie die Klasse **tblFotos.vb** als Entitätsklasse für einen Datensatz der Tabelle **tblFotos**.

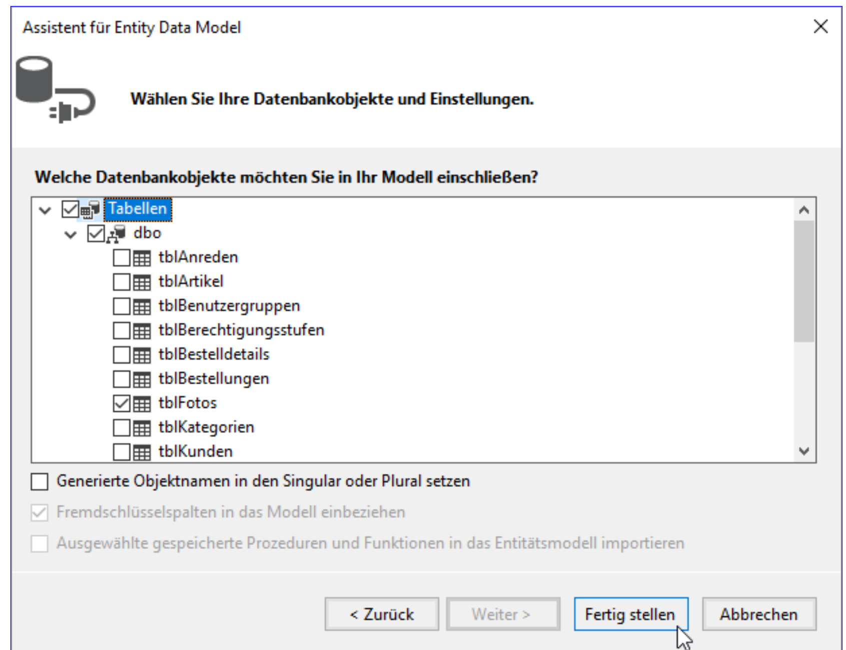


Bild 5: Auswahl der gewünschten Tabelle

Die Benennung der Klasse und des Namens des **DbSet**-Elements entspricht nicht unseren Vorstellungen, also ändern wir das ein wenig ab. Wie das genau gelingt, zeigen wir Ihnen im Artikel **EDM für bestehende Datenbank mit Code First**. Zum schnellen Nachbauen hier die notwendigen Änderungen. Als Erstes ändern wir den Namen der Klasse **tblFotos.vb** in **Foto.vb**.

In der Klasse **Foto.vb** müssen wir den Namen der Eigenschaft **Foto** in Fotodaten ändern, da keine Eigenschaft genauso heißen darf wie die Klasse:

```
Partial Public Class Foto
    Public Property ID As Integer
    <Column(TypeName:="image")>
    <Required>
    Public Property Fotodaten As Byte()
End Class
```

Damit die Klasse **Foto** auf die Tabelle **tblFotos** gemappt wird und die Eigenschaft **Fotodaten** der Klasse auf das Feld **Foto** der Tabelle, fügen wir folgende Zeilen zur Methode **OnModelCreating** der Klasse **FotoverwaltungContext.vb** hinzu:

```
Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
    modelBuilder.Entity(Of Foto)().
        ToTable("tblFotos").
        Property(Function(t) t.Fotodaten).HasColumnName("Foto")
End Sub
```

Damit können wir nun testen, ob der Zugriff auf die Datensätze der Tabelle **tblFotos** gelingt. Dazu fügen wir der Klasse **MainWindow.xaml.vb** zunächst eine Konstruktor-Methode hinzu. In dieser erstellen wir einen Datenbankkontext für den Zugriff auf

die Datenbank. Außerdem versuchen wir, auf das erste Element der Auflistung **Fotos** dieses Kontextes zuzugreifen und den Inhalt der Eigenschaften **ID** und **Fotodaten** in einem Meldungsfenster auszugeben:

```
Class MainWindow
    Public Sub New()
        Dim dbContext As FotoverwaltungContext
        dbContext = New FotoverwaltungContext
        Dim foto As Foto
        foto = dbContext.Fotos.First()
        MessageBox.Show(foto.ID.ToString() + vbCrLf + vbCrLf + foto.Fotodaten.ToString())
    End Sub
End Class
```

Sofern Sie bereits einen Datensatz zu dieser Tabelle hinzugefügt haben, werden die entsprechenden Informationen beim Start der Anwendung ausgegeben.

Bild im WPF-Fenster anzeigen

Nun schauen wir uns an, wie wir den Inhalt der Eigenschaft **Fotodaten**, also des Feldes **Foto** der Tabelle **tblFotos**, in einem Steuerelement in einem WPF-Fenster anzeigen. Dazu benötigen wir zunächst ein geeignetes Steuerelement. Wir fügen dem Fenster **MainWindow.xaml** zunächst ein **Image**-Steuerelement hinzu, das wir mit dem Bild aus der Eigenschaft **Fotodaten** füllen wollen. Das Steuerelement ziehen wir einfach aus der Toolbox in das WPF-Fenster. Dadurch entsteht etwa der folgende XAML-Code:

```
<Image HorizontalAlignment="Left" Height="288" Margin="112,50,0,0" VerticalAlignment="Top" Width="364"/>
```

Das Bild ist im Feld **Foto** der Tabelle **tblFotos** unter dem Datentyp **image** gespeichert (siehe Bild 6). Wir müssen nun einen Weg finden, den Inhalt dieses Feldes im **Image**-Steuerelement anzuzeigen. Dazu verarbeiten wir dieses zunächst in der Code behind-Klasse **MainWindow.xaml.vb**. Hier verschieben wir die Deklaration des Datenbankkontextes zunächst aus der Konstruktor-Methode in den allgemeinen Teil der Klasse:

```
Class MainWindow
    Dim dbContext As FotoverwaltungContext
```

Dann deklarieren wir eine private Variable namens **m_Fotodaten** mit dem Datentyp **BitmapImage**, die wir über die folgende **Property** namens **Fotodaten** mit dem gleichen Datentyp für den lesenden und schreibenden Zugriff verfügbar machen:

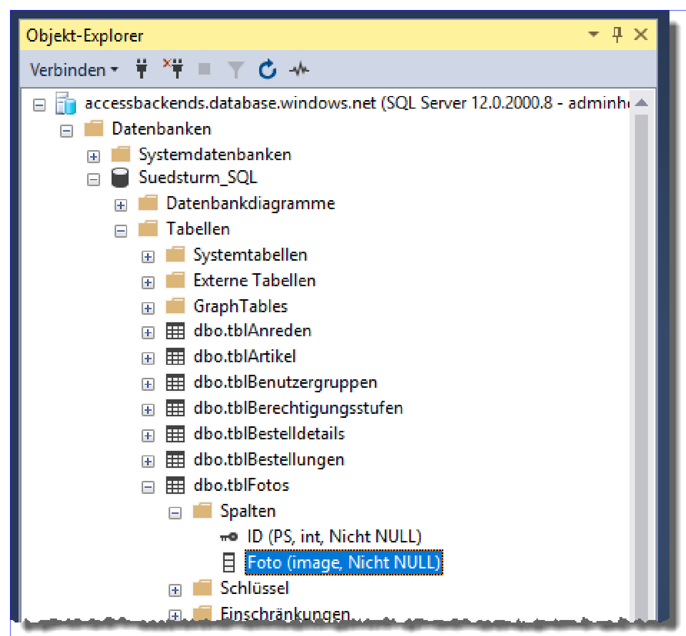


Bild 6: Das Feld **Foto** in der Tabelle **tblFotos**

```
Dim m_Fotodaten As BitmapImage
Public Property Fotodaten As BitmapImage
    Get
        Return m_Fotodaten
    End Get
    Set(value As BitmapImage)
        m_Fotodaten = value
    End Set
End Property
```

Danach erweitern wir die Konstruktor-Methode **New**. Nach dem Zuweisen des Datenbankkontextes zur Variablen dbContext deklarieren wir ein Objekt des Typs **Foto** und weisen diesem das erste Element der mit der Auflistung **Fotos** eingelesenen Tabelle **tblFotos** zu:

```
Public Sub New()
    dbContext = New FotoverwaltungContext
    Dim foto As Foto
    foto = dbContext.Fotos.First()
```

Dann lesen wir zunächst den Inhalt der Eigenschaft **Fotodaten** (also eigentlich des Feldes **Foto** der Tabelle **tblFotos**) in ein Objekt des Typs **Byte** mit dem Namen **blob** ein. Danach erstellen wir ein Objekt namens **stream** mit dem Typ **MemoryStream**, in das wir den Inhalt des Byte-Arrays **blob** mit der **Write**-Methode schreiben.

Dann stellen wir die Position des **Stream**-Objekts auf die Position **0** ein:

```
    Dim blob As Byte() = foto.Fotodaten
    Dim stream As MemoryStream = New MemoryStream()
    stream.Write(blob, 0, blob.Length)
    stream.Position = 0
```

Nun kommt erstmals das Objekt des Typs **BitmapImage** ins Spiel, das wir anschließend auch dem **Image**-Steuerelement als Quelle zuweisen können. Dieses erstellen wir unter dem Namen **img** und initialisieren es mit der Methode **BeginInit**.

Dann weisen wir als **StreamSource** den Inhalt der **MemoryStream**-Variablen **stream** zu und beenden die Initialisierung von **img** mit der **EndInit**-Methode:

```
    Dim img As BitmapImage
    img = New BitmapImage
    img.BeginInit()
    img.StreamSource = stream
    img.EndInit()
```

Schließlich weisen wir **img** der öffentlichen Eigenschaft **Fotodaten** zu und stellen die aktuelle Klasse als **DataContext** der **WPF**-Seite ein:

```
        Fotodaten = img
        DataContext = Me
    End Sub
End Class
```

Danach müssen wir nur noch der Eigenschaft **Source** des **Image**-Elements im XAML-Code den Pfad zur Eigenschaft **Fotodaten** mitteilen:

```
<Image HorizontalAlignment="Left" Height="288" Margin="112,50,0,0"
VerticalAlignment="Top" Width="364" Source="{Binding Path=Fotodaten}"/>
```

Das Ergebnis sieht dann wie in Bild 7 aus.

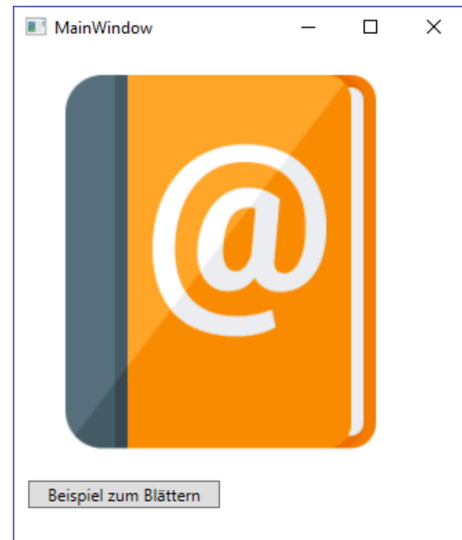


Bild 7: Ein Bild im WPF-Fenster

Durch Bilder blättern

Wenn wir nun nicht nur ein Bild anzeigen, sondern auch durch alle vorhandenen Bilder blättern wollen, benötigen wir prinzipiell die gleichen Techniken, die wir auch sonst zum Blättern in Datensätzen verwenden. Dazu erstellen wir ein neues Fenster namens **FotosBlaettern.xaml**. Dieses öffnen wir vom Startfenster **MainWindow.xaml** aus mit einer neuen Schaltfläche, der wir die folgenden Codezeilen hinzufügen:

```
Private Sub btnBeispielZumBlaettern_Click(sender As Object, e As RoutedEventArgs)
    Dim wnd As FotosBlaettern
    wnd = New FotosBlaettern
    wnd.ShowDialog()
End Sub
```

Die Techniken, die wir weiter oben kennengelernt haben, können wir für das Blättern durch Datensätze mit Bildern ebenfalls verwenden. Außerdem nutzen wir einige Techniken, die wir im Artikel **EDM: Blättern in Datensätzen** vorgestellt haben. Im XAML-Code verwenden wir in den Ressourcen einige DataTrigger, mit denen wir die Schaltflächen aktivieren und deaktivieren wollen (siehe ebenfalls im oben genannten Artikel):

```
<Window x:Class="FotosBlaettern" ... Title="FotosBlaettern" Height="450" Width="800">
    <Window.Resources>
        <Style x:Key="ButtonVorheriger" TargetType="{x:Type Button}">
            <Style.Triggers>
                <DataTrigger Binding="{Binding Erster}" Value="False">
                    <Setter Property="IsEnabled" Value="False"></Setter>
                </DataTrigger>
            </Style.Triggers>
```

```
        </Style>
        <Style x:Key="ButtonNaechster" TargetType="{x:Type Button}">
            <Style.Triggers>
                <DataTrigger Binding="{Binding Letzter}" Value="False">
                    <Setter Property="IsEnabled" Value="False"></Setter>
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>
```

Dann definieren wir drei Zeilen, von denen die erste das Feld mit der ID des Datensatzes anzeigt, die zweite das **Image**-Steuerelement und die dritte die Schaltflächen zum Navigieren in den Datensaätzen:

```
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
        </Grid.RowDefinitions>
```

Die TextBox binden wir an das Feld **Foto.ID**:

```
        <TextBox Grid.Row="0" Margin="5,5,5,5" Height="25" x:Name="txtId" Text="{Binding Foto.ID}"></TextBox>
```

Im gleichen Bereich bringen wir auch noch eine Schaltfläche zum Ändern des aktuellen Bildes unter:

```
        <Button x:Name="btnBildAendern" Grid.Row="0" Margin="5,5,5,5" HorizontalAlignment="Right"
            Width="90" Content="Bild ändern" Click="btnBildAendern_Click"></Button>
```

Das **Image**-Steuerelement fassen wir in ein **Border**-Element ein. Es bindet an die Eigenschaft **Foto.FotoImage**. Wobei Sie feststellen werden, dass unsere Tabelle gar kein solches Feld enthält – und auch nicht unsere **Foto**-Klasse. Dazu kommen wir später:

```
        <Border Grid.Row="1" Margin="5,5,5,5" BorderThickness="1" BorderBrush="#ff000000" >
            <Image Source="{Binding Foto.FotoImage}" />
        </Border>
```

Für die dritte Zeile sehen wir ein **StackPanel**-Element vor, dass die Schaltflächen in horizontaler Anordnung aufnimmt:

```
        <StackPanel Orientation="Horizontal" Grid.Row="2" ...>
```

```
            <Button x:Name="btnErster" Style="{StaticResource ButtonVorheriger}" Content="&lt;&lt;"
                Click="btnErster_Click" Width="25" Margin="2,2,2,2"/>
            <Button x:Name="btnVorheriger" Style="{StaticResource ButtonVorheriger}" Content="&lt;"
                Click="btnVorheriger_Click" Width="25" Margin="2,2,2,2"/>
            <Button x:Name="btnNaechster" Style="{StaticResource ButtonNaechster}" Content="&gt;"
                Click="btnNaechster_Click" Width="25" Margin="2,2,2,2"/>
            <Button x:Name="btnLetzter" Style="{StaticResource ButtonNaechster}" Content="&gt;&gt;"
                Click="btnLetzter_Click" Width="25" Margin="2"/>
            <Button x:Name="btnNeu" Content="*" Click="btnNeu_Click" Width="25" Margin="2,2,2,2"/>
        </StackPanel>
    </Grid>
</Window>
```

Im Entwurf sieht das Fenster nun wie in Bild 8 aus.

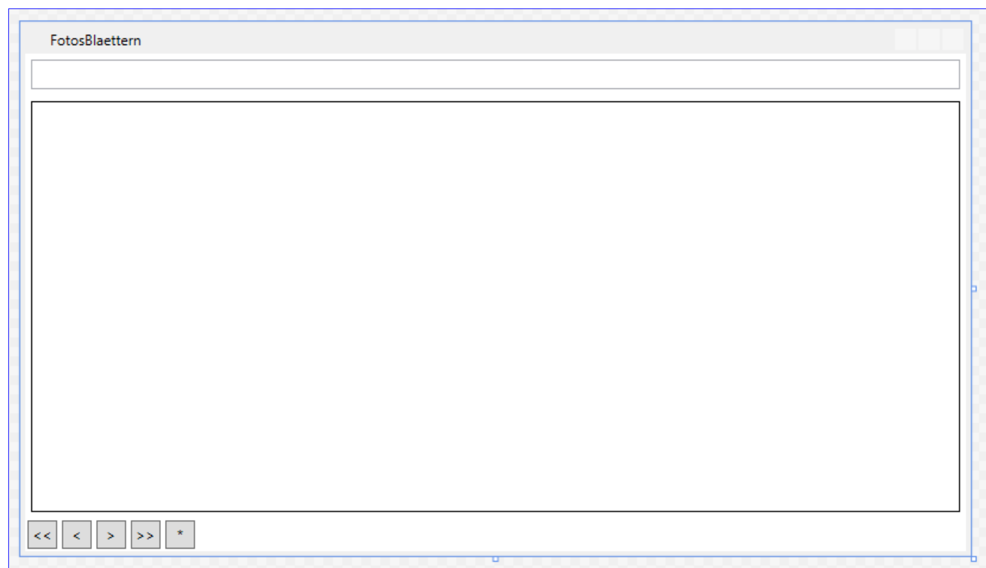


Bild 8: Entwurf des Fensters **FotosBlaettern**

Prozeduren zum Blättern durch die Bilder

Damit der Benutzer mit den Schaltflächen durch die Datensätze blättern kann und dabei immer die jeweiligen Bilder angezeigt werden, benötigen wir zunächst ein paar Verweise auf Namespaces:

```
Imports System.Threading
Imports System.Globalization
Imports System.IO
Imports System.ComponentModel
Imports System.Collections.ObjectModel
Imports System.Data.Entity.Validation
Imports Microsoft.Win32
```

Die Code behind-Klasse **FotosBlaettern.xaml.vb** implementiert die Schnittstelle **INotifyPropertyChanged** und deklariert die folgenden privaten Variablen:

```
Public Class FotosBlaettern
    Implements INotifyPropertyChanged
```

EDM für bestehende Datenbank mit Code First

Wenn Sie ein Entity Data Model mit der Vorlage »Code First aus Datenbank« auf Basis einer bestehenden Datenbank erstellen, haben Sie vielleicht Pech und die Namen der Tabellen der Datenbank und der enthaltenen Felder lauten nicht so, wie Sie die Entitätsklassen, die DbSet-Elemente und die Eigenschaften der Klassen nennen möchten. Dann haben Sie verschiedene Möglichkeiten: Zum Beispiel können Sie die Bezeichnungen in der Datenbank anpassen. Das geht aber oft nicht, weil vielleicht noch andere Frontends auf die gleiche Datenbank zugreifen. Dann haben Sie noch die Möglichkeit, die Bezeichnungen von Datenbank und Entity Data Model so zu mappen, dass beide Seiten zufrieden sind. Wie letzteres gelingt, zeigen wir im vorliegenden Artikel am Beispiel der Südsturm-Datenbank.

