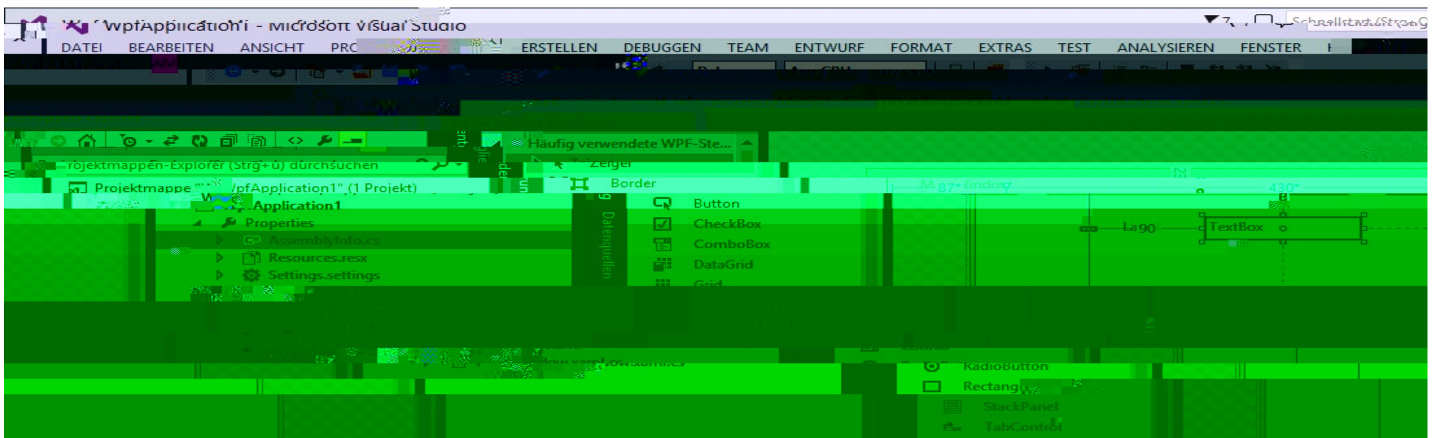
AUSGABE 05/2016

DATENBANK ENTWICKLER

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



TOP-THEMEN:

SQL SERVER	SQL Server: Datenbanken per Skript kopieren	SEITE 3
C#-GRUNDLAGEN	Die MessageBox-Klasse	SEITE 11
WPF-BASICS	Ribbons mit WPF	SEITE 21
WPF-BASICS	EDM: Einfaches Detailfenster	SEITE 34
WPF-BASICS	EDM: Lookup-Kombinationsfelder	SEITE 47

SQL SERVER UND CO.	SQL Server: Datenbanken per Skript kopieren	3
C#-GRUNDLAGEN	Die MessageBox-Klasse	11
	InputBox im Eigenbau	14
WPF-GRUNDLAGEN	Ribbons mit WPF	21
	EDM: Einfaches Detailfenster	34
	EDM: Lookup-Kombinationsfelder	47
	EDM: Kombinationsfelder erweitern	50
	EDM: 1:n-Beziehung als Parent-Child-Ansicht	57
TIPPS UND TRICKS	Tipps und Tricks	63
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
© 2016 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.

SQL Server: Datenbanken per Skript kopieren

Die Beispieldatenbanken auf Basis des SQL Servers sollen Sie schnell und unkompliziert auf Ihrem System nutzbar machen können. Leider geht das nicht immer so einfach – zum Beispiel, weil verschiedene Versionen des SQL Servers uns einen Strich durch die Rechnung machen. Also stellen wir in diesem Artikel eine einfache Lösung vor, wie Sie die Beispieldatenbanken nach dem Download schnell nutzen können – nämlich indem wir diese als Skript bereitstellen, das die komplette Definition der Datenbank sowie die enthaltenen Daten anlegt.

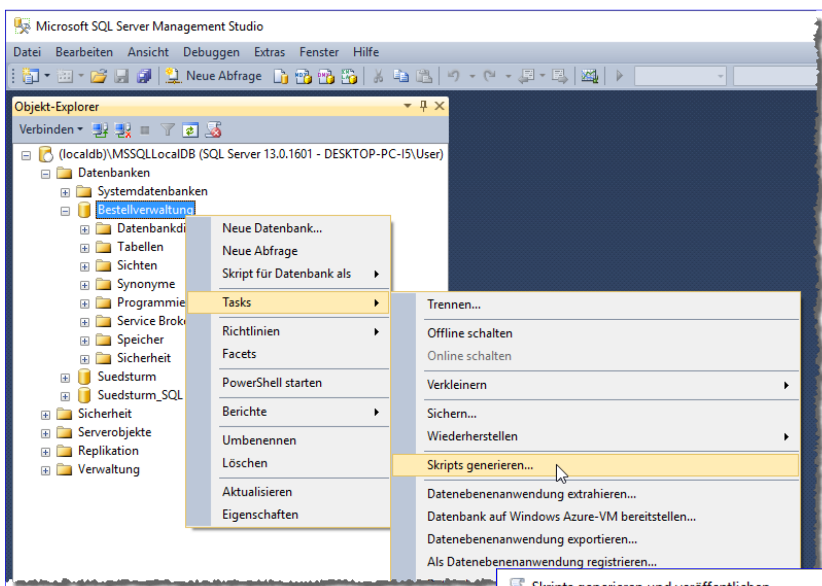


Bild 1: Hinzufügen der Verbindung

Für die Nutzung der Beispieldatenbanken verwenden Sie optimalerweise neben Ihrer SQL Server-Installation, bei der es sich auch um die einfache LocalDB-Variante handeln kann, noch das kostenlos verfügbar SQL Server Management Studio.

Wenn ich die Beispieldatenbank für den Download vorbereite, habe ich diese bereits im SQL Server Management Studio eingebunden – in diesem Beispiel in der Version 2014. Hier kann ich dann im Objekt-Explorer unter **Datenbanken** die gewünschte Datenbank auswählen (hier **Bestellverwaltung**), das Kontextmenü anzeigen und dort den Befehl **Tasks|Skripts generieren...** ausführen (siehe Bild 1).

Skript-Assistent starten

Dies öffnet den Dialog aus Bild 2. Hier können Sie entweder die Option **Skripterstellung für gesamte Datenbank und alle Datenbankobjekte** beibehalten oder auf **Bestimmte Datenbankobjekte auswählen** wechseln und hier eine individuelle Auswahl festlegen. Bei unserer begrenzten Menge von Datenbankobjekten sind beide

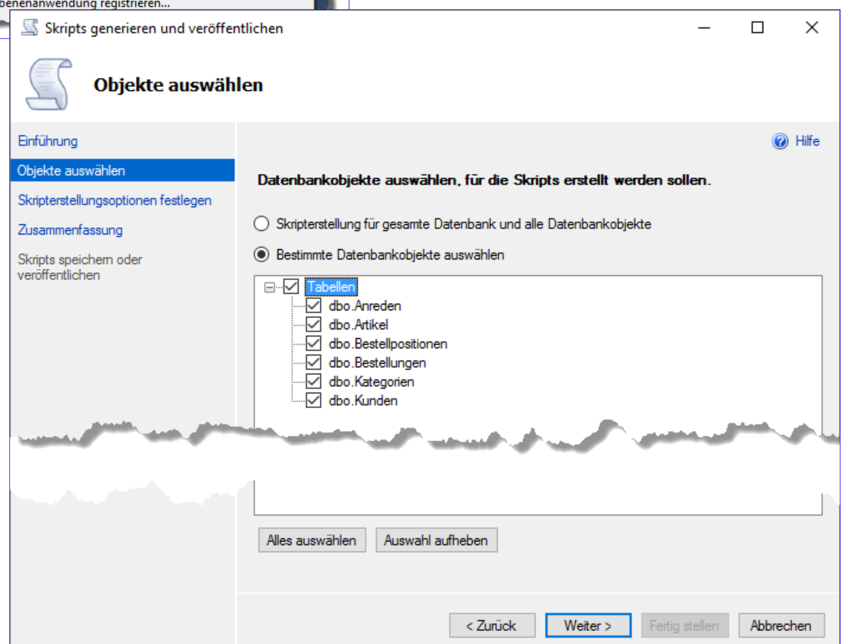


Bild 2: Auswahl der zu exportierenden Datenbankobjekte

Methoden gleich schnell ausgeführt. Aber es gibt einen entscheidenden Unterschied:

Wenn Sie **Bestimmte Datenbankobjekte auswählen** selektieren, werden, selbst wenn Sie dann alle Elemente selektieren, nur die **CREATE TABLE**-Statements für die einzelnen Tabellen (beziehungsweise **CREATE...**-Anweisungen für die übrigen Objekttypen) erstellt, aber kein **CREATE DATABASE** – Sie können mit dem erstellten Skript dann keine neue Datenbank generieren, sondern lediglich die Datenbankobjekte in eine bestehende Datenbank eintragen.

Wenn Sie eine komplett neue Datenbank erstellen wollen, wählen Sie die Option **Skripterstellung für gesamte Datenbank und alle Datenbankobjekte**.

Einen Dialogschritt weiter unter **Skripterstellungsoptionen festlegen** finden Sie die Optionen aus Bild 3. Hier legen Sie zunächst als Ausgabetyp die Option **Skripts an einem bestimmten Speicherort speichern** aus. Darunter finden Sie reichlich Möglichkeiten, um das Ziel des Exports zu definieren.

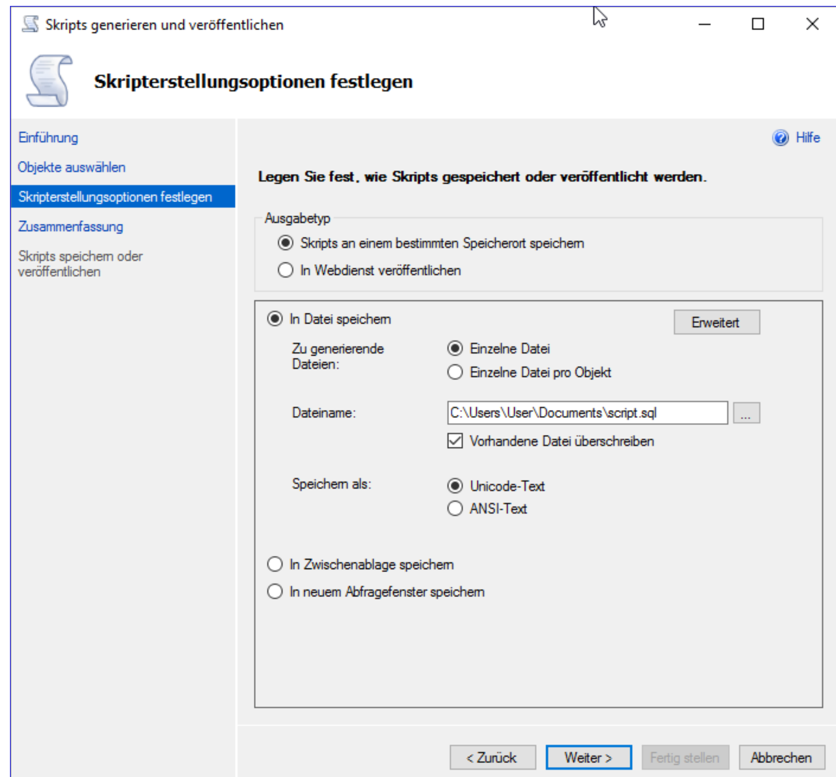


Bild 3: Festlegen der Optionen zum Exportieren der Datenbankobjekte

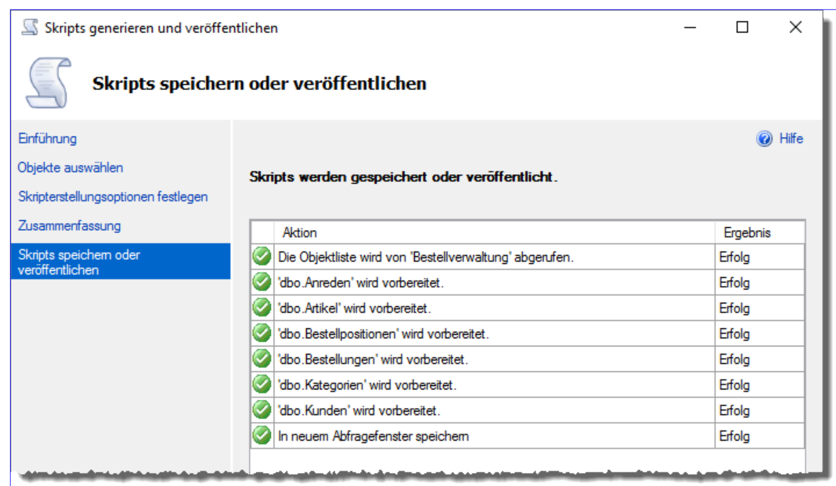


Bild 4: Ergebnis des Exports

Die erste Option geht davon aus, dass Sie die Definition aller Elemente in einer einzelnen Datei speichern wollen. Dazu geben Sie dann den gewünschten Dateinamen an und ob eventuell bereits vorhandene Dateien gleichen Namens überschrieben werden sollen. Außerdem stellen Sie hier das Exportformat fest und wählen dabei zwischen **Unicode-Text** und **ANSI-Text**.