Einfaches Beispiel: tblFotos

Wir starten mit einem sehr einfachen Beispiel, nämlich der Tabelle **tblFotos**. Diese haben wir als Tabelle zum Speichern von Fotos erstellt, die wir mit einer PowerApp über das Smartphone aufnehmen wollen. Dabei haben wir die Tabelle leichtsinnigerweise **tblFotos** genannt, statt einfach **Fotos** ohne Präfix. Wenn wir nun ein Entity Data Model erstellen, erhalten wir für die Klasse **FotoverwaltungContext.db** etwa den folgenden Code (siehe Projekt **BilderWPF**, Beispieldatenbank in **Suedsturm_SQL.sql**):

```
Imports System.ComponentModel.DataAnnotations.Schema

Partial Public Class FotoverwaltungContext
    Inherits DbContext
    Public Sub New()
        MyBase.New("name=FotoverwaltungContext")
    End Sub
    Public Overridable Property tblFotos As DbSet(Of tblFotos)
    Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
    End Sub
End Class
```

Hier hätten wir gern den Namen der Property für das **DbSet** so geändert, dass es **Fotos** statt **tblFotos** heißt und Elemente des Typs **Foto** enthält (die auch noch **tblFotos** heißen). In der Entitätsklasse **tblFotos** geht es so weiter. Diese hat nach dem Erstellen des Entity Data Models den folgenden Code erhalten:

```
Imports System.ComponentModel.DataAnnotations
Imports System.ComponentModel.DataAnnotations.Schema

Partial Public Class tblFotos
    Public Property ID As Integer
```

```
    <Column(TypeName:="image")>
    <Required>
    Public Property Foto As Byte()
End Class
```

Wir wollen dies Schritt für Schritt so anpassen, dass wir mit einer Konstruktor-Methode für das Fenster **MainWindow** wie der folgenden auf die Daten zugreifen können:

```
Class MainWindow
    Public Sub New()
        Dim dbContext As FotoverwaltungContext
        dbContext = New FotoverwaltungContext
        Dim foto As Foto
        foto = dbContext.Fotos.First()
        MessageBox.Show(foto.ID.ToString())
    End Sub
End Class
```

Wir wollen also ein **DbSet** namens **Fotos** verwenden und damit auf Elemente des Typs **Foto** zugreifen. Dazu müssen wir dem Entity Data Model auf irgendeine Weise mitteilen, dass es das **DbSet** namens **Fotos** auf die Tabelle **tblFotos** mappen soll und die Klasse **Foto** auf die einzelnen Datensätze.

Mapping in der Methode OnModelCreating

Der richtige Ort für ein solches Mapping ist die Methode **OnModelCreating**, die beim Erstellen des Entity Data Models auf Basis der Vorlage **Code First aus Datenbank** automatisch in der Klasse **FotoverwaltungContext** angelegt wurde. Der erste Schritt ist das Umbenennen der Klasse **tblFotos** in **Foto**. Das erledigen wir ganz einfach, indem wir den entsprechenden Eintrag im Projektmappen-Explorer markieren, diesen nochmals anklicken und dann die Bezeichnung ändern. Danach erscheint noch eine Meldung, die fragt, ob Verweise auf das Codeelement angepasst werden sollen (siehe Bild 1).

Wenn Sie hier auf **Ja** klicken, werden in unserer kleinen Beispielanwendung folgende Änderungen durchgeführt:

- Die Bezeichnung der Klasse wird ebenfalls in **Foto** geändert.

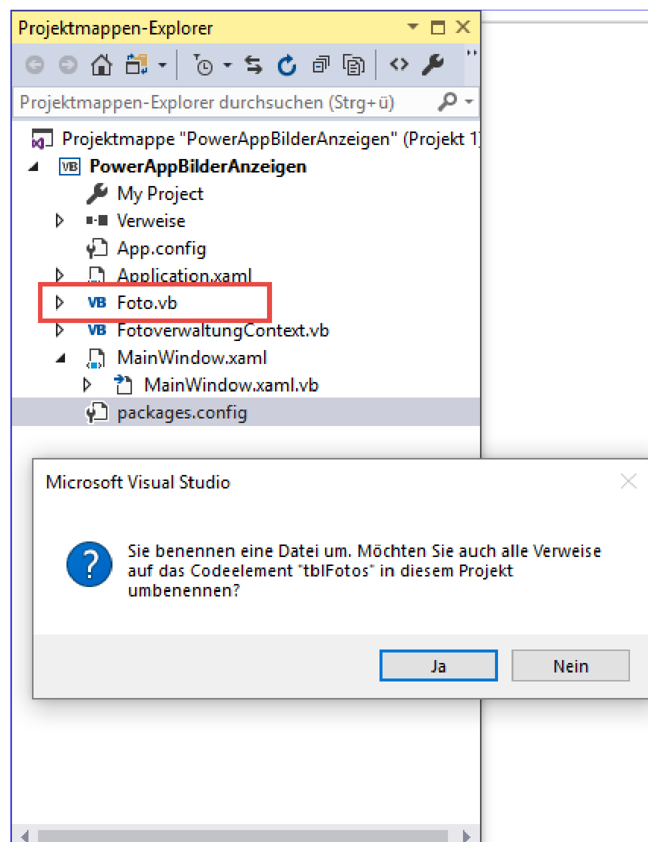


Bild 1: Ändern eines Klassennamens

- In der Klasse **FotoverwaltungContext** wird der Typ der Klasse des **DbSets** ebenfalls geändert:

```
Partial Public Class FotoverwaltungContext
    Inherits DbContext
    ...
    Public Overridable Property tblFotos As DbSet(Of Foto)
    ...
End Class
```

In dieser Klasse sind dann weitere Änderungen nötig. Anschließend sieht die oben bereits teilweise geänderte Zeile mit der Definition des **DbSet** wie folgt aus:

```
Public Overridable Property Fotos As DbSet(Of Foto)
```

Mapping hinzufügen

Damit passen die Deklarationen der Klasse und des **DbSet**-Elements nun zu dem Code, den wir für die Konstruktor-Methode unseres Fensters **MainWindow.xaml** erstellt haben. Was geschieht nun, wenn wir die Anwendung starten?

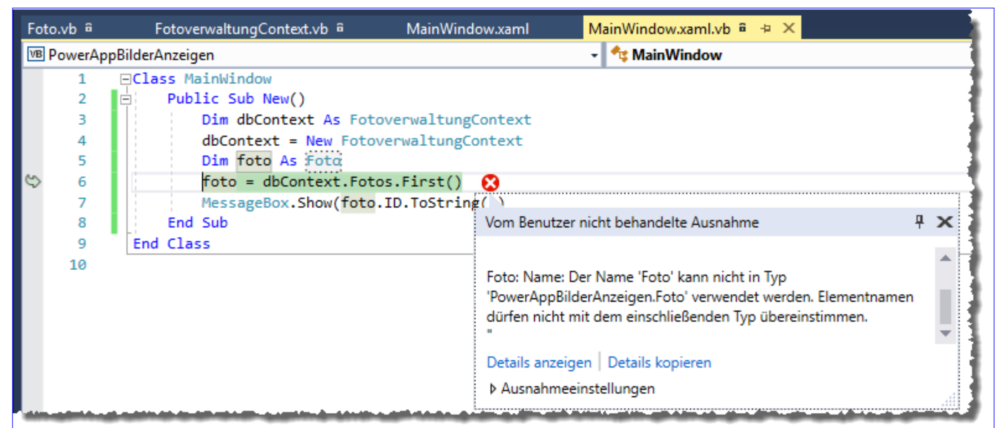


Bild 2: Fehler beim Zugriff auf die Klasse

Wir erhalten einen unerwarteten Fehler: Visual Studio bemängelt, dass wir eine Eigenschaft namens **Foto** in der gleichnamigen Klasse verwenden (siehe Bild 2). Damit erhalten wir also noch ein Problem, das aus der Benennung der Tabellen und Felder der Beispieldatenbank resultiert. Das Feld **Foto** können wir nicht mit dem Eigenschaftsnamen **Foto** ansprechen, da eine Klasse keine Eigenschaften besitzen darf, die genauso heißen wir die Klasse selbst. Also ändern wir den Namen der Eigenschaft in der Klasse **Foto** auf **Fotodaten**:

```
Partial Public Class Foto
    ...
    Public Property Fotodaten As Byte()
End Class
```

Nach einem erneuten Start der Anwendung erhalten wir dann die Fehler, mit denen wir gerechnet hätten. Der erste lautet wie folgt und er tritt beim Zugriff auf die Daten über **dbContext.Fotos.First** auf (siehe auch Bild 3):

```
System.InvalidOperationException: "Die Sequenz enthält keine Elemente."
```

Das ist etwas überraschend, denn wir hatten mit einem Fehler gerechnet, der durch eine fehlende Tabelle ausgelöst wird. Schauen wir uns die Eigenschaften der Auflistung **Fotos** wie in Bild 4 an, sehen wir, dass das Entity Framework anscheinend versucht, auf eine Tabelle namens **Fotoes** zuzugreifen. Das ist offensichtlich die Plural-Form der Klasse **Foto**, die Entity Framework automatisch gebildet hat, um auf die Tabelle zuzugreifen.

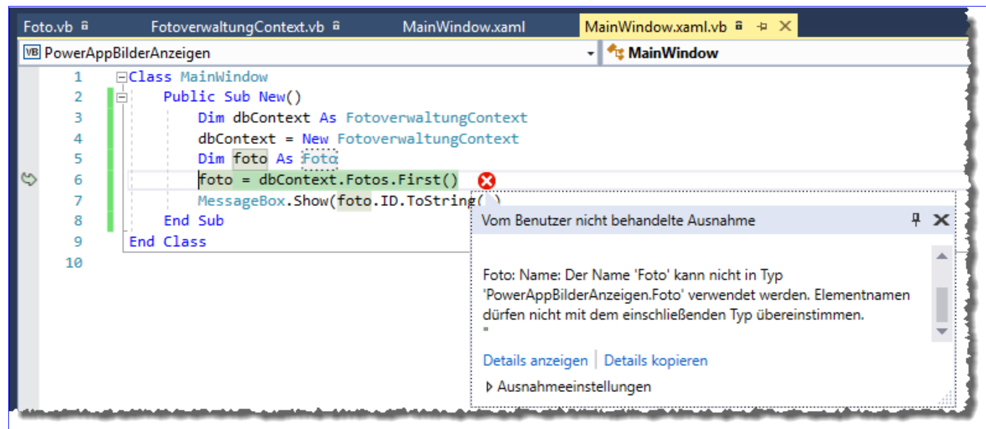


Bild 3: Fehler beim Zugriff auf die Tabelle

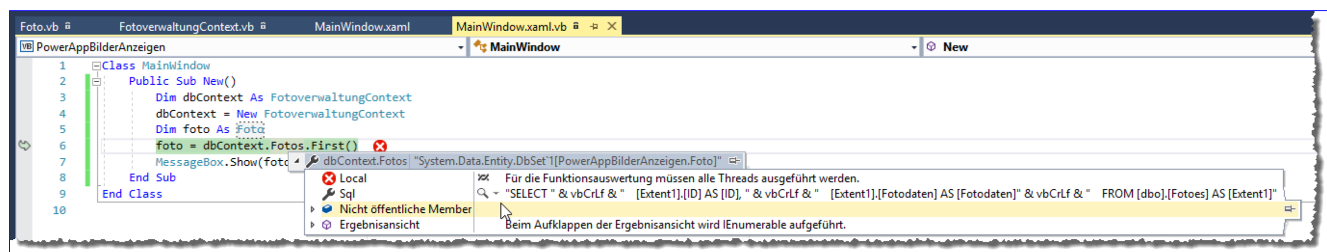


Bild 4: Der Zugriff erfolgt auf die nicht vorhandene Tabelle **Fotoes**

Damit Entity Framework erkennt, dass wir über die **DbSet**-Auflistung Fotos auf die Daten der Tabelle **tblFotos** zugreifen wollen, fügen wir der Methode **OnModelCreating** die folgende Anweisung hinzu:

```
Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
    modelBuilder.Entity(Of Foto)().ToTable("tblFotos")
End Sub
```

Daraufhin erhalten wir die Meldung aus Bild 5, die uns darauf hinweist, dass das Unterstützungsmodell geändert worden sei und wir mit einer Code First-Migration die Änderungen im Datenmodell in die Datenbank übertragen könnten. Was ist damit gemeint?

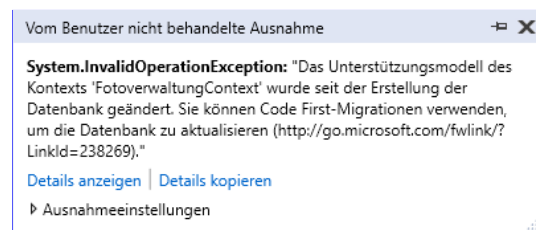


Bild 5: Unterstützungsmodell geändert?

Schauen wir uns nochmal den Inhalt des Objekts **Fotos** im Debug-Modus an, sehen wir, dass Entity Framework zwar nun auf die richtige Tabelle namens **tblFotos** zugreift (siehe Bild 6). Allerdings lautet der Name des Feldes nun **Fotodaten**. In der Tabelle **tblFotos** der Datenbank heißt es allerdings **Foto**. Entity Framework denkt also nun offensichtlich anhand des Unterschiedes zwischen dem in der Abfrage genannten Feldnamen **Fotodaten** und dem in der Tabelle vorgefundenen Feld **Foto**, dass der Benutzer das Entity Data Modell geändert hat

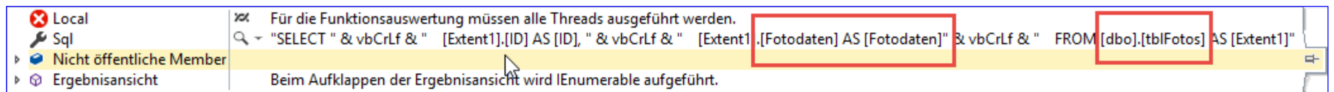


Bild 6: Die Anwendung versucht zwar, auf die richtige Tabelle zuzugreifen (**tblFotos**), aber noch nicht auf das richtige Feld.

und bietet eine Möglichkeit an, diese Änderung in die Datenbank zu übertragen. Das wollen wir allerdings nicht, sondern wir möchten das Mapping so anpassen, dass für die Eigenschaft **Fotodaten** der Entität **Foto** auf das Feld **Foto** der Tabelle **tblFotos** zugegriffen wird.

Also fügen wir noch einen weiteren Teil zur Methode **OnModelCreating** hinzu, mit der wir das Feld **Fotodaten** auf das Feld **Foto** der Tabelle **tblFotos** mappen. Das sieht dann wie folgt aus:

```
Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
    modelBuilder.Entity(Of Foto)().
        ToTable("tblFotos").
        Property(Function(t) t.Fotodaten).HasColumnName("Foto")
End Sub
```

Beim nächsten Start erhalten wir allerdings wieder die gleiche Meldung mit dem Hinweis auf den Einsatz der Code First-Migrationen. Wenn wir uns den Inhalt von **dbContect.Fotos** ansehen, finden wir allerdings folgende SQL-Anweisung vor:

```
"SELECT " & vbCrLf & "    [Extent1].[ID] AS [ID], " & vbCrLf & "    [Extent1].[Foto] AS [Foto]" & vbCrLf & "    FROM 
[dbo].[tblFotos] AS [Extent1]"
```

Wenn wir diese **SELECT**-Anweisung im SQL Server Management Studio in einer neuen Abfrage für die hier verwendete Datenbank ausführen, erhalten wir allerdings das gewünschte Ergebnis (siehe Bild 7).

Unterschiede per Migration aufdecken

In den Artikeln **Datenbank-Initialisierung** und **Datenbank-Migration** haben wir die Migration von Code First-Datenbanken in das jeweilige Datenbanksystem beschrieben. Wie dort erläutert, werden wir nun die Migration aktivieren. Wir wollen allerdings keine Migration ausführen, sondern diese nur nutzen, um herauszufinden, welche Unterschiede zwischen dem Entity Data Model und dem Datenmodell der Datenbank zu dem aufgetretenen Fehler führen. Also öffnen Sie die Paket-Manager-Konsole und geben dort den folgenden Befehl ein:

```
enable-migrations
```

Dadurch wird der **Migrations**-Ordner angelegt, der auch direkt eine Klasse namens **...InitialCreate.vb** enthält. Das ist normalerweise nicht der Fall – üblicherweise müssen Sie

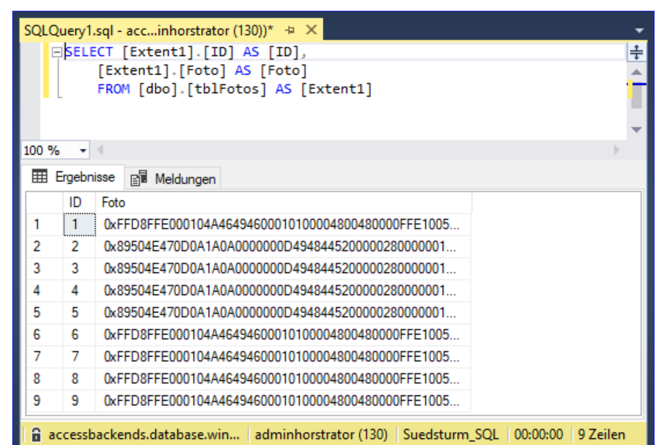


Bild 7: Die Abfrage funktioniert wie gewünscht.

die erste Migration mit dem Befehl **add-migration <name>** erstellen.