Vielleicht wollen Sie aber gar keine Datei anlegen, sondern sich das Skript erst einmal ansehen oder es in die Zwischenablage speichern? Beides ist kein Problem, wenn Sie die zweite Option **In Zwischenablage speichern** oder die dritte Option **In neuem Abfragefenster speichern** selektieren.

Wir wählen letztere Version, um uns das Ergebnis zunächst anzusehen. Der Assistent liefert uns zur Unterstützung einen

```
USE [Bestellverwaltung]
GO
/****** Object:  Table [dbo].[Anreden]    Script Date: 17.11.2016 20:26:12 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Anreden](
    [ID] [int] IDENTITY(1,1) NOT NULL,
    [Bezeichnung] [nvarchar](50) NOT NULL,
 CONSTRAINT [PK_Anrede] PRIMARY KEY CLUSTERED
(
    [ID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, A
) ON [PRIMARY]

GO
/****** Object:  Table [dbo].[Artikel]    Script Date: 17.11.2016 20:26:12 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Artikel](
    [ID] [int] IDENTITY(1,1) NOT NULL,
    [Bezeichnung] [nvarchar](255) NOT NULL,
    [Einzelpreis] [money] NOT NULL,
    [Mehrwertsteuersatz] [decimal](3, 2) NOT NULL,
    [KategorieID] [int] NOT NULL,
```

Bild 5: Das fertige SQL-Skript

Bericht mit den Ergebnissen der Skripterstellung (siehe Bild 4).

Ergebnis ansehen

Nach dem Schließen des Assistenten können Sie zum SQL Server Management Studio zurückwechseln und finden dort ein neues Abfragefenster wie in Bild 5 vor. Das sieht schon recht gut aus – das Skript liefert die SQL-Anweisungen zum Erstellen der Tabellen unserer Datenbank. Wenn wir jedoch ein wenig weiter nach unten scrollen, stellen wir fest, dass dies leider alles ist: Neben ein paar weiteren Anweisungen zum Ändern einiger Felder in Fremdschlüsselfelder folgen keine weiteren Befehle. Damit erstellt das Skript zwar die gewünschte Tabellenstruktur, füllt die Tabellen aber nicht mit den Daten der in das Skript exportierten Datenbank. Außerdem fällt auf, dass man auch die Datenbank noch selbst anlegen muss – die erste Anweisung des Skripts lautet **USE [Bestellverwaltung]** statt **CREATE DATABASE [Bestellverwaltung]**. Eine bestehende Datenbank dieses Namens ist also Voraussetzung für das Funktionieren des Skripts.

Anpassen der »Weiteren Einstellungen«

Doch wir haben noch nicht alle Möglichkeiten des Skript-Assistenten genutzt, denn wir haben die Schaltfläche **Erweitert** im Schritt **Skripterstellungsoptionen festlegen** ignoriert.

Wenn wir den Vorgang nochmal starten und in diesem Schritt die erweiterten Optionen anzeigen lassen, erhalten wir den Dialog aus Bild 6. Hier finden Sie nun eine ganze Reihe Optionen, mit denen Sie das Aussehen des Exports sehr fein einstellen können. Für uns sind hier die folgenden Optionen interessant:

Datentypen, für die ein Skript erstellt wird: Legt fest, ob Sie nur die Daten, nur das Schema oder Schema und Daten

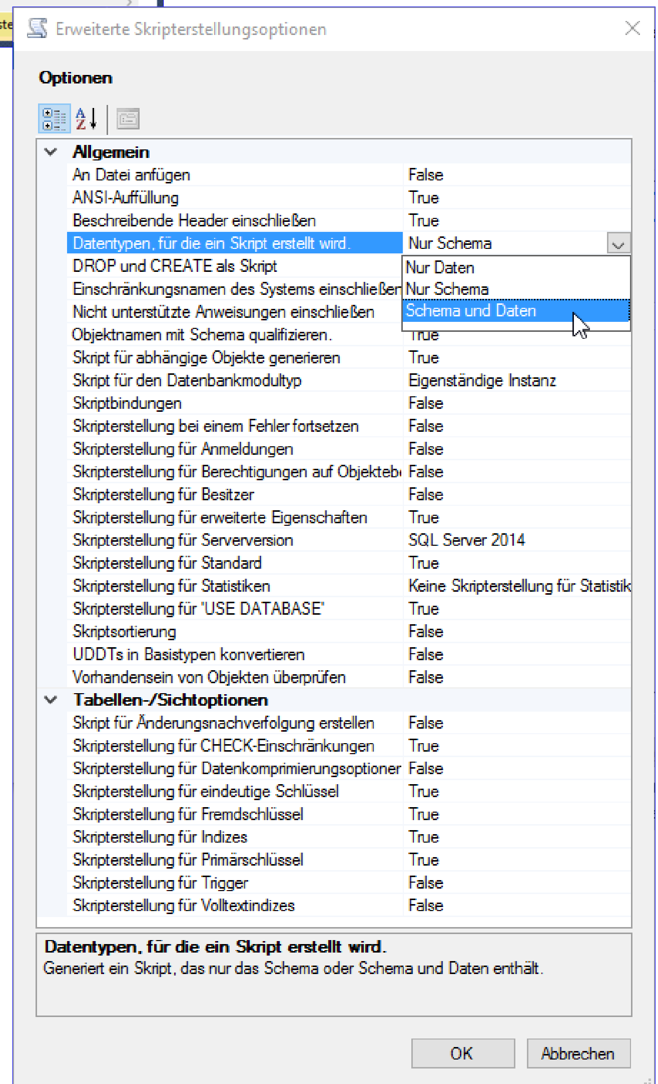


Bild 6: Optionen für das Erstellen des Skripts

exportieren wollen. Wir benötigen in diesem Fall Schema und Daten.