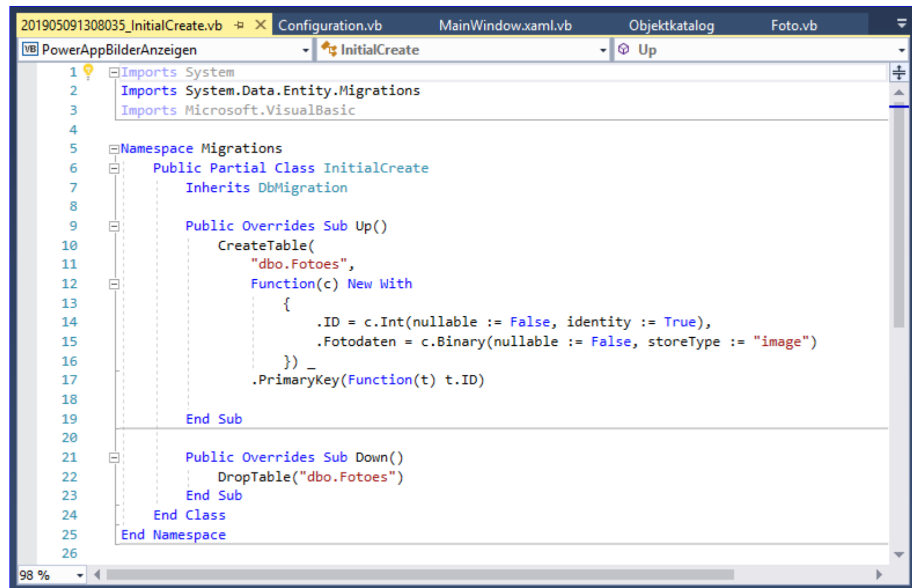


Bild 8: Die durchzuführenden Befehle, um das Entity Data Model in das Datenmodell zu migrieren

Diese Klasse schauen wir uns nun an und finden sie wie in Bild 8 vor. Wie hier zu erkennen, will das Entity Framework eine Tabelle namens **Fotoes** erstellen. Das erklärt auch direkt das Problem: Mit den aktuellen (Standard-)Einstellungen erstellt Entity Framework aus den Namen der Entitätsklassen jeweils die Pluralform, es wird also **-es** oder **-s** an den Entitätsnamen angehängt. Aber auch hierfür gibt es noch eine Einstellung, die wir in der Methode **OnModelCreating** vornehmen können. Danach sieht die Methode wie folgt aus:

```
Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
    modelBuilder.Conventions.Remove(Of PluralizingTableNameConvention)()
    modelBuilder.Entity(Of Foto)().
        ToTable("tblFotos").
        Property(Function(t) t.Fotodaten).HasColumnName("Foto")
End Sub
```

Nach der Ergänzung erhalten wir allerdings nach wie vor die gleiche Fehlermeldung. Also löschen wir den Ordner **Migrations** und führen nochmals die Anweisung **enable-migrations** in der Paket-Manager-Konsole aus. Auch dies schafft jedoch keine Abhilfe. Entity Framework möchte nach wie vor eine Tabelle namens **Fotoes** erstellen. Aber auch das Erstellen einer Tabelle namens **Foto** wäre ja falsch. Wir wollen eine Tabelle namens **tblFotos** haben, die ja auch schon in der Datenbank vorhanden ist.

Wo aber kommt immer noch die Idee der **InitialCreate.vb**-Klasse her, dass wir eine Tabelle namens **Fotoes** in der Datenbank benötigen? Des Rätsels Lösung ist die Tabelle **_MigrationsHistory** in der Datenbank. Diese wurde wohl zwischendurch angelegt und enthält Informationen, die Entity Framework an dieser Stelle irritiert haben. Nachdem wir diese Tabelle in der Datenbank gelöscht haben, konnten wir ohne Probleme mit unserer Anwendung mit dem **DbSet** namens **Fotos** und der Klasse **Foto** auf die Daten der Tabelle **tblFotos** zugreifen.

Zusammenfassung der ersten Schritte

Die Aufgabe lautete, die Tabelle **tblFotos** mit den beiden Feldern **ID** und **Foto** über Entity Framework mit einem Entity Data Model zu verbinden, das eine Klasse namens Foto und eine **DbSet**-Auflistung namens **Fotos** aufweist und bei der wir auch noch den Namen des Feldes **Foto** in **Fotodaten** ändern mussten. Das gelingt, indem wir die folgenden einfachen Schritte durchführen:

- Name der Entitätsklasse ändern (von **tblFotos** in **Foto**)
- Name des Feldes **Foto** in **Fotodaten** ändern
- Name des **DbSet**-Elements **tblFotos** in **Fotos** ändern
- Klasse **OnModelCreating** erweitern um das Mapping der Entität **Foto** zur Tabelle **tblFotos** und der Eigenschaft **Fotodaten** zum Feld **Fotos**
- Prüfen, ob der Zugriff auf ein Element der Tabelle **tblFotos** gelingt und oben beschriebene Fehler als Anhaltspunkte für noch vorhandene Probleme nutzen

Komplexere Anpassungen

Wie Sie sehen, kann schon das Anpassen des Entity Data Models an unsere Wünsche und das damit verbundene, notwendige Mapping auf die Tabellen und Felder der Datenbank eine sehr interessante Aufgabe werden. Wenn Sie einmal die hier aufgetretenen Schwierigkeiten umschifft haben, können Sie sich größeren Anpassungen zuwenden wie etwa der kompletter Datenmodelle. Also legen wir ein neues Projekt an, das diesmal alle Tabellen der Südsturm-Datenbank aufnehmen soll (siehe Projekt **Suedsturm_DBFirst**, Beispieldatenbank aus **Suedsturm_SQL.sql**). Dem neuen Projekt fügen wir dann wieder ein neues Element des Typs **ADO.NET Entity Data Model** hinzu, das wir **SuedsturmContext** nennen. Wählen Sie die Vorlage **Code First aus Datenbank** aus und stellen Sie im nächsten Dialog die entsprechende Verbindung ein. Im nächsten Schritt wählen Sie die Tabellen wie in Bild 9 aus.

Danach finden Sie für jede Tabelle eine Entitätsklasse im Projektmappen-Explorer vor (siehe Bild 10).

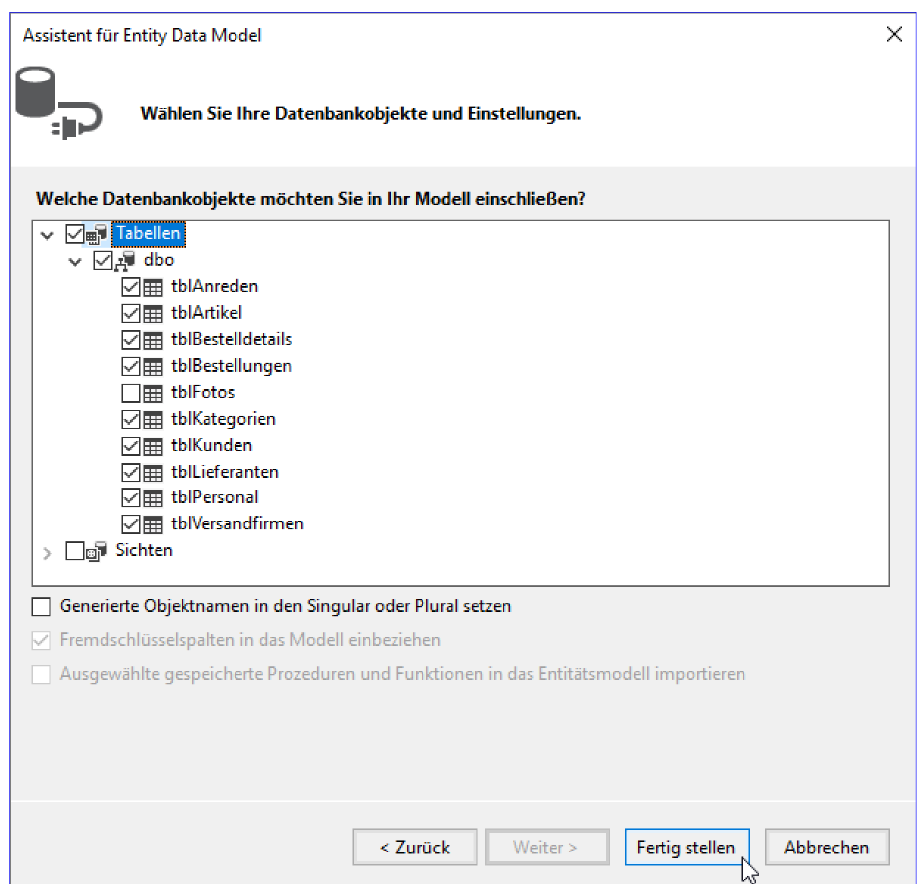


Bild 9: Auswahl der gewünschten Tabellen

Test direkt nach dem Anlegen des Entity Data Models

Nachdem wir das Entity Data Model erstellt haben, testen wir den Zugriff auf die Tabellen über das Entity Data Model mit der folgenden Konstruk-

EDM: Blättern in Datensätzen

Wenn Sie und Ihre Kunden die Arbeit mit der Detailansicht von Access-Formularen gewohnt sind, möchten Sie vielleicht auch in einem WPF-Fenster mit der Detailansicht der Daten einer Klasse blättern können. Dieser Artikel zeigt, wie Sie einem Detailformular die Daten einer Tabelle über die entsprechende DbSet-Auflistung zuweisen und wie Sie mit entsprechenden Schaltflächen in den Datensätzen blättern können.

Ziel der Erweiterung

Welche Möglichkeiten zum Blättern in den Datensätzen wollen wir dem Benutzer anbieten? Der Benutzer soll zum ersten, zum letzten, zum vorherigen und zum nächsten Datensatz blättern können. Außerdem wäre es hilfreich, wenn direkt ein neuer, leerer Datensatz angezeigt werden könnte.

Voraussetzungen

Wir bauen uns auf die Schnelle eine Beispielanwendung zusammen. Dazu nutzen wir die Techniken, die wir im Artikel **Detailformulare mit weiteren Steuerelementen** vorgestellt haben. Alternativ können Sie auch einfach das Beispielprojekt aus dem Download verwenden.

Die Schritte sehen in aller Kürze wie folgt aus:

- Neues Projekt auf Basis der Vorlage **Visual Basic|Windows Desktop|WPF-App** erstellen.
- Neues Element des Typs **ADO.NET Entity Data Model** namens **BestellverwaltungContext** mit der Vorlage **Leeres Code First-Modell** hinzufügen.
- In der Access-Datenbank **AccessZuWPFFormulare.accdb** aus dem Download die Prozedur **EDMErstellen** des Moduls **mdlEDM** aufrufen und den entstehenden Code aus der Zwischenablage nach den Anweisungen in den Kommentaren an den entsprechenden Stellen platzieren. Hier finden sich auch die Hinweise auf die Aufrufe der Befehle Enable-Migrations, Add-Migration und Update-Database, mit denen die Datenbank erstellt und gefüllt wird.
- Den XAML-Code für das gewünschte Detailformular, zum Beispiel **frmKundendetails**, mit dem folgenden Aufruf der Prozedur **FormularNachWPF** in die Zwischenablage kopieren: **FormularNachWPF "frmKundendetails", "tblKunden"**
- Den Inhalt der Zwischenablage in das Fenster **MainWindow.xaml** kopieren (oder in ein anderes, neues Fenster)
- Den VB-Code für die Code behind-Datei mit dem folgenden Aufruf der Prozedur **FormularNachWPF_CodeBehind** in die Zwischenablage kopieren: **FormularNachWPF_CodeBehind "frmKundendetails", "Kunde", "Kunden", "BestellverwaltungContext"**
- Den Inhalt der Zwischenablage in das Code behind-Modul **MainWindow.xaml.vb** oder das entsprechende andere Code behind-Modul kopieren.

Das Ergebnis nach dem Start der Anwendung sieht wie in Bild 1 aus. Hier wollen wir nun die Schaltflächen zum Blättern in den Datensätzen hinzufügen.

Bild 1: Fenster ohne Schaltflächen zum Blättern

Schaltflächen hinzufügen

Die Schaltflächen fügen wir in einem Element namens **StackPanel** hinzu, das ein Container für nebeneinander oder übereinander angeordnete Steuerelemente ist. Dieses Element füllen wir mit fünf **Button**-Elementen, deren Code wir wie folgt ergänzen:

```
<StackPanel Orientation="Horizontal"
HorizontalAlignment="Left" Height="28" Margin="0,273,0,0"
VerticalAlignment="Top" Width="339">
    <Button x:Name="btnErster" Content="&lt;&lt;"
Click="btnErster_Click" Width="25" Margin="2,2,2,2"/>
    <Button x:Name="btnVorheriger" Content="&lt;"
Click="btnVorheriger_Click" Width="25" Margin="2,2,2,2"/>
    <Button x:Name="btnNaechster" Content="&gt;" Click="btnNaechster_Click" Width="25" Margin="2,2,2,2"/>
    <Button x:Name="btnLetzter" Content="&gt;&gt;" Click="btnLetzter_Click" Width="25" Margin="2,2,2,2"/>
    <Button x:Name="btnNeu" Content="*" Click="btnNeu_Click" Width="25" Margin="2,2,2,2"/>
</StackPanel>
```

Dabei müssen die Kleiner- und Größer-Zeichen im Code in der Eigenschaft **Content** codiert werden, und zwar das Größer-Zeichen als **<** und das Kleiner-Zeichen als **>**. Für jede Schaltfläche haben wir ein Attribut namens **Click** angelegt, für das wir den Namen der beim Anklicken auszuführenden Ereignismethode angeben. Um die Eigenschaft schnell zu füllen, geben Sie das erste Anführungszeichen ein und wählen dann den IntelliSense-Eintrag **Neuer Ereignishandler** aus. Dadurch werden dann auch direkt die entsprechenden, noch leeren Ereignismethoden in der Code behind-Klasse erstellt.

Anzeigen des ersten Datensatzes

Damit beim Öffnen des Fensters direkt der erste Datensatz der Datenherkunft angezeigt wird, haben wir eine Konstruktor-Methode hinterlegt, welche die folgenden Befehle ausführt. Als Erstes füllen wir dabei die Variable **dbContext** mit einer neuen Instanz unserer Context-Klasse **BestellverwaltungContext**. Dann füllen wir die Auflistung **Kunden** mit einer **Observable-Collection**, die alle Einträge des **DbSet**-Objekts **Kunden** liefert. Schließlich füllen wir die Eigenschaft **Kunde** mit dem ersten Element der Auflistung **Kunden** (**Kunden.First()**) und füllen auch noch die Auflistung **Anreden**, welche die Datensätze für das **ComboBox**-Element **Anreden** enthält:

```
Public Sub New()
    InitializeComponent()
    dbContext = New BestellverwaltungContext
    Kunden = New ObservableCollection(Of Kunde)(dbContext.Kunden)
```

```
    Kunde = Kunden.First()
    Anreden = New List(Of Anrede)(dbContext.Anreden)
    DataContext = Me
End Sub
```

Ereignismethoden füllen

Damit zeigt das Fenster nun immer den ersten Datensatz an. Wir wollen aber mit den bereits angelegten Schaltflächen zwischen den Datensätzen navigieren können. Dazu schauen wir uns zunächst an, wie wir vom aktuellen Datensatz zum nächsten Datensatz gelangen, also zum zweiten Datensatz der **ObservableCollection** namens **Kunden**.

Das Problem ist, dass wir nicht durch eine ObservableCollection navigieren können wie etwa durch ein Recordset und seinen Methoden **MoveNext**, **MovePrevious** und so weiter. Wir können aber immerhin die Methode Item() mit dem gewünschten Index nutzen, um ein anderes Element zu referenzieren. Im folgenden Code-Beispiel weisen wir der Variablen **Kunde** etwa das zweite Element der Kundenliste zu:

```
Private Sub btnNaechster_Click(sender As Object, e As RoutedEventArgs)
    Kunde = Kunden.Item(1)
End Sub
```

Wenn wir das Projekt nun starten und auf die Schaltfläche **btnNaechster** klicken, wird allerdings immer noch der erste Datensatz angezeigt.

Der Grund ist schnell gefunden: Wir haben keinerlei Mechanismus implementiert, der dafür sorgen würde, dass Änderungen an dem in der Eigenschaft **Kunde** gespeicherten Element sich auf die Benutzeroberfläche auswirken beziehungsweise die Daten eines neu zugewiesenen **Kunde**-Elements angezeigt würden. Für diesen Fall ist die Schnittstelle **INotifyPropertyChanged** vorgesehen. Diese müssen wir in der Code behind-Klasse implementieren. Dazu fügen wir zunächst einen Verweis auf den Namespace **System.ComponentModel** hinzu:

```
Imports System.ComponentModel
```

Die Schnittstelle **INotifyPropertyChanged** müssen wir nun für die Klasse **frmKundendetails** implementieren. Dazu hängen wir in der Zeile unter der Definition der Klasse die folgende Zeile an:

```
Public Class frmKundendetails
    Implements INotifyPropertyChanged
    ...
End Class
```

Anschließend klicken Sie mit der rechten Maustaste auf **INotifyPropertyChanged**, wählen den Kontextmenü-Eintrag **Schnellaktionen und Refactorings...** aus und klicken dann auf **Schnittstelle implementieren** (siehe Bild 2). Daraufhin fügt Visual Studio die folgende Anweisung zur Klasse hinzu:

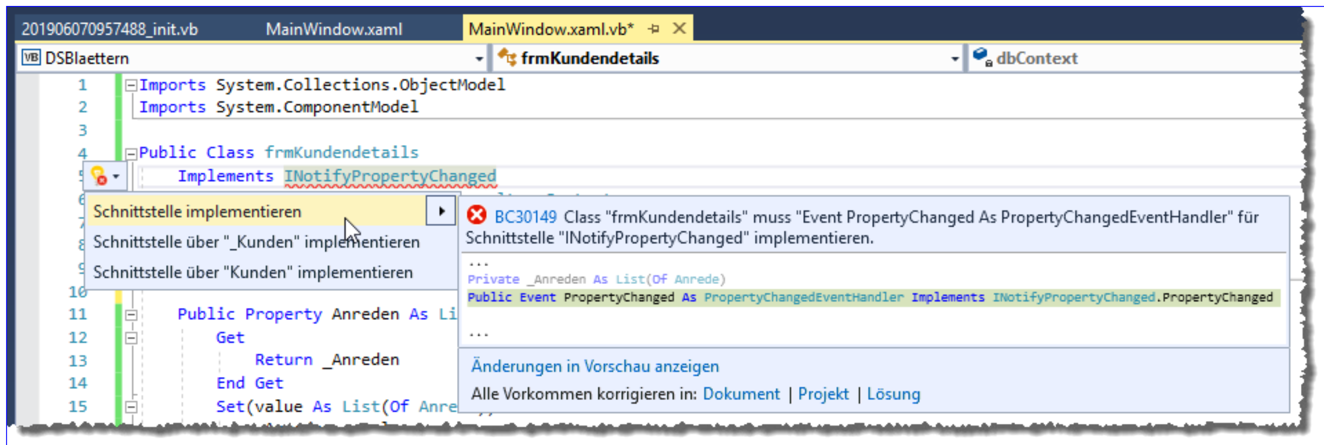


Bild 2: Implementieren der Schnittstelle **INotifyPropertyChanged**

```
Public Event PropertyChanged As PropertyChangedEventHandler Implements INotifyPropertyChanged.PropertyChanged
```

Nun müssen wir noch dafür sorgen, dass dieses Ereignis beim Ändern der Eigenschaft **Kunde** ausgelöst wird. Das erreichen wir durch Hinzufügen einer Zeile zur Definition der entsprechenden Property:

```
Public Property Kunde
    Get
        Return _Kunde
    End Get
    Set
        _Kunde = Value
        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("Kunde"))
    End Set
End Property
```

Wenn wir nun das Projekt neu starten und mit der Schaltfläche **btnNaechster** den Kunden wechseln wollen, wird dieser auch angezeigt.

Hinweis: Diese Änderung wurde bereits in den Code zum Erstellen der Code behind-Klasse von Access aus hinzugefügt.