DROP und CREATE als Skript: Dies legt fest, ob zu jedem zu berücksichtigenden Datenbankobjekt eine **DROP**-Anweisung (zum Löschen), eine **CREATE**-Anweisung (zum Erstellen) oder beide angelegt werden sollen. Wir benötigen auf jeden Fall die **CREATE**-Anweisungen. Wenn Sie die Datenbankobjekte in eine bestehende Datenbank importieren wollen, die bereits Datenbankobjekte gleichen Namens enthält, sollten Sie zusätzlich die **DROP**-Anweisungen hinzufügen. Sie können auch ein Skript erstellen, das lediglich Datenbankobjekte löscht!

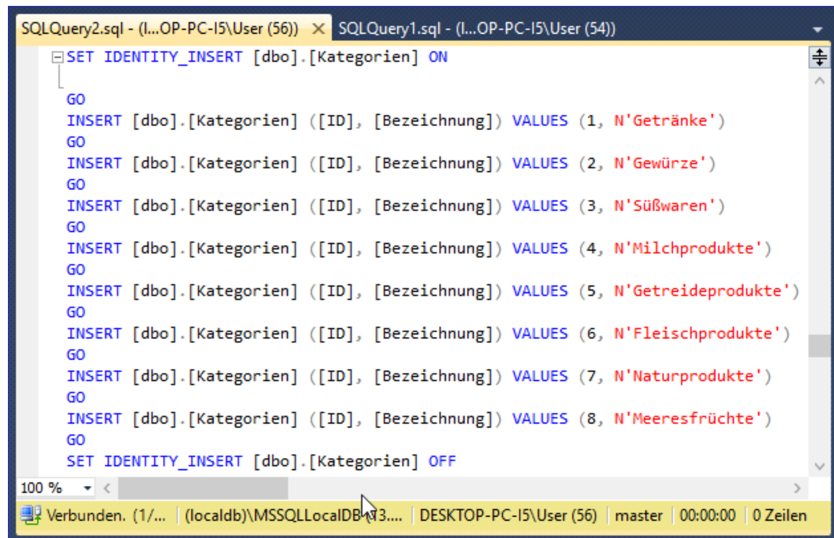


Bild 7: Skript mit **INSERT INTO**-Anweisungen

Mit **Skripterstellung für Serverversion** können Sie eine Zielversion angeben. Hier sind sie aus verständlichen Gründen auf alle zum Zeitpunkt der Veröffentlichung dieser Version von SQL Server Management Studio verfügbaren Versionen des SQL Servers beschränkt.

Wenn Sie die Skriptgenerierung nun erneut starten, erhalten Sie im erstellten Skript auch die SQL-Anweisungen, welche die Daten der Ursprungsdatenbank in die neu erstellten Tabellen einträgt (siehe Bild 7).

Datenbank neu erstellen

Auf die oben beschriebene Weise werden nun unter anderem die Beispielbanken für das Magazin bereitgestellt. Sie können die resultierenden Skriptdateien nun dazu nutzen, die Beispieldatenbank auf Ihrem eigenen Rechner anzulegen. Dazu müssen Sie lediglich eine der folgenden Aktionen durchführen:

- Das Skript etwa aus einem Texteditor in die Zwischenablage kopieren und von dort in einer neuen Abfrage im SQL Server Management Studio einfügen. Nach dem Betätigen der Taste **F5** wird das Skript nun ausgeführt und die Datenbank samt Tabellen und Daten erstellt.
- Sie können auch eine Datei mit dem Skript, beispielsweise **Bestellverwaltung.sql**, mit dem SQL Server Management Studio öffnen und dann per **F5** ausführen.

In beiden Fällen kann es natürlich zu Problemen kommen, die entweder mit der Kompatibilität des Skriptes mit der verwendeten Version oder auch mit eventuell bereits vorhandenen Datenbankdateien zusammenhängen. Dann werden die fehlerhaften Zeilen jedoch ausgegeben – beim Versuch, eine vorhandene Datenbank zu überschreiben, etwa so:

```
Meldung 1801, Ebene 16, Status 3, Zeile 4
Database 'Bestellverwaltung' already exists. Choose a
different database name.
Meldung 5069, Ebene 16, Status 1, Zeile 46
ALTER DATABASE statement failed.
Die Abfrage wurde vom Benutzer abgebrochen.
```

Datenbank in Visual Studio erstellen

Wenn Sie SQL Server Management Studio nicht installiert haben oder lieber alles von einer Plattform aus erledigen, können Sie die neue SQL Server-Datenbank auch von Visual Studio aus installieren. Dazu wählen Sie den Menübefehl **Ansicht|SQL Server-Objekt-Explorer** aus, was den gleichnamigen Bereich in Visual Studio einblendet. Dort können Sie über den Kontextmenü-Eintrag **Neue Abfrage...** des Daten-

Die MessageBox-Klasse

Die MessageBox-Klasse unter C# bietet ähnliche Features wie die von VBA bekannte MsgBox-Funktion. Allerdings ist es erstens eine Klasse und zweitens kann sie natürlich doch noch einiges mehr als das VBA-Pendant. Dieser Artikel zeigt, wie Sie Meldungsfenster unter C# anzeigen und diese mit Texten, Schaltflächen und anderen Elementen ausstatten. Schließlich interessiert uns auch noch, wie wir die vom Benutzer angeklickten Schaltflächen und somit die Antworten auf die in der Meldung gestellten Fragen auswerten können.

Unter C# nutzen wir also eine Klasse namens **MessageBox**. Diese ist eine statische Klasse, das heißt, dass Sie diese nicht instanzieren müssen, bevor Sie auf ihre Methoden und Eigenschaften zugreifen. Während Sie unter VBA einfach die **MsgBox**-Funktion unter Angabe des anzuzeigenden Textes aufgerufen haben, brauchen Sie bei der **MessageBox** eine entsprechende Methode, die in diesem Fall **Show** heißt.

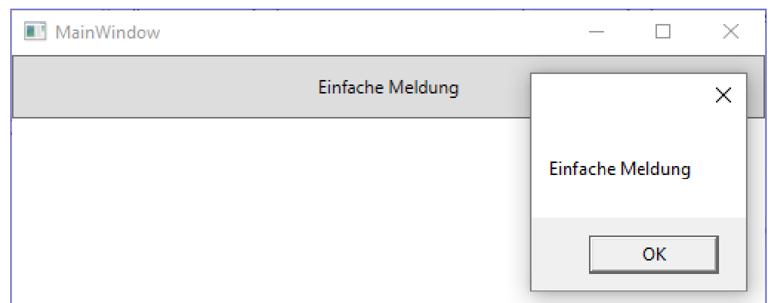


Bild 1: MessageBox mit WPF

Im Beispielprojekt zu diesem Artikel haben wir ein einfaches WPF-Fenster mit Schaltflächen gefüllt, welche die in diesem Artikel vorgestellten Meldungsfenster aufruft (siehe Bild 1). Das erste soll einfach nur einen Text anzeigen. Die Methode, die durch das **Click**-Ereignis der entsprechenden Schaltfläche aufgerufen wird, sieht wie folgt aus und nutzt die erste der zwölf angebotenen Überladungen:

```
private void btnEinfacheMeldung_Click(object sender,
        RoutedEventArgs e) {
    MessageBox.Show("Einfache Meldung");
}
```

Im Gegensatz etwa zum **Window**-Objekt, das die beiden Methoden **Show** und **ShowDialog** zum Anzeigen anbietet, gibt die **MessageBox**-Klasse nur die **Show**-Methode her. Das bedeutet allerdings nicht, dass Meldungsfenster nicht als modale Fenster ausgegeben werden – ganz im Gegenteil: **Show** liefert immer ein modales Fenster. Dementsprechend müssen Sie es erst mit einer der angezeigten Schaltflächen schließen, bevor die aufrufende Methode fortgesetzt wird.

Titel anzeigen

Mit der folgenden Überladung übergeben Sie im zweiten Parameter den anzuzeigenden Titel für die **MessageBox**:

```
MessageBox.Show("Meldung mit Titel", "Titelzeile");
```

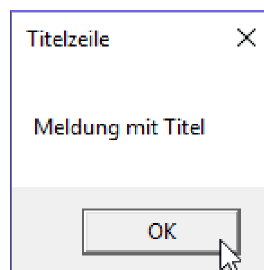


Bild 2: MessageBox mit Titelzeile

Dies sieht dann etwa wie in Bild 2 aus.

Schaltflächen der MessageBox

Die **MessageBox**-Klasse kann natürlich noch mehr Schaltflächen liefern als nur eine **OK**-Schaltfläche. Dazu verwenden Sie eine weitere Überladung, die als ersten Parameter den Text, als zweiten den Titel und als dritten eine Konstante für die anzuzeigenden Schalt-

flächen entgegennimmt. Die möglichen Werte lauten hier (siehe auch Bild 3):

- **OK**
- **OKCancel**
- **YesNo**
- **YesNoCancel**

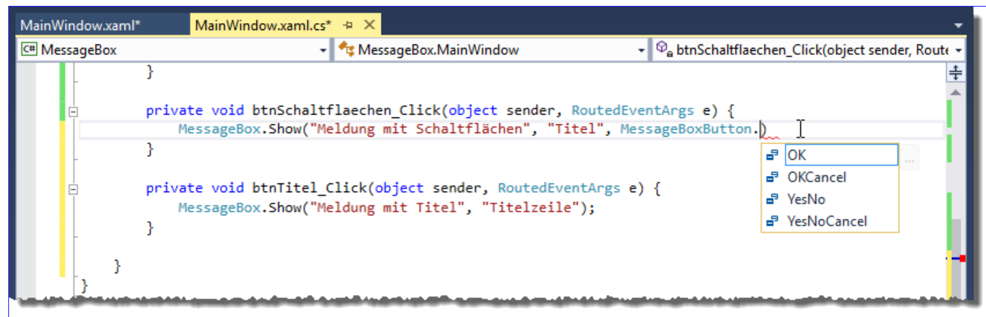


Bild 3: Auswahl der Schaltflächen-Kombinationen

Die folgende Anweisung ruft beispielsweise das Meldungsfenster aus Bild 4 hervor:

```
MessageBox.Show("Meldung mit Schaltflächen", "Titel", 
MessageBoxButton.YesNoCancel);
```

Hier ist noch anzumerken, dass die **Schließen**-Schaltfläche oben rechts nur aktiviert ist, wenn entweder nur eine Schaltfläche angezeigt wird (also **OK**) oder wenn mehrere Schaltflächen aktiviert sind und sich die Schaltfläche **Cancel** darunter befindet.

Auf Schaltflächen reagieren

Wenn Sie nicht nur eine einfache **OK**-Schaltfläche nutzen, wollen Sie sicher erfahren, mit welcher Schaltfläche der Benutzer die **MessageBox** geschlossen hat. In diesem Fall weisen Sie das Ergebnis einfach einer Variablen zu und werten den Inhalt dieser Variablen danach aus. Die Variable erhält den Typ **MessageBoxResult** und wird wie folgt zugewiesen:

```
MessageBoxResult result = MessageBox.Show("Klicken Sie auf 
eine der Schaltflächen.", "Ja oder Nein", MessageBoxBut-
ton.YesNo);
```

Dies liefert dann die MessageBox aus Bild 5. Danach können Sie den Wert per **if**-Bedingung prüfen und eine entsprechende Bestätigung ausgeben – natürlich per **MessageBox**:

```
if (result==MessageBoxResult.Yes) {
    MessageBox.Show("Sie haben auf 'Ja' geklickt.");
}
else {
    MessageBox.Show("Sie haben auf 'Nein' geklickt.");
}
```

Für die Antwort gibt es die folgenden Konstanten:

- **Cancel**
- **No**
- **Ok**
- **Yes**

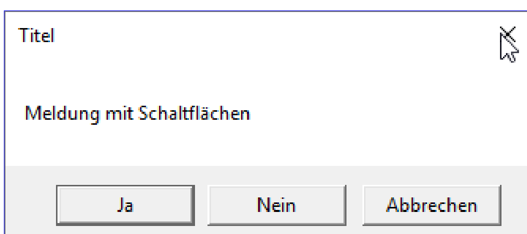


Bild 4: MessageBox mit drei Schaltflächen

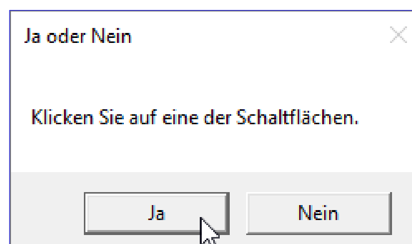


Bild 5: Testen der beiden Schaltflächen

Wie oben erwähnt, wird die **Schließen**-Schaltfläche des Fensters nur aktiviert, wenn entweder nur die **OK**-Schaltfläche oder aber eine der Kombinationen mit der **Cancel**-

InputBox im Eigenbau

Wer von Access/VBA kommt, kann unter WPF die »MsgBox«-Anweisung leicht mit den Funktionen der Klasse »MessageBox« ersetzen. Allerdings sucht man vergeblich nach einem Pendant zur InputBox-Funktion, die ja ein einfaches Eingabefenster für die Eingabe eines Textes zur Verfügung stellt. Doch das ist kein Problem: Unter C#/WPF können Sie ja eigene Fenster erstellen. Warum dann nicht ein Fenster mit Titel, Text, Eingabefeld und OK/Abbrechen-Schaltflächen bauen und dieses bei Bedarf aufrufen?