Erster, vorheriger, nächster und letzter Datensatz

Damit können wir uns nun im Detail um die vier Schaltflächen **btnErster**, **btnVorheriger**, **btnNaechster** und **btnLetzter** kümmern. Den ersten Kunden finden wir leicht über die Methode **First** der **Kunden**-Auflistung. Also füllen wir die Methode **btnErster_Click** wie folgt:

```
Private Sub btnErster_Click(sender As Object, e As RoutedEventArgs)
    Kunde = Kunden.First
End Sub
```

Genauso leicht ist es, den letzten Datensatz anzusteuern. Dazu verwenden wir analog die **Last**-Methode:

```
Private Sub btnLetzter_Click(sender As Object, e As RoutedEventArgs)
    Kunde = Kunden.Last
End Sub
```

Für den vorherigen und nächsten Datensatz müssen wir ein wenig mehr investieren. Methoden wie **Previous** oder **Next** gibt die **ObservableCollection** nicht her. Wir wissen aber, dass wir einen beliebigen Eintrag mit der Eigenschaft **Item** und dem nullbasierten Indexwert ansteuern können. Wenn wir den Index des aktuellen Eintrags herausfinden können, brauchen wir nur noch eins hinzuzuzählen oder abzuziehen, um den Index des nächsten oder des vorherigen Datensatzes zu ermitteln. Den Index liefert die **IndexOf**-Eigenschaft der **Kunden**-Auflistung mit dem zu untersuchenden **Kunde**-Element als Parameter, also mit **Kunden.IndexOf(Kunde)**. Wenn wir diese Erkenntnisse kombinieren, erhalten wir die folgenden beiden Anweisungen für die Schaltflächen **btnVorheriger** und **btnNaechster**:

```
Private Sub btnVorheriger_Click(sender As Object, e As RoutedEventArgs)
    Kunde = Kunden.Item(Kunden.IndexOf(Kunde) - 1)
End Sub

Private Sub btnNaechster_Click(sender As Object, e As RoutedEventArgs)
    Kunde = Kunden.Item(Kunden.IndexOf(Kunde) + 1)
End Sub
```

Damit können wir nun wie gewünscht durch die Datensätze navigieren. Leider erhalten wir noch eine Fehlermeldung, wenn wir uns auf dem ersten Element befinden und die Schaltfläche **btnVorheriger** betätigen oder wenn wir uns auf dem letzten Element befinden und dann auf **btnNaechster** klicken. Der Grund ist, dass es vor dem ersten und hinter dem letzten Element keine weiteren Elemente mehr gibt. Dem können wir vorbeugen, indem wir die Schaltflächen **btnVorheriger** und **btnNaechster** deaktivieren, wenn aktuell das erste oder letzte Element angezeigt wird.

Dazu verwenden wir eine kleine Prozedur, die prüft, welchen Index der aktuell angezeigte Kunde aufweist. Dazu fragen wir die Eigenschaft **IndexOf** ab und speichern das Ergebnis in einer Variablen namens **lngIndex**. Danach aktivieren wir die Schaltfläche **btnVorheriger**, wenn **lngIndexID** nicht **0** ist und **btnNaechster**, wenn **lngIndexID** nicht der Anzahl der Elemente von **Kunden** entspricht:

```
Private Sub SchaltflaechenAktivieren()
    Dim lngIndex As Long
    lngIndex = Kunden.IndexOf(Kunde)
    btnVorheriger.IsEnabled = (Not lngIndex = 0)
    btnNaechster.IsEnabled = (Not lngIndex = Kunden.Count - 1)
End Sub
```

Diese Prozedur rufen wir in allein vier bisher beschriebenen Ereignismethoden wie folgt aus:

Validieren mit VB und EDM

Im Artikel EDM: Validieren von Entitäten mit IDataErrorInfo haben wir bereits erläutert, wie Sie einer auf dem Entity Framework basierenden Anwendung eine Validierung hinzufügen können. Im vorliegenden Artikel schauen wir uns an, wie Sie die Validierung nicht unter C#, sondern unter VB realisieren und welche Änderungen in den Methoden zum Migrieren einer Access-Anwendung nach WPF/EDM notwendig sind, um die Validierung vorzubereiten oder gegebenenfalls auch direkt umzusetzen.

Für die Programmierung einer Validierung in den Klassen des Entity Data Models und der mit WPF definierten Benutzeroberfläche sind einige Schritte auf den verschiedenen Ebenen erforderlich. Der erste ist, die Entitätsklassen mit der Implementierung der Schnittstelle **IDataErrorInfo** zu versehen, der zweite die Definition der Anzeige der Validierungsmeldungen in der Benutzeroberfläche – und das geschieht durch entsprechende WPF-Elemente.

IDataErrorInfo – wo implementieren?

Im oben genannten Artikel haben wir unter C# die Schnittstelle **IDataErrorInfo** für partielle Klassen implementiert. Das bedeutet, dass wir zwei gleichnamige Teilklassen programmiert haben, wobei wir in der einen die Definition der Properties und in der anderen die Implementierung der Schnittstelle **IDataErrorInfo** eingefügt haben. Warum haben wir das dort so gemacht? Weil wir den Code für die Entitätsklassen aus dem Datenmodell der Datenbank generiert haben, was bedeutet, dass wir bei Änderungen des Datenmodells auch die Entitätsklassen neu generieren mussten. Wenn wir dann die Implementierung der Schnittstelle **IDataErrorInfo** in der automatisch generierten Entitätsklasse programmiert hätten, wäre diese bei jeder Aktualisierung wieder überschrieben worden. Deshalb haben wir diese Definition direkt in eigenen, partiellen Klassen programmiert. Partielle Klasse bedeutet dabei, dass wir einfach eine weitere Klasse gleichen Namens mit dem Schlüsselwort **Partial** erstellt und dort die Elemente für die Implementierung der Schnittstelle hineingeschrieben haben. Die Klassenbezeichnungen der per Generator erzeugten Entitätsklassen wurden dabei praktischerweise bereits mit dem Schlüsselwort Partial ausgezeichnet – was keinen Nachteil bringt, auch wenn es keine weiteren partiellen Klassen mehr gibt.

Nun verwenden wir in der aktuell in diesem Magazin beschriebenen Vorgehensweise eine Access-Datenbank als Vorlage für die Erstellung eines Entity Data Models und auch für die Erstellung von Fenstern auf Basis der Formulare der Access-Datenbank. Wir könnten also theoretisch auch gleich die Implementierung der Schnittstelle **IDataErrorInfo** in die Entitätsklassen schreiben – und hier direkt die im Datenmodell festgelegten Restriktionen wie keine Nullwerte et cetera eintragen. Allerdings kann es auch hier sein, dass Sie die Implementierung der Schnittstelle **IDataErrorInfo** selbst anpassen wollen. Also werden wir auch hier mit partiellen Klassen arbeiten und per Parameter beim Erstellen des Entity Data Models auf Basis der Tabellen der Access-Datenbank die Möglichkeit bieten, entweder nur das Entity Data Model zu erstellen oder auch noch die partiellen Klassen mit den Validierungsfunktionen neu zu generieren. Zunächst schauen wir uns allerdings an, wie die partiellen Klassen unter VB überhaupt aussehen sollen.

Partielle Klasse mit IDataErrorInfo erstellen

Das Anlegen der zusätzlichen partiellen Klasse etwa für die Klasse **Kunde** erfordert zwei Schritte. Als Erstes fügen wir der eigentlichen Klasse das Schlüsselwort **Partial** als erstes Schlüsselwort noch vor **Public** hinzu:

```
<Table("Kunden")>
Partial Public Class Kunde
    ...
End Class
```

Danach können wir die zweite partielle Klasse erstellen:

```
Partial Public Class Kunde
    ...
End Class
```

Für diese Klasse wollen wir nun die Schnittstelle **IDataErrorInfo** implementieren. Dazu benötigen wir zunächst die Bibliothek System.**ComponentModel**, die wir per **Imports**-Anweisung einbinden:

```
Imports System.ComponentModel
```

Danach fügen Sie in der Zeile unter der Klassendefinition die Anweisung zur Implementierung der Schnittstelle hinzu:

```
Partial Public Class Kunde
    Implements IDataErrorInfo
    ...
```

Wenn Sie mit der rechten Maustaste auf **IDataErrInfo** klicken, erscheint der Kontextmenü-Eintrag **Schnellaktionen und Refactorings...**, den Sie anklicken und danach den nun erscheinenden Befehl **Schnittstelle implementieren** anklicken (siehe Bild 1).

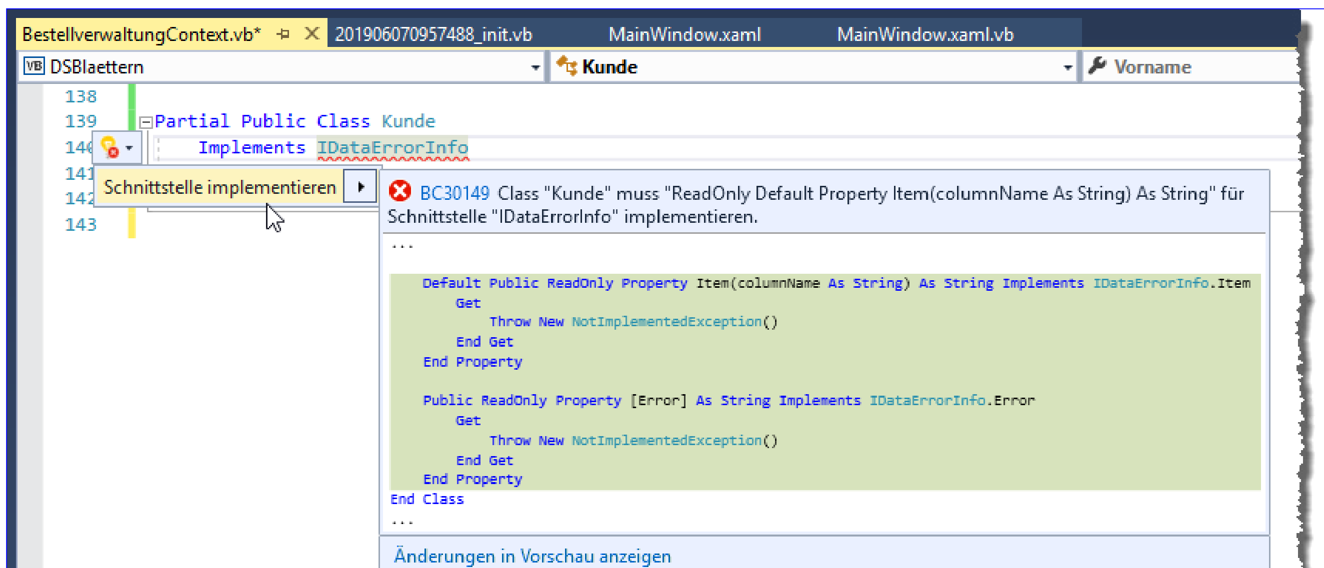


Bild 1: Automatisches Implementieren der Schnittstelle

Das füllt die Klasse mit den beiden Properties **Item** und **[Error]**:

```
Partial Public Class Kunde
    Implements IDataErrorInfo

    Default Public ReadOnly Property Item(columnName As String) As String Implements IDataErrorInfo.Item
        Get
            Throw New NotImplementedException()
        End Get
    End Property

    Public ReadOnly Property [Error] As String Implements IDataErrorInfo.Error
        Get
            Throw New NotImplementedException()
        End Get
    End Property
End Class
```

Uns interessiert dabei speziell die **Property** namens **Item**. Diese liefert mit dem Parameter **columnName** den Namen des Feldes, für das eine Validierung durchgeführt werden soll. Um eine Validierung für das Feld **Vorname** durchzuführen, die prüfen soll, ob das Feld **Vorname** leer ist, fügen Sie die folgende **Select Case**-Bedingung mit dem **Case**-Zweig für den Feldnamen **Vorname** hinzu.

Die Prüfung stellt mit der Methode **IsNullOrEmpty** fest, ob **Vorname** leer ist. In diesem Fall wird der Rückgabeparameter **strErrorMessage** mit einer entsprechenden Fehlermeldung gefüllt:

```
Default Public ReadOnly Property Item(columnName As String) As String Implements IDataErrorInfo.Item
    Get
        Dim strErrorMessage As String = ""
        Select Case columnName
            Case "Vorname"
                If (String.IsNullOrEmpty(Vorname)) Then
                    strErrorMessage = "Bitte geben Sie einen Vornamen ein."
                End If
        End Select
        Return strErrorMessage
    End Get
End Property
```

Nun stellt sich nur noch die Frage, wodurch diese Methode aufgerufen wird und wie wir die Fehlermeldung aus **strErrorMessage** verarbeiten.

Auslösen der Validierung

Dass für das Feld **Vorname** eine Validierung ausgelöst wird, legen wir in der Definition des an dieses Feld gebundenen Steuerelements fest, in diesem Fall des Textfeldes **txtVorname**. Hier erweitern wir den Inhalt der Bindung für das Attribut **Text** um **ValidatesOnDataErrors=True**:

```
<TextBox x:Name="txtVorname" Text="{Binding Kunde.Vorname, ValidatesOnDataErrors=True}" Height="21" Margin="118,58,0,0"
Width="221"/>
```

Damit allein wird nun bereits die Validierung ausgelöst, das heißt, wenn das Feld **Vorname** angezeigt wird, ruft dies auch die Methode **Item** der Implementierung von **IDataErrorInfo** auf. Das Ergebnis ist allerdings noch keine Anzeige der Meldung, sondern zunächst nur das rote Einrahmen des Feldes, das nicht erfolgreich validiert werden konnte (siehe Bild 2).

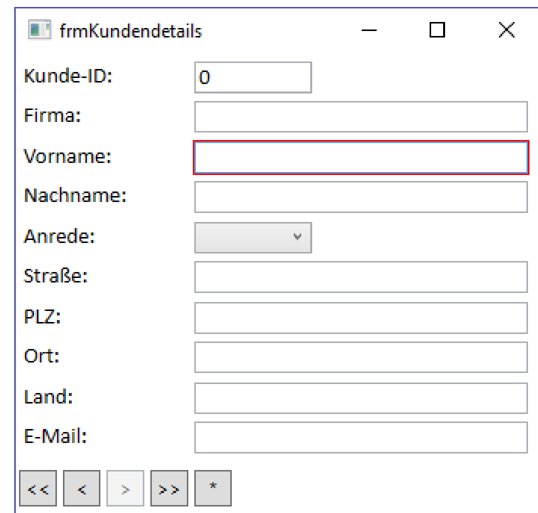


Bild 2: Die Validierung funktioniert bereits.

Wir wollen allerdings noch konkretere Hinweise auf den Grund für die farbige Markierung liefern, damit der Benutzer weiß, was zu tun ist. Also verwenden wir den Setter, den wir auch schon im oben genannten Artikel genutzt haben, um die Fehlermeldung an entsprechender Stelle unterzubringen. Dazu erweitern wir den Code für das **Style**-Element in der **.xaml**-Datei wie folgt:

```
<Style TargetType="{x:Type TextBox}">
    ...
    <Setter Property="Validation.ErrorTemplate">
        <Setter.Value>
            <ControlTemplate>
                <DockPanel>
                    <Border Background="Red" DockPanel.Dock="right" Margin="5,0,0,0" Width="20" Height="20"
                            CornerRadius="10" ToolTip="{Binding ElementName=customAdorner,
                            Path=AdornedElement.(Validation.Errors)[0].ErrorContent}">
                        <TextBlock Text="!" VerticalAlignment="center" HorizontalAlignment="center"
                            FontWeight="Bold" Foreground="white" />
                    </Border>
                    <AdornedElementPlaceholder Name="customAdorner" VerticalAlignment="Center" >
                        <Border BorderBrush="red" BorderThickness="1" />
                    </AdornedElementPlaceholder>
                </DockPanel>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

Danach erscheint die Meldung, die wir für das Feld **Vorname** definiert haben, wie in Bild 3.

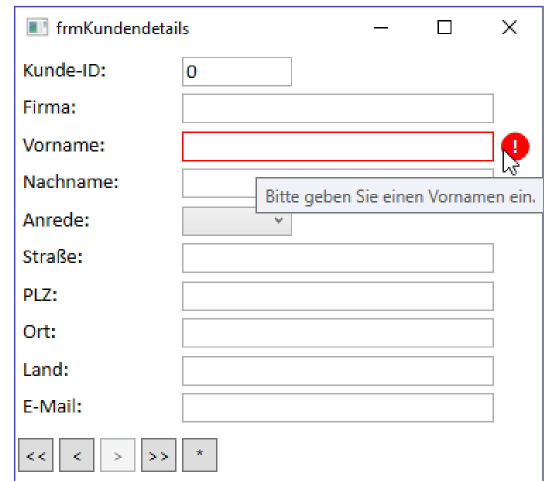


Bild 3: Validierung mit Hinweismeldung

Validierung von Kombinationsfeldern

Für die Validierung des Kombinationsfeldes **Anrede** fügen Sie in der **.xaml**-Datei für die Definition des **Style**-Elements für den Typ **ComboBox** genau die gleiche Property namens **Validation.ErrorTemplate** wie für den Typ **TextBox** hinzu:

```
<Style TargetType="{x:Type ComboBox}">
    ...
    <Setter Property="Validation.ErrorTemplate">
    ...
    </Setter>
</Style>
```

Außerdem passen Sie in der Definition des **ComboBox**-Elements das Attribut **SelectedItem** so an, dass die Bindung mit der Option **ValidatesOnDataErrors=True** erfolgt:

```
<ComboBox x:Name="cboAnredeID" Padding="0" Height="21" Margin="118,111,0,0" Width="78"
    ItemsSource = "{Binding Anreden}"
    SelectedItem = "{Binding Kunde.Anrede, ValidatesOnDataErrors=True}"
    ...
</ComboBox>
```

Schließlich erweitern wir die Implementierung der Schnittstelle **IDataErrorInfo** noch um einen entsprechenden **Case**-Zweig. Hier verwenden wir allerdings die **IsNothing**-Funktion, um zu prüfen, ob für die Eigenschaft **Anrede** eines der Elemente der Auflistung **Anreden** zugewiesen wurde:

```
Default Public ReadOnly Property Item(columnName As String) As String Implements IDataErrorInfo.Item
    Get
        Dim strErrorMessage As String = ""
        Select Case columnName
            ...
            Case "Anrede"
                If (IsNothing(Anrede)) Then
                    strErrorMessage = "Bitte wählen Sie eine Anrede aus."
                End If
        End Select
        Return strErrorMessage
    End Get
```