Vorgaben InputBox

Was soll eine InputBox überhaupt alles leisten? Wir richten uns im Wesentlichen nach der InputBox von Access/VBA. Diese wird wie folgt aufgerufen, wenn man die hinteren Parameter etwa für die Position und für die Onlinehilfe weglässt:

```
Debug.Print InputBox("Geben Sie einen Text ein.", "Texteingabe", "[Hier eingeben]")
```

Die **Debug.Print**-Anweisung gibt den zurückgelieferten Wert im Direktfenster des VBA-Editors aus. Wie Sie sehen, verwenden wir hier drei Parameter:

- **Prompt**: Text, der den Benutzer informiert, welche Eingabe von ihm erwartet wird
- **Title**: Text für die Titelleiste des Fensters (optional)
- **Default**: Text, der als Vorgabe im Textfeld angezeigt werden soll (optional)

Die InputBox selbst sieht dann wie in Bild 1 aus. Hier gibt es vier Steuerelemente: das Bezeichnungsfeld für den Text aus dem Parameter **Prompt**, das Textfeld zur Eingabe des gesuchten Wertes durch den Benutzer, das gegebenenfalls den Wert aus **Default** anzeigt, sowie die beiden Schaltflächen **OK** und **Abbrechen**.

Wenn der Benutzer die **OK**-Schaltfläche betätigt, soll die Funktion **InputBox** den aktuell im Textfeld befindlichen Text zurückliefern, bei Betätigung der **Abbrechen**-Schaltfläche soll eine leere Zeichenfolge zurückgegeben werden.

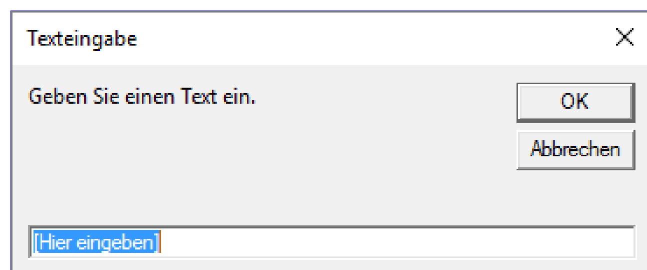


Bild 1: InputBox unter VBA

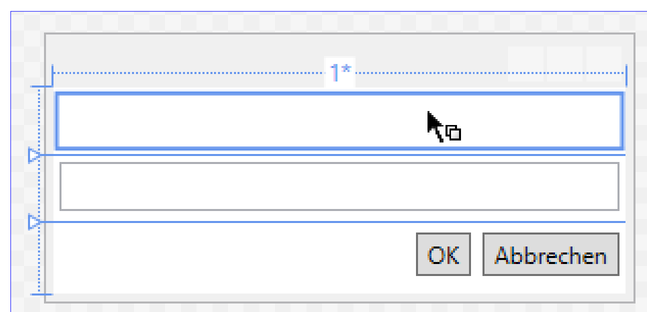


Bild 2: Fenster zur Anzeige einer InputBox im Entwurf

Wenn der Benutzer die Eingabetaste drückt, löst er die Funktion der **OK**-Schaltfläche aus, die **Escape**-Taste bewirkt selbiges für die **Abbrechen**-Schaltfläche.

InputBox als Fenster

Um ein Projekt mit einem Fenster auszustatten, das als InputBox dient, fügen Sie diesem einfach ein normales Objekt des Typs **Fenster (WPF)** hinzu. Nennen Sie das Fenster im Dialog **Neues Element hinzufügen** schlicht **InputBox**. Das Fenster statten Sie mit den Steuerelementen aus Bild 2 aus (das Bezeichnungsfeld haben wir markiert, damit man es überhaupt erkennt).

```
<Window x:Class="InputBoxImEigenbau.InputBox"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:InputBoxImEigenbau"
        mc:Ignorable="d"
        Title="{Binding Titel}" Height="136" Width="300" WindowStartupLocation="CenterScreen">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="33"></RowDefinition>
            <RowDefinition Height="33"></RowDefinition>
            <RowDefinition Height="33"></RowDefinition>
        </Grid.RowDefinitions>
        <Label x:Name="lblBezeichnung" Grid.Row="0" Margin="3"></Label>
        <TextBox x:Name="txtEingabe" Grid.Row="1" Margin="3" Height="25"></TextBox>
        <StackPanel Orientation="Horizontal" Grid.Row="2" HorizontalAlignment="Right">
            <Button x:Name="btnOK" Height="22" Padding="5,2,5,2" Margin="3" Content="OK" Click="btnOK_Click"
                IsDefault="True"></Button>
            <Button x:Name="btnCancel" Height="22" Padding="5,2,5,2" Margin="3" Content="Abbrechen"
                Click="btnCancel_Click" IsCancel="True"></Button>
        </StackPanel>
    </Grid>
</Window>
```

Listing 1: Code des WPF-Fensters, das als InputBox dienen soll

Der XAML-Code des Fensters sieht wie in Listing 1 aus.

Hier stecken einige Besonderheiten: Das Fenster soll zentriert erscheinen, also erhält das **Window**-Element den Wert **CenterScreen** für die Eigenschaft **WindowStartupLocation**.

Die Steuerelemente haben wir in einem Grid arrangiert. In der letzten Zeile des Grids haben wir außerdem ein **StackPanel** mit dem Wert **Horizontal** für das Attribut **Orientation** eingebaut, damit wir darin die Schaltflächen nebeneinander anordnen können. Die Schaltfläche **cmdOK** erhält für die Eigenschaft **IsDefault** den Wert **true**, damit sie beim Betätigen der Eingabetaste ausgelöst wird. **cmdAbbrechen** statten wir mit dem Wert **true** für das Attribut **IsCancel** aus, so löst das Betätigen der Taste **Escape** diese Schaltfläche aus.

Code hinter dem InputBox-Fenster

Die Code behind-Datei zu unserer InputBox finden Sie in Listing 2. Die Konstruktor-Methode namens **InputBox** erwartet die Übergabe dreier Parameter namens **Titel**, **Bezeichnung** und **Standardwert**. Sie initialisiert das Fenster und trägt dann den Wert des Parameters **Bezeichnung** für das Attribut **Content** des Elements **lblBezeichnung** ein. Der Wert des Parameters **Titel** wird zunächst überprüft. Wenn der Aufruf diesen Parameter nicht enthält, erhält dieser zunächst als Standardwert eine leere Zeichenfolge. Die Methode prüft dann, ob **Titel** leer ist, und ersetzt diesen Wert dann durch den aktuellen Anwendungstitel, der mit **System.AppDomain.CurrentDomain.FriendlyName** ermittelt wird. Danach landet der Inhalt von **Titel** im Attribut **Title** des **Window**-Elements, das mit **this** referenziert wird. Der Wert des Parameters

```
using System.Windows;

namespace InputBoxImEigenbau {
    public partial class InputBox : Window {
        string eingabe;
        public InputBox(string Bezeichnung, string Titel = "", string Standardwert="") {
            InitializeComponent();
            this.lblBezeichnung.Content = Bezeichnung;
            if (Titel=="") {
                Titel = System.AppDomain.CurrentDomain.FriendlyName;
            }
            this.Title = Titel;
            this.txtEingabe.Text = Standardwert;
            this.txtEingabe.SelectAll();
            this.txtEingabe.Focus();
        }
        public string Eingabe
        {
            set { eingabe = value; }
            get { return eingabe; }
        }
        private void btnCancel_Click(object sender, RoutedEventArgs e) {
            eingabe = "";
            this.Visibility = Visibility.Hidden;
        }

        private void btnOK_Click(object sender, RoutedEventArgs e) {
            eingabe = this.txtEingabe.Text;
            this.Visibility = Visibility.Hidden;
        }
    }
}
```

Listing 2: Code behind-Modul des Fensters **InputBoxImEigenbau**

Standardwert wird schließlich in die Eigenschaft **Text** des Textfeldes **txtEingabe** eingesetzt.

Das Textfeld erfährt noch eine Sonderbehandlung: Die Methode **SelectAll** markiert den kompletten Inhalt und die Methode **Focus** verschiebt den Fokus auf dieses Steuerelement.

Um das Ergebnis zu betrachten, benötigen wir jedoch noch eine Methode, die das **InputBox**-Fenster anzeigt. Diese fügen wir einer neuen Schaltfläche des Fensters **MainWindow** hinzu. Die Methode lautet wie folgt:

```
private void btnInputBox_Click(object sender,
        RoutedEventArgs e) {
    InputBox inputBox = new InputBox("Bezeichnung",
        "Beispieltitel", "Standardwert");
    inputBox.ShowDialog();
    MessageBox.Show(inputBox.Eingabe);
    inputBox.Close();
}
```

Die Methode erstellt ein neues Objekt auf Basis der Klasse **InputBox** und übergibt der Konstruktor-Methode dabei gleich

Ribbons mit WPF

Der geneigte Access-Entwickler weiß mit dem Begriff Ribbon natürlich etwas anzufangen. Aber gibt es so etwas auch für .NET-Anwendungen? Ist das nicht ein reines Office-Feature? Immerhin gibt es ja auch in der Benutzeroberfläche von Visual Studio kein Ribbon, sondern eine Menüleiste und Symbolleisten, wie Sie sie von älteren Office-Versionen bis Office 2003 kennen (man munkelt, einige Entwickler wünschten sich die Menüleisten zurück ...). In diesem Artikel schauen wir uns an, wie weit es mit dem Ribbon unter .NET und speziell unter WPF bestellt ist und welche Unterstützung uns Visual Studio für die Programmierung von Ribbons bietet.

Um herauszufinden, ob es unter WPF Ribbons gibt und wie diese programmiert werden, starten wir mit einem neuen Projekt unter dem Namen **RibbonsInWPF** mit der Vorlage **Visual C#|Windows|WPF-Anwendungen**. Dort beginnen wir mit dem Fenster **MainWindow**, das ja automatisch als Startfenster angelegt wird.

Die schnelle Eingabe des Suchbegriffs Ribbon in das Suchfenster der Toolbox von Visual Studio liefert schon einmal keine Ergebnisse. Also werfen wir einen Blick in den Dialog **Neues Element hinzufügen**, den Sie mit dem gleichnamigen Befehl des Eintrags **Projekt** der Menüleiste öffnen. Auch hier gibt es eine Suchmöglichkeit, die wir allerdings ebenfalls ohne Erfolg nutzen. Gibt es etwa gar keine Ribbons unter WPF?

Vielleicht müssen wir dafür eine spezielle Bibliothek hinzufügen. Also öffnen wir den **Verweis-Manager** (Menüeintrag **Projekt|Verweis hinzufügen...**) und suchen hier nach dem Ausdruck **Ribbon**. Bingo! Hier werden wir endlich fündig und fügen den Eintrag aus Bild 1 per Mausklick zu den eingebundenen Referenzen hinzu. In den Steuerelementen und auch im Dialog **Neues Element hinzufügen** finden sich allerdings weiterhin keine neuen Elemente. Um es kurz zu machen: Auch in Visual Studio müssen Sie manuell XML-Code eingeben, um ein Ribbon zu definieren. Also legen wir los und fügen im **Grid**-Element ein erstes **Ribbon**-Element hinzu:

```
<Window x:Class="RibbonMitWPF.MainWindow">
    <Grid>
        <Ribbon></Ribbon>
    </Grid>
</Window>
```

Das Ergebnis bekommen wir allerdings immerhin direkt in Form eines entsprechenden Elements im Entwurf des WPF-