```
End Property
```

Verlassen des Datensatzes ohne erfolgreiche Validierung unterbinden

Nun müssen wir noch sicherstellen, dass der Benutzer den Datensatz nicht verlassen kann, ohne dass alle erforderlichen Daten eingegeben wurden. Dazu haben wir im oben genannten Artikel eine Technik kennengelernt, die auf den Ergebnissen unserer Validierung aufbaut und davon abhängig Steuerelemente aktiviert oder deaktiviert. Dabei haben wir für die betroffene Schaltfläche **DataTrigger**-Elemente als **Style**-Elemente festgelegt und für jeden Trigger eine Bedingung formuliert, unter der eine bestimmte Eigenschaft auf einen Wert eingestellt werden soll – in diesem Fall etwa der Wert der Validierung eines oder mehrerer gebundener Steuerelemente und damit verbunden die Eigenschaft **IsEnabled** der zu deaktivierenden Schaltfläche.

Daraus leiten wir für unsere Steuerelemente, hier etwa für die Schaltfläche **btnErster**, die entsprechende **Style**-Definition ab. Die **Style.Triggers**-Auflistung nimmt dabei mehrere **DataTrigger**-Elemente auf. Diese referenzieren jeweils die Eigenschaft **Validation.HasError** eines der Steuerelemente, in der nachfolgenden verkürzten Form etwa die Textfelder **txtFirma** oder **txtEMail**. Hat **Validation.HasError** den Wert **True**, wird der eingefasste **Setter** aktiviert, der hier den Wert der Eigenschaft **IsEnabled** auf **True** einstellt:

```
<Button x:Name="btnErster" Content="&lt;&lt;" Click="btnErster_Click" Width="25" Margin="2,2,2,2">
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <Style.Triggers>
                <DataTrigger Binding="{Binding ElementName=txtFirma, Path=(Validation.HasError)}" Value="True">
                    <Setter Property="IsEnabled" Value="False"></Setter>
                </DataTrigger>
                ...
                <DataTrigger Binding="{Binding ElementName=txtEMail, Path=(Validation.HasError)}" Value="True">
                    <Setter Property="IsEnabled" Value="False"></Setter>
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </Button.Style>
</Button>
```

Diesen Code müssten wir für jede betroffene Schaltfläche festlegen, und zwar mit **DataTrigger**-Elementen für alle zu validierenden Steuerelemente. Da der Benutzer für ein nicht erfolgreich validiertes Element keine der Schaltflächen **btnErster**, **btnVorheriger**, **btnNaechster**, **btnLetzter** und **btnNeu** anklicken können soll, benötigen wir diesen Code also fünf Mal. Das ist schon eine Menge Code, den wir vermutlich noch kürzer fassen können. Vorher schauen wir uns allerdings an, ob das Fenster nun wie gewünscht arbeitet. Wie Bild 4 zeigt, ist das nicht der Fall.

Bild 4: Die Schaltflächen **btnVorheriger** und **btnNaechster** werden nicht deaktiviert.

Wir haben hier für einen bereits vorhandenen Datensatz das Feld **Ort** geleert. Dadurch werden zwar die Schaltflächen **btnErster**, **btnLetzter** und **btnNeu** deaktiviert, was korrekt ist. Aber warum sind die Schaltflächen **btnVorheriger** und **btnNaechster** noch aktiviert? Ganz einfach: Weil wir noch die Methode **SchaltflaechenAktivieren** aufrufen, sobald der Benutzer eine der Schaltflächen anklickt. Und diese aktiviert in diesem Fall die beiden Schaltflächen **btnVorheriger** und **btnNaechster** wieder:

```
Private Sub SchaltflaechenAktivieren()
    btnVorheriger.IsEnabled = Not (Kunde Is Kunden.First)
    btnNaechster.IsEnabled = Not (Kunde Is Kunden.Last)
End Sub
```

Wir lernen an dieser Stelle also auch, dass die Zuweisung der Eigenschaftswerte durch den VB-Code erst nach der Zuweisung durch den XAML-Code erfolgt. Diese Baustelle können wir durch die folgende Anpassung schließen. Dazu hinterlegen wir zwei private Eigenschaften, die speichern, ob gerade der erste beziehungsweise der letzte Kunde angezeigt wird:

```
Private _ErsterKunde As Boolean
Private _LetzterKunde As Boolean
```

Für diese legen wir dann zwei öffentliche Properties an:

```
Public Property ErsterKunde As Boolean
    Get
        Return _ErsterKunde
    End Get
    Set
        _ErsterKunde = Value
        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("ErsterKunde"))
    End Set
End Property

Public Property LetzterKunde As Boolean
    Get
        Return _LetzterKunde
    End Get
    Set
        _LetzterKunde = Value
        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("LetzterKunde"))
    End Set
End Property
```

Danach brauchen wir nur noch diese beiden Werte für das Attribut **IsEnabled** der beiden Elemente **btnVorheriger** und **btnNaechster** zu hinterlegen, was wir durch entsprechende **DataTrigger**-Elemente erledigen:

Entity Framework: Der ChangeTracker

Wenn Sie Daten etwa aus den Tabellen einer Datenbank in ein Entity Data Model geladen haben, finden Sie dort einige Funktionen für den Umgang mit den enthaltenen Daten vor. Ein sehr wichtiges Element ist dabei der ChangeTracker. Auch wenn Sie mit der SaveChanges-Methode automatisch alle Änderungen im Entity Data Model erkennen und diese in die Datenbank übertragen können, so treten doch Fälle auf, in denen Sie zuvor prüfen wollen, welche Änderungen überhaupt im Entity Data Model vorgenommen wurden – und ob diese in die Datenbank übernommen oder vielleicht sogar verworfen werden sollen.

Wenn Sie eine Anwendung bauen, wie wir sie in den letzten Ausgaben verwendet haben, dann bezieht diese ihre Daten aus den Tabellen eines SQL Servers und stellt diese in einem Entity Data Model bereit. Die Elemente des Entity Data Models binden Sie dann an die Benutzeroberfläche und die dort befindlichen Steuerelemente.

Nun können nach dem Anzeigen der Daten in einem Bearbeitungsfenster verschiedene Fälle auftreten: Sie ändern ein oder mehrere Felder eines Datensatzes, fügen einen neuen Datensatz hinzu oder löschen einen Datensatz. Diese Änderungen liegen dann im Entity Data Model vor, aber werden nicht automatisch in die Datenbank übertragen. Das erledigen Sie erst, wenn Sie die **SaveChanges**-Methode des Datenbankkontextes aufrufen. **SaveChanges** ist effizient, denn es prüft, ob Änderungen vorliegen und speichert nur die geänderten Elemente in der Datenbank.

Was aber geschieht, wenn Sie beispielsweise erfahren wollen, ob an einem bestimmten Datensatz Änderungen vorgenommen wurden? Für ein Fenster mit Minimalanforderungen benötigen Sie diese Information nicht, denn Sie können die Änderungen ja einfach in der Datenbank speichern. Wenn Sie dem Benutzer jedoch ein paar weitere Features wie etwa eine Schaltfläche zum Verwerfen der Änderungen zur Verfügung stellen wollen, kann es schon interessant sein, ob der Benutzer überhaupt schon Änderungen am aktuellen Datensatz vorgenommen hat. Dabei könnten Sie die Information, ob der Datensatz bereits geändert wurde, etwa dazu nutzen, die **Abbrechen**- oder **Verwerfen**-Schaltfläche nur zu aktivieren, wenn der Benutzer bereits Änderungen vorgenommen hat – oder wenn er soeben einen neuen Datensatz angelegt hat, der ja mitunter auch verworfen werden könnte.

Und hier kommt der **ChangeTracker** von Entity Framework ins Spiel. Dabei handelt es sich um eine Klasse, die zum Beispiel den Zugriff auf alle Elemente mit einem bestimmten Status zulässt – beispielsweise geändert oder nicht geändert. In diesem Artikel schauen wir uns nun an, wie Sie mit dem **ChangeTracker** von Entity Framework arbeiten können.

Beispielprojekt

Als Beispielprojekt verwenden wir ein Projekt auf Basis der Vorlage **Visual Basic|Windows Desktop|WPF-App** namens **ChangeTracker**. Dieser haben wir ein Entity Data Model mit der Methode aus der Artikelreihe **Von Access zu Entity Framework** hinzugefügt und eine SQL Server-Datenbank damit erstellt. Um die SQL Server-Datenbank auf ihrem Rechner zu erstellen, rufen Sie die Anweisung **Update-Database** in der Paket-Manager-Konsole auf. Dadurch wird eine Datenbank auf Basis der

in der Datei **App.config** angegebenen Verbindungszeichenfolge erstellt, die Sie gegebenenfalls noch anpassen können. Die verschiedenen Beispielcodes der folgenden Abschnitte finden Sie hinter den Ereignismethoden der Schaltflächen des Fensters **MainWindow.xaml**.

Die ChangeTracker-Klasse

Die **ChangeTracker**-Klasse referenzieren über die gleichnamige Eigenschaft der Datenbankkontext-Klasse, die in unserem Beispielen meist **dbContext** genannt wird. Mit dieser Klasse können Sie den Zustand jedes der Elemente des aktuellen Datenbankkontexts nachverfolgen. Jeder der Einträge hat einen Wert für die Eigenschaft **EntityState**. Die möglichen Werte lauten:

- **Added**: Hinzugefügtes Element, das noch nicht in der Datenbank existiert
- **Modified**: Geändertes Element, dessen Änderungen noch nicht in die Datenbank übertragen wurden
- **Deleted**: Gelöschtes Element, das noch in der Datenbank existiert
- **Unchanged**: Nicht geändertes Element
- **Detached**: Nicht verbundenes, also nicht zu einem **DbSet** gehörendes Element. Dieses wird auch nicht vom ChangeTracker verfolgt.

Beispiele zur ChangeTracker-Klasse

Wir schauen uns die verschiedenen Zustände an einfachen Beispielen an. Dazu fügen wir der Klasse **MainWindow.xaml.vb** zunächst die folgenden Namespaces hinzu:

```
Imports System.Data.Entity
Imports System.Data.Entity.Infrastructure
```

Danach erstellen wir die folgende Ereignismethode für die erste Schaltfläche **btnUnveraendert**. Hier füllen wir die Variable **dbContext** mit einer neuen Instanz des Datenbankkontextes und füllen das Objekt **Kunde** des Typs **Kunde** mit dem ersten Eintrag der Liste **Kunden** des Datenbankkontextes:

```
Private Sub btnUnveraendert_Click(sender As Object, e As RoutedEventArgs)
    Dim dbContext As BestellverwaltungContext
    Dim Element As EntityEntry
    Dim Kunde As Kunde
    dbContext = New BestellverwaltungContext
    Kunde = dbContext.Kunden.First()
```

Danach folgt der erste Zugriff auf den **ChangeTracker**. Dabei durchlaufen wir dessen **Entries**-Auflistung und referenzieren das aktuelle Element jeweils mit der Variablen **Element**, die übrigens den Typ **EntityEntry** aufweist (dafür auch der Verweis auf den Namespace). Anschließend geben wir für jeden Eintrag zwei Informationen aus – den Namen des Elementtyps und den

Zustand. Um den Namen des Elementtyps zu erhalten, referenzieren wir zunächst die in **Element** enthaltene Entität mit der Eigenschaft **Entity** und ermitteln den Namen des Typs mit **GetType().Name**. Den Zustand dieses Eintrags liefert die Eigenschaft **State** des **EntityEntry**-Elements:

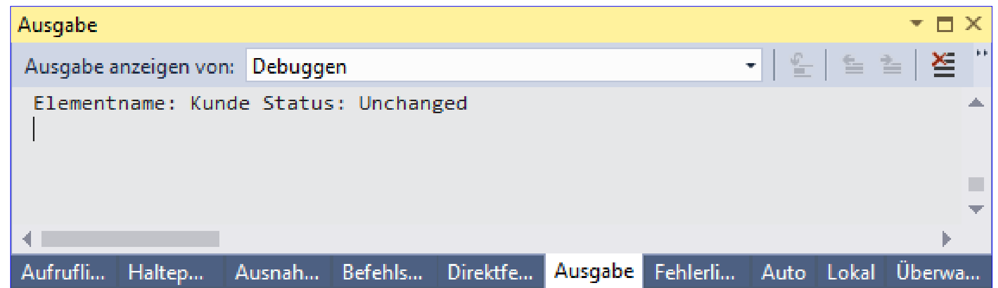


Bild 1: Ausgabe von Typ und Status einer Entität

```
    For Each Element In dbContext.ChangeTracker.Entries
        Debug.Print("Elementname: " + Element.Entity.GetType().Name + " Status: " + Element.State.ToString())
    Next
End Sub
```

Das Ergebnis sieht schließlich wie in Bild 1 aus. Der Typ **Kunde** wird korrekt erkannt und auch der Status entspricht mit **Unchanged** den Erwartungen.

ChangeTracker verfolgt alle Elementtypen

Wir fügen eine Variable für ein Element eines weiteren Elementtypen namens **Anrede** hinzu und lesen auch hier den ersten Eintrag aus – in diesem Fall aus der Liste **Anreden**:

```
Dim Anrede As Anrede
...
Anrede = dbContext.Anreden.First()
```

Die Ausgabe im Ausgabefenster sieht nach dem erneuten Start der Anwendung und dem Betätigen der Schaltfläche wie folgt aus:

```
Elementname: Kunde Status: Unchanged
Elementname: Anrede Status: Unchanged
```

Wir sehen also, dass die **Entries**-Auflistung der **ChangeTracker**-Klasse Elemente aller Typen umfasst – sie müssen nur in den Speicher geladen sein.

Ein Element löschen

Im nächsten Beispiel wollen wir untersuchen, was beim Löschen eines Eintrags geschieht. Dazu legen wir eine neue Schaltfläche namens **btnGeloeschterEintrag** hinzu, für deren **Click**-Ereignis wir die folgende Ereignismethode hinterlegen:

```
Private Sub btnGeloeschterEintrag_Click(sender As Object, e As RoutedEventArgs)
    Dim dbContext As BestellverwaltungContext
```

```
    Dim Bestellposition As Bestellposition
    Dim Element As DbEntityEntry
    dbContext = New BestellverwaltungContext
    Bestellposition = dbContext.Bestellpositionen.First()
    dbContext.Bestellpositionen.Remove(Bestellposition)
    For Each Element In dbContext.ChangeTracker.Entries
        MessageBox.Show("Elementname: " + Element.Entity.GetType().Name + vbCrLf + " Status: " _
            + Element.State.ToString())
    Next
End Sub
```

Hier laden wir nun das erste Element der Auflistung **Bestellpositionen**, da wir diese ohne Probleme löschen können – die Einträge haben keine Beziehungen, deren Restriktion ein Löschen verhindern könnte. Dann löschen wir dieses Element mit der **Remove**-Methode aus der Auflistung **Bestellpositionen**. Anschließend rufen wir wieder die gleiche **For Each**-Schleife über die **Entries**-Auflistung. Diesmal liefert die **State**-Eigenschaft den Wert **Deleted**.

Ein Element hinzufügen

Nun erstellen wir ein neues **Kunde**-Element, füllen seine Eigenschaften und fügen das **Kunde**-Element mit der **Add**-Methode zur Auflistung **Kunden** hinzu:

```
Kunde = New Kunde
With Kunde
    .Vorname = "André"
    .Nachname = "Minhorst"
    .AnredeID = 1
    .Strasse = "Borkhofer Str. 17"
    .PLZ = "47137"
    .Ort = "Duisburg"
    .Land = "Deutschland"
    .EMail = "andre@minhorst.com"
End With
dbContext.Kunden.Add(Kunde)
Element = dbContext.ChangeTracker.Entries.First
MessageBox.Show("Elementname: " + Element.Entity.GetType().Name + vbCrLf + " Status: " + Element.State.ToString())
```

Hier lautet der Wert von **State** nun **Added**.

Ein Element ändern

Wir legen noch eine neue Schaltfläche an, mit der wir ein bestehendes Element ändern. Dazu weisen wir dem Feld **Vorname** des **Kunde**-Elements einen anderen Wert zu. Da wir wissen, dass wir gerade nur einen Eintrag geändert haben, durchlaufen wir keine Schleife mehr, sondern greifen über die **First**-Eigenschaft auf das geänderte Element zu:

```
Kunde = dbContext.Kunden.First()
Kunde.Vorname = "André"
Element = dbContext.ChangeTracker.Entries.First
MessageBox.Show("Elementname: " + Element.Entity.GetType().Name + vbCrLf + " Status: " + Element.State.ToString())
```

In diesem Fall liefert die Eigenschaft **State** den Wert **Modified**.

Nicht verbundene Elemente

Es kann auch sein, dass Sie ein Element neu erstellen, aber dieses nicht mit der **Add**-Methode zur Auflistung des entsprechenden Elementtyps hinzufügen. Um zu untersuchen, welchen Zustand dieses Element hat und ob es vom **ChangeTracker** erfasst wird, müssen wir hier hier einen anderen Weg gehen. Wir finden nämlich, wie die zweite **MessageBox** zeigt, keinen Eintrag in der **ChangeTracker**-Liste **Entries** vor – das Ergebnis der **Count**-Eigenschaft ist **0**. Um ein **DbEntityEntry**-Element auf Basis des neuen **Kunde**-Objekts zu erstellen, greifen wir nun nicht auf die **Entries**-Liste der **ChangeTracker**-Klasse zu, sondern auf die **Entry**-Eigenschaft von **dbContext** für das **Kunde**-Objekt. Für dieses erhalten wir dann als **State** den Wert **Detached**, also nicht verbunden:

```
Dim dbContext As BestellverwaltungContext
Dim Kunde As Kunde
Dim Element As DbEntityEntry
dbContext = New BestellverwaltungContext
Kunde = New Kunde
With Kunde
    .Vorname = "André"
    ...
End With
Element = dbContext.Entry(Kunde)
MessageBox.Show("Elementname: " + Element.Entity.GetType().Name + vbCrLf + " Status: " + Element.State.ToString())
MessageBox.Show("Anzahl Elemente Changetracker: " + dbContext.ChangeTracker.Entries.Count().ToString())
```

Unterschied zwischen Entry(Element) und ChangeTracker.Entries

In den vorherigen Beispielen haben Sie gelernt, dass wir das **DbEntityEntry**-Objekt zu einem Element des Entity Data Models sowohl über die Eigenschaft **Entry** von **dbContext** holen können als auch über die Einträge der Auflistung **Entries** der **ChangeTracker**-Klasse. Welche der beiden Möglichkeiten Sie nutzen, hängt vom Anwendungszweck ab. Wenn Sie etwa alle Elemente mit bestimmten Eigenschaften in einer Schleife durchlaufen wollen, nutzen Sie **Entries**. Wenn Sie vorher schon einen Verweis auf das zu untersuchende Element haben, können Sie dieses auch direkt etwa über **Entry(Kunde)** referenzieren. Die Werte der Eigenschaften wie etwa **State** bleiben davon unberührt.

Geänderte Elemente untersuchen

Wir haben bereits gesehen, dass geänderte Elemente mit dem **State**-Wert **Modified** versehen werden. Nun interessiert uns noch, wie wir herausfinden, welche Eigenschaft der Elemente geändert wurden. Dazu ermitteln wir wieder das erste Element der **Kunden**-Auflistung und speichern es in der Variablen **Kunde**. Wir ändern den Wert des Feldes **Vorname** und weisen das

Änderungen erkennen und verwerfen

Unter Access können Sie beispielsweise mit der Escape-Taste die aktuellen ungespeicherten Änderungen an einem Datensatz verwerfen. Das Formular zeigt dann direkt die Daten an, die beim letzten Speichern in den gebundenen Feldern enthalten waren. Dieses Verhalten wollen wir auch für Fenster abbilden, die an ein Element gebunden sind und die Daten in einer Detailansicht anzeigen. Wir wollen erkennen, ob der Benutzer Änderungen an einem Datensatz vorgenommen hat und in Abhängigkeit davon eine Rückgängig-Schaltfläche aktivieren oder deaktivieren. Durch einen Klick auf die Schaltfläche sollen die vorherigen Werte wiederhergestellt werden. Wie das gelingt, zeigt der vorliegende Beitrag.

Ein WPF-Fenster bietet nicht automatisch eine **Dirty**-Eigenschaft wie ein gebundenes Access-Formular. Eine solche Funktion müssen wir selbst zu einem Fenster hinzufügen. Das Ergebnis dieses Artikels soll etwa wie in Bild 1 aussehen. Das Ändern eines Eintrags und das Verlassen dieses Eintrags sollen dafür sorgen, dass die **Rückgängig**-Schaltfläche aktiviert wird.

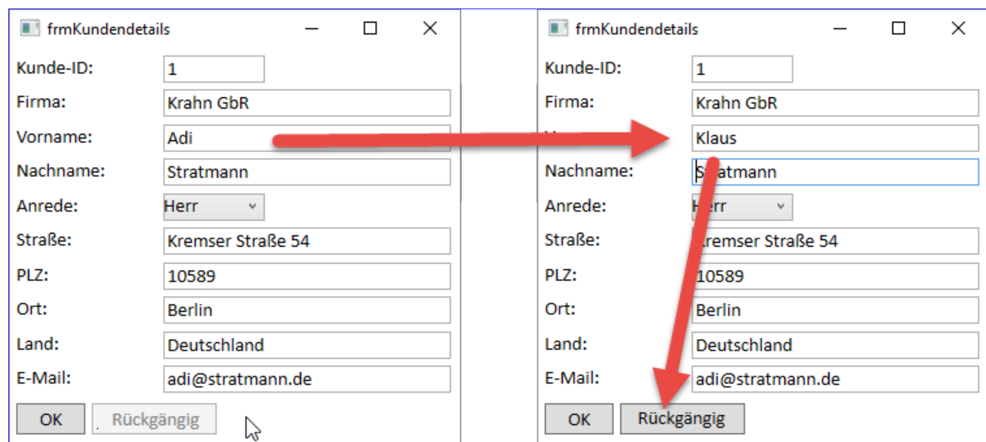


Bild 1: Aktivieren der **Abbrechen**-Schaltfläche nach einer Änderung

Betätigt der Benutzer dann die **Rückgängig**-Schaltfläche, soll der Datensatz auf die Version beim Laden zurückgesetzt werden (siehe Bild 2).

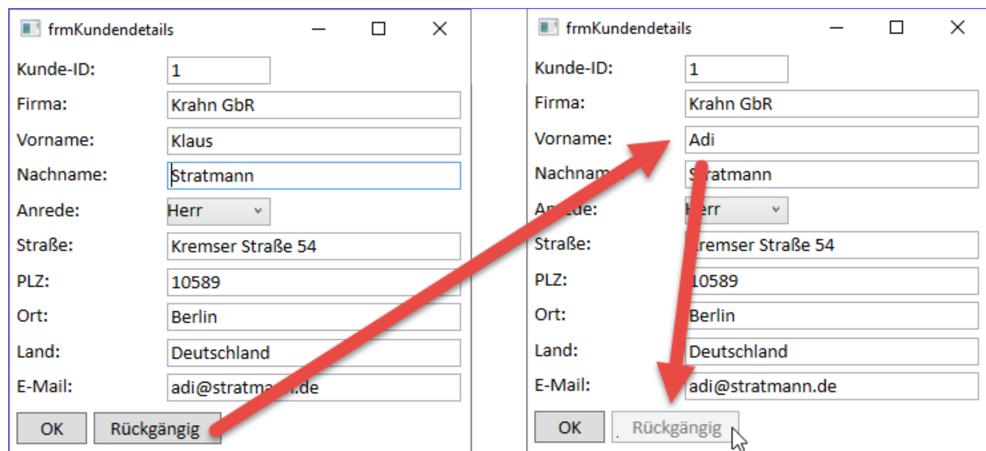


Bild 2: Zurücksetzen des Datensatzes nach Betätigen der **Abbrechen**-Schaltfläche

Vorbereitung

Wir nutzen wieder eine Anwendung auf Basis der Vorlage **Visual Basic|Windows Desktop|WPF-App (.NET Framework)**. Dieser fügen wir ein neues Element des Typs **ADO.NET Entity Data Model** hinzu und legen für dieses den Namen **BestellverwaltungContext** fest. Dann fügen wir mit den Methoden der Datenbankdatei **Access-**

ZuWPFFormulare.accdb (siehe dortigen Readme-Bericht) ein Fenster auf Basis des Formulars **frmKundendetails** hinzu. Der Entwurf dieses Fensters sieht anschließend wie in Bild 3 aus.

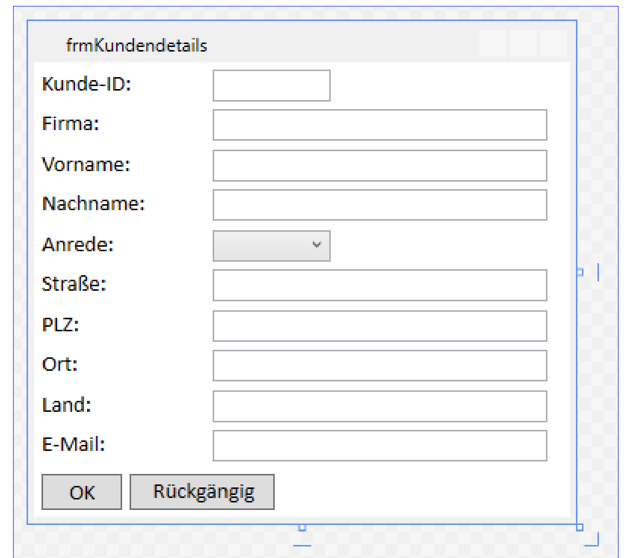


Bild 3: Entwurf des Fensters **frmKundendetails**

Nun wollen wir erreichen, dass die **Rückgängig**-Schaltfläche beim Anzeigen des Fensters deaktiviert ist und erst aktiviert wird, wenn der Benutzer bei mindestens einem Feld eine Änderung vorgenommen hat. Wenn der Benutzer die dann aktivierte Schaltfläche **Rückgängig** betätigt, sollen die Änderungen rückgängig gemacht werden beziehungsweise die Werte zum Zeitpunkt des Öffnens des Fensters wieder eingestellt werden.

Speichern beim Schließen

Wir beginnen mit den einfachen Teil: Die Schaltfläche **btnOK** soll den geänderten Datensatz in die Datenbank übertragen. Das erledigen wir mit der folgenden Ergänzung der Ereignismethode **btnOK_Click**:

```
Private Sub cmdOK_Click(sender As Object, e As RoutedEventArgs)
    dbContext.SaveChanges()
    Close()
End Sub
```

Ereignis beim Ändern auslösen

Dann wollen wir ein Ereignis finden, das durch das Ändern eines Eintrags und das Verlassen des Feldes ausgelöst wird. Das scheint uns ausreichend zu sein – wir müssen ja nicht nach der Eingabe eines jeden Buchstaben prüfen, ob sich etwas geändert hat. Der erste Kandidat für eine passende Ereignismethode heißt **LostFocus**. Also fügen wir das Attribut zum XAML-Code hinzu.

```
<TextBox x:Name="txtFirma" Text="{Binding Kunde.Firma}" Height="21" Margin="118,31,0,0" Width="221" LostFocus="txtFirma_
LostFocus"/>
```

Wenn wir **LostFocus="** eingeben und dann die Tabulator-Taste betätigen, wird automatisch der Methodenname **txtFirma_LostFocus** vorgeschlagen. Durch Betätigen von **F12** landen wir dann direkt bei dieser Methode im Code behind-Modul.

Mit der folgenden Testanweisung können wir gleich nach dem Starten der Anwendung testen, ob das Verlassen des Textfeldes **txtFirma** das Ereignis auslöst:

```
Private Sub txtFirma_LostFocus(sender As Object, e As RoutedEventArgs)
    MessageBox.Show("LostFocus")
End Sub
```

Herausfinden, ob eine Änderung vorliegt

Dieser Ereignismethode wollen wir nun Code hinzufügen, mit dem wir prüfen können, ob eine Änderung im aktuell angezeigten Element des Typs **Kunde** vorliegt.

Dazu benötigen wir zunächst einen Verweis auf den Namespace **System.Data.Entity.Infrastructure**, den wir im Kopf des Code behind-Moduls einfügen. **System.Data.Entity** benötigen wir später auch noch:

```
Imports System.Data.Entity.Infrastructure
Imports System.Data.Entity
```

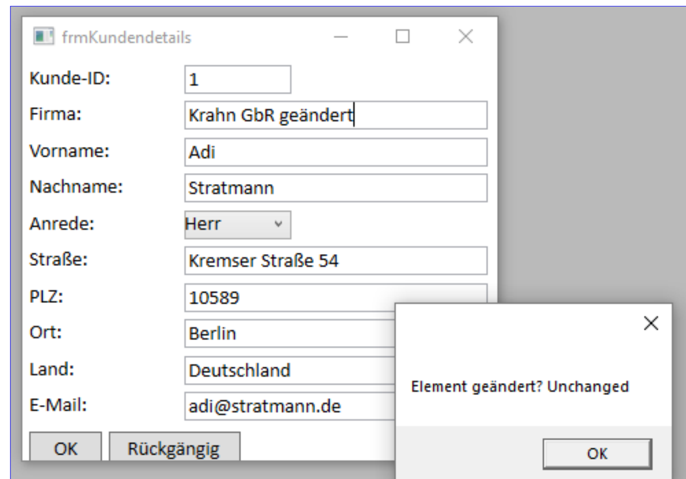


Bild 4: Aktueller Zustand des Elements

Danach erweitern wir die Ereignismethode wie folgt: Wir fügen eine Variable des Typs **DbEntityEntry** namens **GeaendertesElement** hinzu. Dieses füllen wir über die Auflistung **Entries** des **ChangeTracker**-Objekts, das dem aktuell in Kunde gespeicherten Element entspricht. Dieser Auflistung entnehmen wir mit der **First**-Eigenschaft das erste und einzige Element. Dann geben wir in einer weiteren **MessageBox** den aktuellen Zustand dieses Elements aus, den wir über die Eigenschaft **State** ermitteln:

```
Private Sub txtFirma_LostFocus(sender As Object, e As RoutedEventArgs)
    Dim GeaendertesElement As DbEntityEntry
    GeaendertesElement = dbContext.ChangeTracker.Entries().Where(Function(x) x.Entity Is Kunde).First()
    MessageBox.Show("Element geändert? " + GeaendertesElement.State.ToString())
End Sub
```

Das Ergebnis sieht wie in Bild 4 aus. Der Zustand **Unchanged** entspricht nun nicht dem, was wir erwartet haben.

Erst, wenn wir zum geänderten Datensatz zurückwechseln und dann erneut das Ereignis **LostFocus** auslösen, erscheint der Wert **Modified** im Meldungsfenster (siehe Bild 5).

Offensichtlich ist der neue Wert im zugrundeliegenden Element **Kunde** beim Auslösen des **LostFocus**-Ereignisses noch nicht angekommen.

Also ist **LostFocus** wohl nicht unser Ereignis. Dementsprechend kommen wir auch mit einem Ereignis wie **TextChanged** nicht weiter, denn möglicherweise benötigen wir einen abgeschlossenen Fokuswechsel, um die Änderung in das **Kunde**-Element zu übertragen.

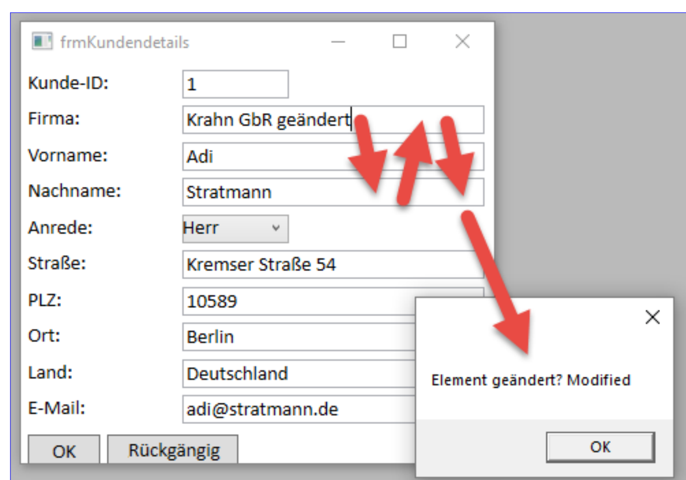


Bild 5: Aktueller Zustand des Elements, diesmal geändert

Änderung nach Fokuserhalt

Schauen wir uns also an, ob unsere Idee stimmt. Dazu fügen wir testweise dem nächsten Textfeld, hier **txtVorname**, das Attribut **GotFocus** hinzu:

```
<TextBox x:Name="txtVorname" Text="{Binding Kunde.Vorname}" Height="21" Margin="118,58,0,0" Width="221"
GotFocus="txtVorname_GotFocus"/>
```

Die Ereignismethode füllen wir wie folgt:

```
Private Sub txtVorname_GotFocus(sender As Object, e As RoutedEventArgs)
    Dim GeaendertesElement As DbEntityEntry
    GeaendertesElement = dbContext.ChangeTracker.Entries().Where(Function(x) x.Entity Is Kunde).First()
    MessageBox.Show("Element geändert? " + GeaendertesElement.State.ToString())
End Sub
```

Wenn wir nun den Inhalt von **txtFirma** ändern und den Fokus auf **txtVorname** verschieben, erhalten wir die erwartete Meldung – das Element hat den Zustand **Modified** (siehe Bild 6). Damit brauchen wir also für jedes Steuerelement, das den Fokus erhalten kann, das Ereignis **GotFocus** und eine entsprechende Ereignismethode.

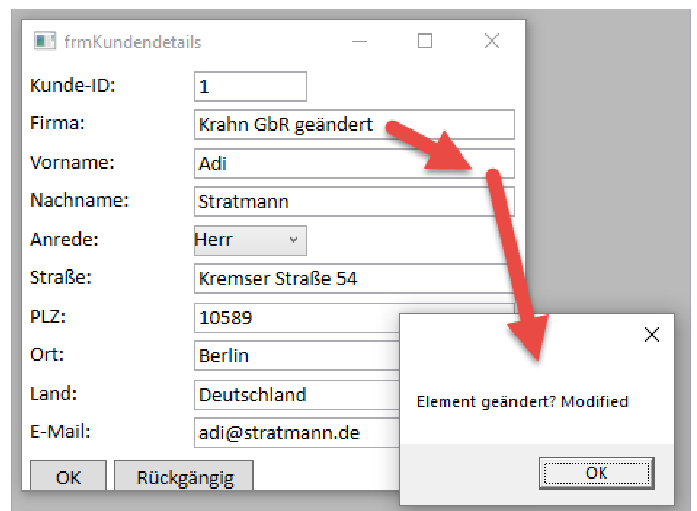


Bild 6: Beim Fokuserhalt ist das Element bereits geändert.

Das Zuweisen des Attributs zu allen Steuerelementen ist aufwendig – vor allem, je mehr Steuerelemente das Fenster enthält. Da das Ereignis immer die gleichen Anweisungen ausführen soll, reicht es allerdings aus, wenn wir nur ein Ereignis etwa namens **GotFocus** erstellen, dass wir dann für das Attribut **GotFocus** der verschiedenen Steuerelemente einrichten:

```
<TextBox x:Name="txtFirma" Text="{Binding Kunde.Firma}" Height="21" Margin="118,31,0,0" Width="221"
GotFocus="FrameworkElement_GotFocus"/>
<TextBox x:Name="txtVorname" Text="{Binding Kunde.Vorname}" Height="21" Margin="118,58,0,0" Width="221"
GotFocus="GotFocus"/>
...
```

Die Ereignismethode sähe dann so aus:

```
Private Sub FrameworkElement_GotFocus(sender As Object, e As RoutedEventArgs)
    ...
End Sub
```

Detailformulare mit Combo, Checkbox und Button

Im Artikel »Access zu WPF: Detailformulare mit Textfeldern« schauen wir uns an, wie die programmgesteuerten Möglichkeiten aussehen, um Formulare automatisch als WPF-Fenster oder -Seiten abzubilden. Damit haben wir einfache Detailformulare samt Textfeldern und Datenbindung unter WPF abgebildet. Nun wollen wir einen Schritt weitergehen und uns um weitere Steuerelemente wie etwa Kombinationsfelder und Kontrollkästchen kümmern. Außerdem wollen wir noch Schaltflächen zum Blättern in den Datensätzen sowie zum Anlegen neuer Datensätze hinzufügen.

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. Die weiteren Vorarbeiten werden im Artikel **Access zu WPF: Detailformulare mit Textfeldern** erläutert.

Kombinationsfelder hinzufügen

Wenn wir etwa das Kombinationsfeld **cboAnredeID** im WPF-Fenster abbilden möchten, benötigen wir einige weitere Elemente. Das erste ist natürlich das **ComboBox**-Element im XAML-Code. Diesem müssen wir die Eigenschaften für die Bindung übergeben. Außerdem brauchen wir im Code behind-Modul der WPF-Seite zusätzlichen Code, der die Auflistung der Anreden als öffentliche Eigenschaft bereithält, damit wir vom XAML-Code aus auf die Anreden zugreifen können. Wir schauen uns als Erstes die notwendigen Änderungen an. Im XAML-Code fügen wir allgemeine Eigenschaften für den Steuerelementtyp **ComboBox** hinzu:

```
<Window x:Class="frmKundendetails" ... Title="frmKundendetails" Height="332" Width="362">
    <Window.Resources>
        ...
        <Style TargetType="{x:Type ComboBox}">
            <Setter Property="HorizontalAlignment" Value="Left"></Setter>
            <Setter Property="VerticalAlignment" Value="Top"></Setter>
            <Setter Property="FontFamily" Value="Calibri"></Setter>
            <Setter Property="FontSize" Value="11pt"></Setter>
        </Style>
    </Window.Resources>
```

Für das Kombinationsfeld selbst fügen wir das **ComboBox**-Element hinzu. Im Beispielformular, das die Daten eines Kunden anzeigen soll, wollen wir mit der folgenden Definition eines **ComboBox**-Elements die Daten der Tabelle **Anreden** zur Auswahl einfügen und die für den aktuellen Kunden ausgewählte Anrede anzeigen:

```
    <Grid>
        ...
        <Label x:Name="Bezeichnungsfeld4" Content="Anrede:" Padding="0" Height="21" Margin="5,111,0,0" Width="64"/>
        <ComboBox x:Name="MehrwertsteuersatzID" Padding="0" Height="21" Margin="118,111,0,0" Width="221"
                  ItemsSource="{Binding Anreden}" SelectedItem="{Binding Kunde.Anrede, ValidatesOnDataErrors=True}"
                  SelectedValuePath="ID" SelectionChanged="Anrede_SelectionChanged">
            <ComboBox.ItemTemplate>
                <DataTemplate>
                    <TextBlock>
                        <TextBlock.Text>
                            <MultiBinding StringFormat="{}{0}">
                                <Binding Path="Name" />
                            </MultiBinding>
                        </TextBlock.Text>
                    </TextBlock>
                </DataTemplate>
            </ComboBox.ItemTemplate>
        </ComboBox>
        ....
    </Grid>
</Window>
```

In der Code behind-Klasse fügen wir zunächst im allgemeinen Teil die private Variable für die Liste der Anreden hinzu sowie die öffentliche Eigenschaft, über welche diese dann verfügbar ist:

```
Private _Anreden As List(Of Anrede)

Public Property Anreden As List(Of Anrede)
    Get
        Return _Anreden
    End Get
    Set(value As List(Of Anrede))
        _Anreden = value
    End Set
End Property
```

Schließlich benötigen wir in den Konstruktor-Methoden jeweils eine Anweisung, welche die Auflistung der Anreden mit den Daten der **DbSet**-Auflistung **Anreden** füllt:

```
Public Sub New()
    ...
```

```
    Anreden = New List(Of Anrede)(dbContext.Anreden)
    DataContext = Me
End Sub

Public Sub New(lngID As Long)
    ...
    Anreden = New List(Of Anrede)(dbContext.Anreden)
    DataContext = Me
End Sub
```

Standardeigenschaften für ComboBox-Element hinzufügen

Das wären zunächst einmal die Anforderungen für die einzufügenden Elemente, die wir in den Prozeduren des Moduls **mdlAccessZuWPF** berücksichtigen müssen. Die Erweiterungen sind teilweise trivial wie etwa die für die Standardeigenschaften für die Kombinationsfelder beziehungsweise **ComboBox**-Elemente. Diese fügen wir der Prozedur **AttributeHinzufuegen** wie folgt hinzu:

```
Public Sub AttributeHinzufuegen(frm As Form, strXAML As String)
    strXAML = strXAML & "        <Window.Resources>" & vbCrLf
    ...
    With frm.DefaultControl(acComboBox)
        strXAML = strXAML & "        <Style TargetType=""{x:Type ComboBox}"">" & vbCrLf
        strXAML = strXAML & "            <Setter Property=""HorizontalAlignment"" Value=""Left""></Setter>" & vbCrLf
        strXAML = strXAML & "            <Setter Property=""VerticalAlignment"" Value=""Top""></Setter>" & vbCrLf
        strXAML = strXAML & "            <Setter Property=""FontFamily"" Value=""" & .FontName & """></Setter>" & vbCrLf
        strXAML = strXAML & "            <Setter Property=""FontSize"" Value=""" & .FontSize & "pt""></Setter>" & vbCrLf
        strXAML = strXAML & "        </Style>" & vbCrLf
    End With
    strXAML = strXAML & "        </Window.Resources>" & vbCrLf
End Sub
```

ComboBox-Element hinzufügen

Der Teil, der die XAML-Definition des **ComboBox**-Elements zusammenstellt, ist etwas umfangreicher als der für das Zusammenstellen eines **Label**- oder eines **TextBox**-Elements. Das liegt daran, dass wir ein paar mehr Informationen sammeln müssen – etwa, um herauszufinden, welche Tabelle beziehungsweise welches **DbSet**-Element die Daten für das aufgeklappte **ComboBox**-Element liefert. Genau genommen ist es sogar recht kompliziert – und umso komplizierter, je mehr mögliche Fälle man abdecken möchte.

Gehen wir einmal davon aus, dass wir etwa das Kombinationsfeld zur Auswahl der Anrede eines Kunden abbilden wollen. Dann hat dieses Kombinationsfeld beispielsweise die folgende Abfrage als Datensatzherkunft:

```
SELECT [tblAnreden].[AnredeID], [tblAnreden].[Anrede] FROM tblAnreden;
```

Daneben verwendet es die erste Spalte dieser Abfrage als gebundene Spalte und die zweite soll angezeigt werden. Schließlich ist das Kombinationsfeld an das Feld **AnredeID** der Tabelle **tblKunden** gebunden. Diese Informationen benötigen wir auch für das **ComboBox**-Element – genau genommen sogar einige davon abgeleitete Informationen. Wir wollen ja nicht den Namen der Tabelle **tblAnreden** für das Attribut **ItemsSource** angeben, sondern benötigen das **DbSet** des Entity Data Models, dass die Daten dieser Tabelle liefert.

Generiertes Entity Data Model nutzen

In den Artikeln **Von Access zu Entity Framework: Datenmodell** und **Von Access zu Entity Framework: Update 1** haben wir gezeigt, wie Sie aus dem Datenmodell einer Access-Datenbank ein Entity Data Model erstellen können. Dort haben wir unter anderem eine Collection zusammengestellt, welche die resultierenden Mappings enthält – also etwa die Namen der Tabellen (zum Beispiel **tblKunden**), der resultierenden Klasse (**Kunde**), des **DbSet**-Elements (**Kunden**) und weiterer Informationen.

Die Daten aus dieser Collection können wir auch noch nutzen, um die Informationen zur Definition der **ComboBox**-Elemente zusammenzustellen. In der Collection, die Elemente des Typs **clsMapping** aufnimmt, fehlen allerdings noch zwei Informationen, die wir noch benötigen: Wenn der Name eines Feldes mit dem resultierenden Namen der Klasse übereinstimmt, was etwa bei der Tabelle **tblAnreden** mit den Feldern **AnredeID** und **Anrede** auftritt, dann soll der Name dieses Feldes als Eigenschaft der Klasse in **Name** umbenannt werden. Da wir zufällig genau beim Beispiel der Tabelle **tblKunden**, welche ja im Fremdschlüsselfeld **AnredeID** auf die Datensätze der Tabelle **tblAnreden** zugreift, auf diesen Fall treffen, wissen wir auch, dass diese Informationen gut noch in den Elementen der Collection **colMappings** untergebracht werden könnten.

Dazu erweitern wir zunächst die Klasse **clsMapping** in der Access-Datenbank, welche die zu migrierenden Formulare enthält, um die folgenden beiden privaten Variablen und die entsprechenden Getter und Setter:

```
Private m_RenamedField As String
Private m_RenamedFieldOriginal As String

Public Property Get RenamedField() As String
    RenamedField = m_RenamedField
End Property

Public Property Let RenamedField(str As String)
    m_RenamedField = str
End Property

Public Property Get RenamedFieldOriginal() As String
    RenamedFieldOriginal = m_RenamedFieldOriginal
End Property

Public Property Let RenamedFieldOriginal(str As String)
    m_RenamedFieldOriginal = str
End Property
```

Die Eigenschaft **RenamedFieldOriginal** soll dann etwa den Namen des Feldes **Anrede** aufnehmen und **RenamedField** den Namen der resultierenden Eigenschaft der Klasse **Anrede**, also **Name**. Die Prozedur **EDMErstellen** aus dem Modul **mdlEDM** wandeln wir in eine gleichnamige Funktion um, welche eine Collection als Funktionswert zurückliefern soll:

```
Public Function EDMErstellen() As Collection
    ...
    Set colMappings = New Collection
    ...
```

In der **For Each**-Schleife über alle Mappings, in denen die Funktion die Felder untersucht, schreibt die Funktion dann die entsprechenden Werte in die beiden Eigenschaften **RenamedField** und **RenamedFieldOriginal** der **clsMapping**-Klasse. Das sieht dann wie folgt aus:

```
    For Each objMapping In colMappings
        With objMapping
            ...
            For Each fld In tdf.Fields
                ...
                If fld.Name = .Entity_Original Then
                    strFieldname = "Name"
                    objMapping.RenamedField = "Name"
                    objMapping.RenamedFieldOriginal = fld.Name
                Else
                    ...
                End If
                ...
            Next fld
            ...
        End With
        ...
    Next objMapping
    ...
```

Schließlich wird die Collection **colMappings** als Funktionswert der Funktion **EDMErstellen** zurückgegeben:

```
    Set EDMErstellen = colMappings
End Function
```

Anpassung der Prozeduren zum Migrieren des Formulars

Die Funktion **EDMErstellen** rufen wir nun nicht mehr nur auf, wenn wir das Entity Data Model auf Basis des Datenmodells der Datenbank erstellen wollen, sondern auch beim Migrieren eines der Formulare mit der Prozedur **FormularNachWPF** des

Moduls **mdlAccessZuWPF** der Access-Datenbank. Dies erledigen wir direkt in der ersten Anweisung nach dem Deklarationsteil der Prozedur:

```
Public Sub FormularNachWPF(strForm As String, strKlasse As String)
    ...
    Dim colMappings As Collection
    Set colMappings = EDMErstellen
    ...
```

Dann übergeben wir diese Collection beim Aufruf der Prozedur **SteuerelementeHinzufuegen** als neuen Parameter:

```
    SteuerelementeHinzufuegen frm, strXAML, strKlasse, colMappings
    ...
End Sub
```

Die Prozedur **SteuerelementeHinzufuegen** erfährt die umfassendsten Anpassungen. Zunächst fügen wir **colMappings** mit dem Datentyp **Collection** als vierten Parameter zur Prozedur hinzu:

```
Public Function SteuerelementeHinzufuegen(frm As Form, strXAML As String, strKlasse As String, colMappings As Collection)
    ...
```

Dann benötigen wir die folgenden zusätzlichen Variablen. Die erste namens **strEntitiesCombo** nimmt den Namen der **DbSet**-Auflistung für die Elemente des **ComboBox**-Elements auf (etwa **Anreden**), das zweite namens **strEntityCombo** den Namen der Klasse für die Auflistung. **strPKCombo** erhält das Primärschlüsselfeld der entsprechenden Entität, also zum Beispiel **ID**.

strSichtbaresFeldCombo schließlich füllen wir mit dem Namen des zweiten Feldes der Datensatzherkunft des Kombinationsfeldes, im Falle der Datensatzherkunft **SELECT [tblAnreden].[AnredeID], [tblAnreden].[Anrede] FROM tblAnreden;** also etwa das Feld **Anrede**.

Diesen Wert ändern wir aber in diesem Fall noch, da **Anrede** ja dem Namen der entstehenden Entität entspricht und dementsprechend umbenannt werden muss.

```
    Dim strEntitiesCombo As String
    Dim strEntityCombo As String
    Dim strPKCombo As String
    Dim strSichtbaresFeldCombo As String
    ...
```

In der Schleife über alle Steuerelemente des Formulars landen wir dann früher oder später bei einem Kombinationsfeld. Für dieses haben wir in einer **Select Case**-Bedingung einen eigenen Zweig eingerichtet. In diesem fügen wir zunächst die Schriftart und Schriftgröße ein, sollte sich diese von der Standardschriftart für dieses Steuerelement unterscheiden:

```
For Each ctl In frm.Controls
    ...
    Select Case ctl.ControlType
        ...
        Case acComboBox
            If Not ctl.FontName = strFontFamilyTextBox Then
                 strFontname = "FontFamily=""" & strFontFamilyTextBox & """ "
            End If
            If Not ctl.FontSize = strFontSizeTextBox Then
                strFontSize = "FontSize=""" & strFontSizeTextBox & "pt"" "
            End If
```

Danach füllen wir die Variable **strFeldname** mit dem Wert des Feldes **Steuerelementinhalt** des Kombinationsfeldes:

```
            strFeldname = ctl.ControlSource
```

Die Variable **strSichtbaresFeldCombo** nimmt das zweite Feld der als Datensatzherkunft angegebenen Tabelle oder Abfrage als Wert:

```
            strSichtbaresFeldCombo = CurrentDb.OpenRecordset(ctl.RowSource).Fields(1).Name
```

In einer **If...Then**-Bedingung verwenden wir dann das Ergebnis eines Aufrufs der Funktion **GetDbSetCombo** als Kriterium. Liefert diese Funktion, welche aus den **Mapping**-Informationen aus **colMappings** und der Datensatzherkunft des Steuerelements den Namen der Entitätenliste, des Primärschlüsselfeldes und des sichtbaren Feldes ermitteln soll, den Wert **True**, wurden diese Werte erfolgreich gefüllt und wir können den Code für das **ComboBox**-Element zusammenstellen:

```
            If GetDbSetCombo(colMappings, ctl.RowSource, strEntitiesCombo, strPKCombo, strSichtbaresFeldCombo, _
                strEntityCombo) Then
```

Ist das der Fall, tragen wir zunächst den Namen des Steuerelements für das Attribut **Name** des **ComboBox**-Elements ein sowie die Höhe, die Breite und die Abstände zum linken und oberen Rand:

```
                strXAML = strXAML & "<ComboBox x:Name=""" & ctl.Name & """ Padding=""0"" Height=""" & lngHeight _
                    & """ Margin=""" & lngLeft & "," & lngTop & ",0,0"" Width=""" & lngWidth & """" & vbCrLf
```

Für das Attribut **ItemsSource** stellen wir eine Bindung zu dem Element aus **strEntitiesCombo** her, in diesem Fall **Anreden**:

```
                strXAML = strXAML & "            ItemsSource = ""{Binding " & strEntitiesCombo & "}""" & vbCrLf
```

Den aktuell ausgewählten Eintrag wollen wir über das Feld **Name** der Klasse **Anrede** ermitteln und tragen diesen für das Attribut **SelectedItem** ein:

```
                strXAML = strXAML & "            SelectedItem = ""{Binding " & strKlasse & "." & strEntityCombo _
                    & ", ValidatesOnDataErrors=True}""" & vbCrLf
```

Zu den allgemeinen Attributen gehört auch noch **SelectedValuePath**, das festlegt, welche Eigenschaft der gebundenen Spalte entspricht, also wonach der Eintrag der **ComboBox** für den aktuellen Datensatz im Fenster ausgewählt wird:

```
                strXAML = strXAML & "            SelectedValuePath=""" & strPKCombo & """> " & vbCrLf
```

Danach folgen die anzuzeigenden Werte des **ComboBox**-Elements. Dazu legen wir das folgende Konstrukt an, mit dem Sie direkt ein **MultiBinding**-Element haben, das Sie nach der Migration leicht um weitere Felder erweitern können. Wir fügen hier für das Element **Binding** die Eigenschaft auf **strSichtbaresFeldCombo** ein, was im Beispiel der Bezeichnung **Name** entspricht:

```
                strXAML = strXAML & "    <ComboBox.ItemTemplate>" & vbCrLf
                strXAML = strXAML & "        <DataTemplate>" & vbCrLf
                strXAML = strXAML & "            <TextBlock>" & vbCrLf
                strXAML = strXAML & "                <TextBlock.Text>" & vbCrLf
                strXAML = strXAML & "                    <MultiBinding StringFormat=""{}{0}"">" & vbCrLf
                strXAML = strXAML & "                        <Binding Path=""" & strSichtbaresFeldCombo & """ />" & vbCrLf
                strXAML = strXAML & "                    </MultiBinding>" & vbCrLf
                strXAML = strXAML & "                </TextBlock.Text>" & vbCrLf
                strXAML = strXAML & "            </TextBlock>" & vbCrLf
                strXAML = strXAML & "        </DataTemplate>" & vbCrLf
                strXAML = strXAML & "    </ComboBox.ItemTemplate>" & vbCrLf
                strXAML = strXAML & "</ComboBox>" & vbCrLf
```

Wenn **GetDbSetCombo** den Wert **False** liefert, konnte kein zur Datensatzherkunft des Kombinationsfeldes passendes Mapping gefunden werden. Das ist etwa der Fall, wenn als Datensatzherkunft eine Abfrage verwendet wurde. Dann erscheint eine entsprechende Meldung:

```
            Else
                MsgBox "ComboBox für das Steuerelement '" & ctl.Name & "' konnte nicht erstellt werden, da die " _
                    & "RowSource keine passende Tabelle enthielt."
            End If
        ...
        End Select
    Next ctl
End Function
```

Nun schauen wir uns noch die Funktion **GetDbSetCombo** an. Dieser übergeben wir die Collection mit den Mapping-Informationen in den verschiedenen **clsMapping**-Klassen sowie die Datensatzherkunft des Kombinationsfelds als Parameter. Die übrigen Parameter sollen in der Funktion gefüllt beziehungsweise angepasst werden:

```
Public Function GetDbSetCombo(colMappings As Collection, strRowSource As String, strEntitiesCombo As String, _
        strPKCombo As String, strSichtbaresFeldCombo As String, strEntityCombo As String) As Boolean
```

Dabei durchlaufen wir alle **clsMapping**-Elemente der Collection **colMappings** in einer **For Each**-Schleife. Darin prüfen wir, ob der in der **clsMapping**-Klasse in der Eigenschaft **Tabelle** enthaltene Tabellenname in der Zeichenkette der Datensatzherkunft aus **strRowSource** vorkommt. So kommt dann irgendwann **tblAnreden** in **SELECT AnredeID, Anrede FROM tblAnreden** vor. Ist das der Fall, schreiben wir den Inhalt der Eigenschaft **Entities** (hier **Anreden**) in den Parameter **strEntitiesCombo** und den Namen des **ID**-Feldes aus der Eigenschaft **ID** in die Variable **strPKCombo**.

Außerdem prüfen wir, ob sich der in der aufrufenden Prozedur ermittelte Name des sichtbaren Feldes, hier **Anrede**, in der Eigenschaft **RenamedFieldOriginal** findet. Falls nicht – und hier heißt dieses Name – wird der Inhalt von **RenamedFieldOriginal** in **strSichtbaresFeldCombo** übernommen und zurückgegeben:

```
    Dim objMapping As clsMapping
    For Each objMapping In colMappings
        If Not InStr(1, strRowSource, objMapping.Tabelle) = 0 Then
            strEntitiesCombo = objMapping.Entities
            strPKCombo = objMapping.ID
            If objMapping.RenamedFieldOriginal = strSichtbaresFeldCombo Then
                strSichtbaresFeldCombo = objMapping.RenamedField
            End If
            strEntityCombo = objMapping.Entity
            GetDbSetCombo = True
            Exit Function
        End If
    Next objMapping
End Function
```

Daten für ComboBox-Elemente in Code behind-Modul bereitstellen

Nun fehlen noch ein paar Elemente im Code behind-Modul. Dazu gehört etwa eine private und eine öffentliche Eigenschaft für die Daten, in diesem Beispiel die Anreden. Die dazu benötigten Elemente haben wir bereits eingangs benannt. Wir müssen nun nur die Prozedur **FormularNachWPF_CodeBehind**, welche bisher den Code für die Code behind-Klasse zusammenstellt, so ergänzen, dass diese direkt untersucht, ob das Formular Kombinationsfelder enthält und diese beim Zusammenstellen des Codes berücksichtigen. Die Erweiterungen sehen wie folgt aus, wobei wir zunächst eine Reihe neuer Variablen deklarieren:

```
Public Sub FormularNachWPF_CodeBehind(strFormular As String, strKlasse As String, strCollection As String, _
        strContext As String)
    Dim strCodeBehind As String
    Dim frm As Form
    Dim ctl As Control
    Dim strPrivateLists As String
```

```
Dim strPublicLists As String
Dim strEntitiesCombo As String
Dim strPKCombo As String
Dim strSichtbaresFeldCombo As String
Dim strEntityCombo As String
Dim strKonstruktorCombo As String
Dim colMappings As Collection
```

frm und **ctl** dienen dazu, das Formular zu referenzieren und die enthaltenen Steuerelemente zu durchlaufen, um herauszufinden, ob dieses gebundene Kombinationsfelder enthält. **strPrivateLists** und **strPublicLists** nehmen die Deklaration der privaten Variablen und die öffentlichen Eigenschaften der Listen auf, die in dem jeweiligen **ComboBox**-Element angezeigt werden sollen. Hierfür und auch für weitere Elemente benötigen wir wieder die Informationen über die Elemente des Entity Data Models, die wir über die Collection **colMappings** beziehen. Daher deklarieren wir diese auch in dieser Prozedur und füllen sie über die Funktion **EDMErstellen**. Später könnte man das Erstellen von Entity Data Model, den XAML-Dateien und den Code behind-Dateien auch in einem Zuge erledigen und EDMErstellen nur einmal aufrufen:

```
Set colMappings = EDMErstellen
```

Dann öffnen wir das Formular nochmals im Entwurf und referenzieren es mit der Variablen **frm**. Dann durchlaufen wir in einer **For Each**-Schleife alle Elemente der **Controls**-Auflistung des Formulars:

```
DoCmd.OpenForm strFormular, acDesign
Set frm = Forms(strFormular)
For Each ctl In frm.Controls
```

Hier prüfen wir, ob es sich beim aktuellen Steuerelement um ein Kombinationsfeld handelt und ob es ein gebundenes Steuerelement ist. Falls ja, ermitteln wir das sichtbare Feld wie bereits weiter oben beschrieben sowie die übrigen Informationen aus dem Entity Data Model für die Inhalte des **ComboBox**-Elements, hier für die Anreden:

```
    If ctl.ControlType = acComboBox Then
        If Not Len(ctl.ControlSource) = 0 Then
            strSichtbaresFeldCombo = CurrentDb.OpenRecordset(ctl.RowSource).Fields(1).Name
            If GetDbSetCombo(colMappings, ctl.RowSource, strEntitiesCombo, strPKCombo, strSichtbaresFeldCombo, _
                    strEntityCombo) Then
```

Wurden passende Mapping-Informationen für die Datensatzherkunft des Kombinationsfeldes gefunden, fügen wir in den Variablen **strPrivateLists**, **strPublicLists** und **strKonstruktorCombo** die Informationen zusammen, die wir später zu den bereits vorbereitenden Inhalten der Code behind-Klasse hinzufügen wollen. Dabei greifen wir auf die aus der Collection **colMappings** gewonnenen Informationen zu:

```
                strPrivateLists = strPrivateLists & "    Private _" & strEntitiesCombo & " As List(Of " _
```

Access zu WPF: Validierung und Navigation

Im Artikel »Access zu WPF: Detailformulare mit Combo, Checkbox und Button« haben wir gezeigt, wie Sie die in Formularen enthaltenen gebundenen Steuerelemente wie Kombinationsfelder und Kontrollkästchen sowie Schaltflächen in WPF-Fenster übertragen. In diesem Artikel wollen wir uns ansehen, wie Sie einem WPF-Fenster auf Basis eines Access-Detailformulars eine Validierung und Navigationsschaltflächen hinzufügen. Die notwendigen Techniken haben wir bereits im Artikel »EDM: Blättern in Datensätzen« vorgestellt, nun integrieren wir diese in unsere VBA-Prozedur zum Erstellen des Codes für ein WPF-Fenster mit Navigationsschaltflächen.

Um festzulegen, ob ein Formular mit oder ohne Validierung und Navigationsschaltflächen erstellt werden soll, fügen wir der Prozedur **FormularNachWPF** zwei weitere Parameter hinzu:

- **bolValidation**: Gibt an, ob die Validierung hinzugefügt werden soll.
- **bolNavigation**: Gibt an, ob die Navigationsschaltflächen hinzugefügt werden sollen.

Die Validierung für einen neuen, leeren Datensatz sieht dann etwa wie in Bild 1 aus.

Bild 1: Validierung bei einem neuen, leeren Datensatz

Die Prozedur **FormularNachWPF** erweitern wir an den folgenden Stellen um die Entgegennahme beziehungsweise Übergabe der Parameter:

```
Public Sub FormularNachWPF(strForm As String, strTabelle As String, Optional bolValidation As Boolean, _
        Optional bolNavigation As Boolean)
    ...
    lngHeight = TwipsToPixelsHeight(frm.Section(0).Height) + 30
    If bolNavigation = True Then
        lngTopNavi = lngHeight - 30
        lngHeight = lngHeight + 28
    End If
```

Bei der Definition des **Window**-Elements fügen wir noch das Ereignisattribut **Closing** mit dem Wert **Window_Closing** hinzu, welches sicherstellt, dass das Fenster nur bei geändertem und gespeichertem Datensatz geschlossen werden kann:

```
    strXAML = strXAML & "<Window x:Class=""" & frm.Name & """" & vbCrLf
```

```
    ...
    strXAML = strXAML & "        Title=""" & strTitle & """ Height=""" & lngHeight & """ Width=""" & lngWidth _
        & """ Closing=""Window_Closing"">" & vbCrLf
```

Für die folgenden Aufruf haben wir die Übergabe der Parameter **bolValidation** und **bolNavigation** erweitert:

```
    ...
    AttributeHinzufuegen frm, strXAML, bolValidation, bolNavigation
    ...
    SteuerelementeHinzufuegen frm, strXAML, strTabelle, colMappings, bolValidation, bolNavigation
    If bolNavigation Then
        NavigationHinzufuegen strXAML, lngTopNavi
    End If
    ...
End Sub
```

Erweiterung der Prozedur AttributeHinzufuegen

Die XAML-Datei enthält mit eingebauter Validierung einige Zeilen Code mehr. Diesem werden wir in der Prozedur **AttributeHinzufuegen** wie folgt gerecht, die ebenfalls die beiden Parameter **bolValidation** und **bolNavigation** erwartet:

```
Public Sub AttributeHinzufuegen(frm As Form, strXAML As String, bolValidation As Boolean, bolNavigation As Boolean)
```

In dieser Prozedur geht es allein um das Zusammenstellen allgemeiner Properties für Steuerelemente im **WindowResources**-Bereich. Da wir hier auch die Steuerelemente durchlaufen, um die **DataTrigger** und **MultiDataTrigger** für die Validierung zu definieren, benötigen wir eine Variable namens **ctl** mit dem Typ **Control**:

```
    Dim ctl As Control
```

Danach geht es dann direkt rund. Wir prüfen, ob **bolValidation** den Wert **True** aufweist und fügen dann die Vorlage für die Markierung nicht erfolgreich validierter Steuerelemente hinzu:

```
    strXAML = strXAML & "        <Window.Resources>" & vbCrLf
    If bolValidation Then
        strXAML = strXAML & "            <Style TargetType=""{x:Type FrameworkElement}"">" & vbCrLf
        strXAML = strXAML & "                <Setter Property=""Validation.ErrorTemplate"">" & vbCrLf
        strXAML = strXAML & "                    <Setter.Value>" & vbCrLf
        strXAML = strXAML & "                        <ControlTemplate>" & vbCrLf
        strXAML = strXAML & "                            <DockPanel>" & vbCrLf
        strXAML = strXAML & "                                <Border Background=""Red"" DockPanel.Dock=""right"" " _
            & "Margin=""5,0,0,0"" Width=""20"" Height=""20"" CornerRadius=""10""" & vbCrLf
        strXAML = strXAML & "                                    ToolTip=""{Binding ElementName=customAdorner, " _
```

```
        & "Path=AdornedElement.(Validation.Errors)[0].ErrorContent}"">" & vbCrLf
    strXAML = strXAML & "                                <TextBlock Text=""!"" VerticalAlignment=""center"" " _
        & "HorizontalAlignment=""center""" & vbCrLf
    strXAML = strXAML & "                                   FontWeight=""Bold"" Foreground=""white"" />" & vbCrLf
    strXAML = strXAML & "                            </Border>" & vbCrLf
    strXAML = strXAML & "                            <AdornedElementPlaceholder Name=""customAdorner"" " _
        & "VerticalAlignment=""Center"" >" & vbCrLf
    strXAML = strXAML & "                                <Border BorderBrush=""red"" BorderThickness=""1"" />" & vbCrLf
    strXAML = strXAML & "                            </AdornedElementPlaceholder>" & vbCrLf
    strXAML = strXAML & "                        </DockPanel>" & vbCrLf
    strXAML = strXAML & "                    </ControlTemplate>" & vbCrLf
    strXAML = strXAML & "                </Setter.Value>" & vbCrLf
    strXAML = strXAML & "            </Setter>" & vbCrLf
    strXAML = strXAML & "        </Style>" & vbCrLf
End If
...
```

Der nächste Teil, der nur von der Notwendigkeit einer Validierung abhängt, sind die Elemente **ComboBox** und **TextBox**, die wir beide mit den **Style**-Attributen für die Validierung ausstatten müssen – also mit dem soeben angelegten Style. Dazu prüfen wir wieder per **If...Then**-Bedingung, ob **bolValidation** den Wert **True** aufweist, und fügen dann eine der beiden Zeilen als erste Zeile des Styles hinzu:

```
...
With frm.DefaultControl(acTextBox)
    If bolValidation Then
        strXAML = strXAML & "        <Style TargetType=""{x:Type TextBox}"" "_
            & "BasedOn=""{StaticResource {x:Type FrameworkElement}}"">" & vbCrLf
    Else
        strXAML = strXAML & "        <Style TargetType=""{x:Type TextBox}"">" & vbCrLf
    End If
    ...
End With
With frm.DefaultControl(acComboBox)
    If bolValidation Then
        strXAML = strXAML & "        <Style TargetType=""{x:Type ComboBox}"" " _
            & "BasedOn=""{StaticResource {x:Type FrameworkElement}}"">" & vbCrLf
    Else
        strXAML = strXAML & "        <Style TargetType=""{x:Type ComboBox}"">" & vbCrLf
    End If
    ...
End With
```

```
...
```

Mit dem **BasedOn**-Attribut sorgen wir dafür, dass der oben definierte, für den Typ **FrameworkElement** vorgesehene Style sowohl an **ComboBox**- als auch **TextBox**-Elemente vererbt wird. Durch das Erfassen der Attribute für die Markierung nicht erfolgreich validierter Steuerelemente in einem eigenen **Style**-Element sparen wir einige Zeilen Code, da wir die gleichen Elemente nicht für beide Steuerelementtypen einzeln hinterlegen müssen.

Erst wollten wir auch das **CheckBox**-Element auf diese Weise ausstatten, aber dieses ist ja ohnehin meist mit dem Wert **False** vorbelegt, sodass keine Validierung notwendig ist.

Danach stellen wir die **DataTrigger**-Elemente für drei Steuerelementtypen **ComboBox**, **TextBox** und **CheckBox** zusammen, die dafür sorgen, dass die Eigenschaft **IsEnabled**, die wir später im Code behind-Modul definieren, für die mit diesem **DataTrigger** ausgestatteten Style auf **False** eingestellt wird, wenn die Validierung auch nur für ein einziges der Steuerelemente fehlschlägt. Dies betrifft etwa die Schaltflächen zum Navigieren, die wir später hinzufügen. Hier verwenden wir zum ersten Mal die Variable **ctl**, mit der wir in einer **For Each**-Schleife alle Steuerelemente des Formulars durchlaufen. Damit fügen wir dem **Style**-Element, das wir über das Attribut **x:Key** mit dem Wert **EnableOnValidation** identifizieren, für jedes Steuerelement drei Zeilen hinzu, die das **DataTrigger**-Element mit der Bedingung und das **Setter**-Element mit dem zu setzenden Wert enthalten:

```
    ...
    If bolValidation Then
        strXAML = strXAML & "        <Style x:Key=""EnableOnValidation"" TargetType=""{x:Type Button}"">" & vbCrLf
        strXAML = strXAML & "            <Style.Triggers>" & vbCrLf
        For Each ctl In frm.Controls
            Select Case ctl.ControlType
                Case acComboBox, acCheckBox, acTextBox
                    If Not Len(ctl.ControlSource) = 0 Then
                        strXAML = strXAML & "                <DataTrigger Binding=""{Binding ElementName=" & ctl.Name _
                            & ", Path=(Validation.HasError)}"" Value=""True"">" & vbCrLf
                        strXAML = strXAML & "                    <Setter Property=""IsEnabled"" " _
                            & "Value=""False""></Setter>" & vbCrLf
                        strXAML = strXAML & "                </DataTrigger>" & vbCrLf
                    End If
            End Select
        Next ctl
        strXAML = strXAML & "            </Style.Triggers>" & vbCrLf
        strXAML = strXAML & "        </Style>" & vbCrLf
...
```

Außerdem benötigen wir noch jeweils ein **MultiDataTrigger**-Element für jedes gebundene Steuerelement der Typen **ComboBox**, **TextBox** und **CheckBox**. Dieses liefert standardmäßig für das Attribut **IsEnabled** den Wert **True**. Wenn alle in den **Condition**-Elementen genannten Bedingungen wahr sind, also die Validierung für alle Elemente keinen Fehler liefert, wird **IsEnabled**

auf **False** eingestellt. Dieser mit dem Schlüssel **EnableOnFailedValidation** benannte Style wird für die **Rückgängig**-Schaltfläche eingesetzt, um diese nur zu aktivieren, wenn mindestens eine Validierung fehlschlägt und diese nur zu deaktivieren, wenn alle Validierungen erfolgreich waren. Auch hier durchlaufen wir in einer **For Each**-Schleife alle gebundenen Steuerelemente:

```
...
        strXAML = strXAML & "        <Style x:Key=""EnableOnFailedValidation"" TargetType=""{x:Type Button}"">" & vbCrLf
        strXAML = strXAML & "            <Setter Property=""IsEnabled"" Value=""True"" />" & vbCrLf
        strXAML = strXAML & "            <Style.Triggers>" & vbCrLf
        strXAML = strXAML & "                <MultiDataTrigger>" & vbCrLf
        strXAML = strXAML & "                    <MultiDataTrigger.Conditions>" & vbCrLf
        For Each ctl In frm.Controls
            Select Case ctl.ControlType
                Case acComboBox, acCheckBox, acTextBox
                    If Not Len(ctl.ControlSource) = 0 Then
                        strXAML = strXAML & "                        <Condition Binding=""{Binding ElementName=" _
                            & ctl.Name & ", Path=(Validation.HasError)}"" Value=""False"" />" & vbCrLf
                    End If
            End Select
        Next ctl
        strXAML = strXAML & "                    </MultiDataTrigger.Conditions>" & vbCrLf
        strXAML = strXAML & "                    <Setter Property=""IsEnabled"" Value=""False"" />" & vbCrLf
        strXAML = strXAML & "                </MultiDataTrigger>" & vbCrLf
        strXAML = strXAML & "            </Style.Triggers>" & vbCrLf
        strXAML = strXAML & "        </Style>" & vbCrLf
    End If
```

Sollen die Navigationsschaltflächen angezeigt werden, brauchen wir auch noch diese beiden **Style**-Elemente, die dafür sorgen, dass die Schaltfläche **btnErster** und **btnVorheriger**, die wir nachfolgend hinzufügen, nur aktiviert werden, wenn gerade nicht der erste Datensatz angezeigt wird. Genauso sollen die Schaltflächen **btnNaechster** und **btnLetzter** nur aktiviert werden, wenn gerade nicht der letzte Datensatz angezeigt wird. Dazu weisen wir den Schaltflächen die folgenden **Style**-Elemente zu, die wiederum auf den **Style**-Elementen **EnableOnValidation** basieren:

```
    If bolNavigation = True Then
        strXAML = strXAML & "            <Style x:Key=""ButtonVorheriger"" BasedOn=""{StaticResource " _
            & "EnableOnValidation}"" TargetType=""{x:Type Button}"">" & vbCrLf
        strXAML = strXAML & "            <Style.Triggers>" & vbCrLf
        strXAML = strXAML & "                <DataTrigger Binding=""{Binding ErsterKunde}"" Value=""False"">" & vbCrLf
        strXAML = strXAML & "                    <Setter Property=""IsEnabled"" Value=""False""></Setter>" & vbCrLf
        strXAML = strXAML & "                </DataTrigger>" & vbCrLf
        strXAML = strXAML & "            </Style.Triggers>" & vbCrLf
        strXAML = strXAML & "        </Style>" & vbCrLf
```

Danach durchläuft sie alle Zeilen der Prozedur im VBA-Modul und fügt diese an die Variable **strCodeBehind** an – allerdings durch ein Kommentarzeichen (**'**) markiert und somit auskommentiert, damit keine Fehler im VB.NET-Modul angezeigt werden:

```
                For lngLine = lngCodeBody + 1 To lngCodeBody + lngCodeLaenge - 1
                    strLine = objCodeModule.Lines(lngLine, 1)
                    If Not Left(Trim(strLine), 3) = "End" And Not Len(Trim(strLine)) = 0 Then
                        strCodeBehind = strCodeBehind & "    '" & objCodeModule.Lines(lngLine, 1) & vbCrLf
                    End If
                Next
```

Schließlich fehlt noch die **End Sub**-Zeile:

```
                strCodeBehind = strCodeBehind & "    End Sub" & vbCrLf & vbCrLf
        End Select
    Next ctl
```

Nach dem Durchlaufen aller Schaltflächen werden die Prozeduren als Ergebnis der Funktion zurückgegeben:

```
    VBAProzeduren = strCodeBehind
End Function
```

Zusammenfassung und Ausblick

Mit dieser Version der Prozeduren zum Migrieren eines Access-Formulars in ein WPF-Fenster sind wir einen Schritt weitergekommen. Zur Erinnerung: Es geht um gebundene Fenster in der Detailansicht, die einen Datensatz zum Betrachten und Bearbeiten anzeigen. Mit der hier vorliegenden Version können Sie eine Validierung hinzufügen, die sich nach den im Access-Datenmodell festgelegten Restriktionen richtet.

Außerdem fügen Sie damit Navigationsschaltflächen hinzu, mit denen Sie zwischen den Datensätzen navigieren können. Damit erhalten Sie fast das gleiche Feeling wie im Access-Formular. Ein Unterschied ist, dass es eine Schaltfläche namens **Rückgängig** gibt, mit denen Sie im Falle fehlgeschlagener Validierungen die Änderungen rückgängig machen können, sodass bereits vorhandene Datensätze wieder in den vorherigen Zustand zurückversetzt werden und neue Datensätze komplett verworfen werden.