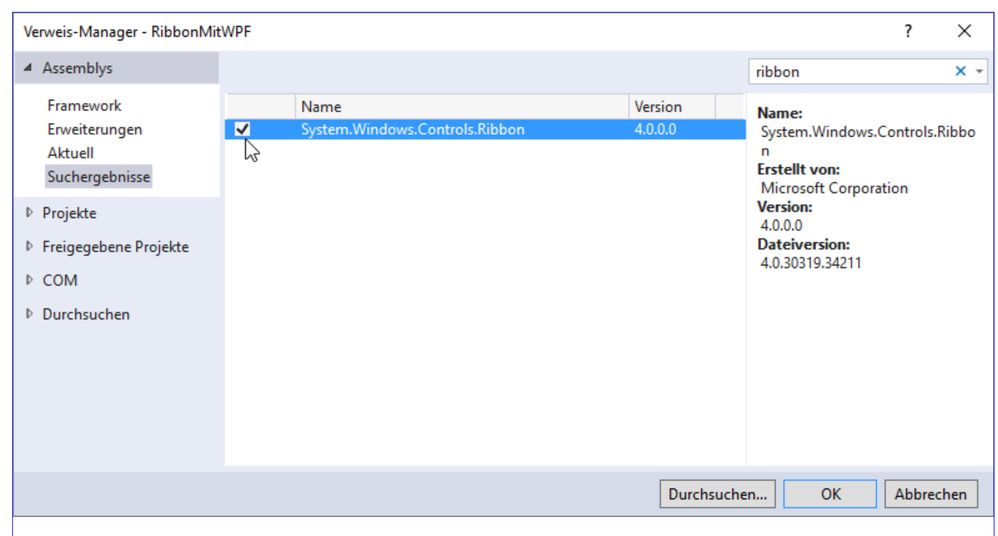


Bild 1: Hinzufügen einer Referenz auf die Bibliothek **System.Windows.Controls.Ribbon**

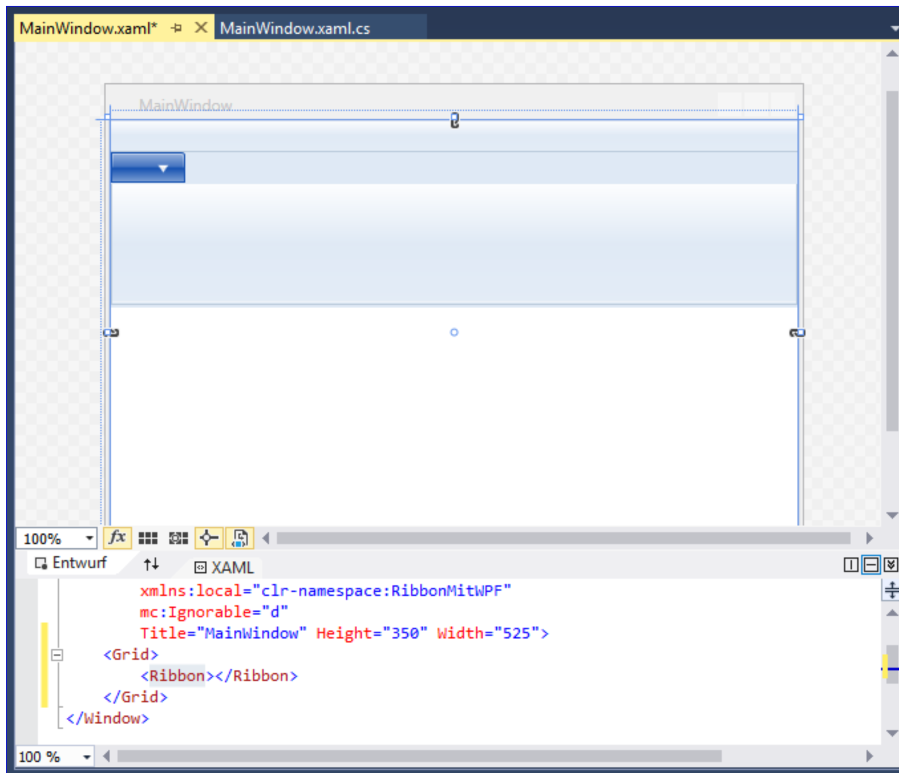


Bild 2: Hinzufügen eines leeren Ribbons

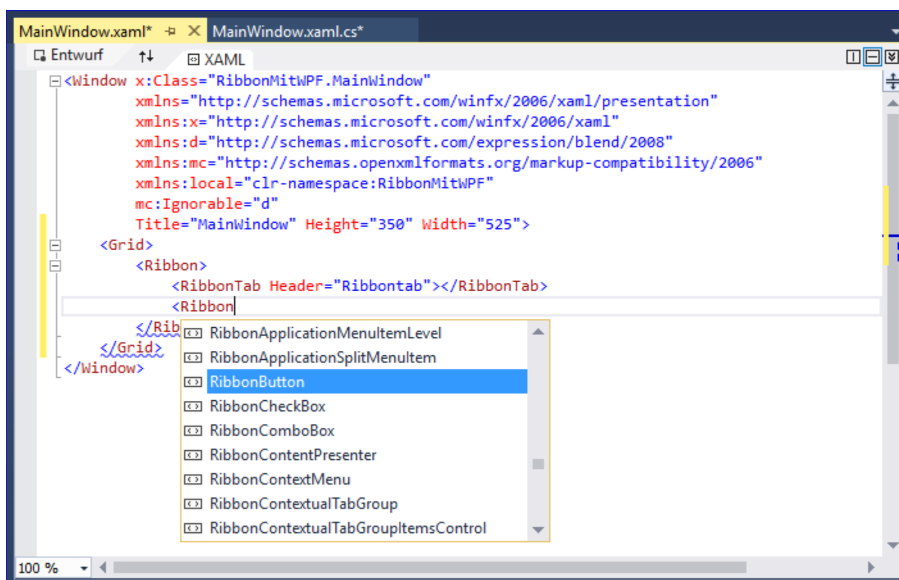


Bild 3: Zugriff auf die Ribbon-Elemente per IntelliSense

Fensters zu sehen (siehe Bild 2). Ein Klick mit der rechten Maustaste auf das Ribbon liefert ebenfalls keine Möglichkeit, auf die Schnelle ein paar Steuerelemente zum Ribbon hinzuzufügen. Also genießen wir zumindest die Bereitstellung von IntelliSense bei der manuellen Eingabe der notwendigen Elemente.

Tab-Element hinzufügen

Ein Ribbon-Tab legen Sie über das **RibbonTab**-Element an, nicht wie unter Office mit dem **tab**-Element:

```
<Ribbon>
    <RibbonTab 
Header="Ribbontab"></RibbonTab>
</Ribbon>
```

Grundsätzlich finden Sie die Ribbon-spezifischen Elemente alle schnell, wenn Sie mit IntelliSense-Unterstützung direkt die Zeichenfolge **Ribbon...** eingeben (siehe Bild 3). Hier tauchen dann auch schnell die bekannten Elemente wie **RibbonButton**, **RibbonGroup** und so weiter auf.

Schaltfläche hinzufügen

Fügen wir als Nächstes eine Schaltfläche hinzu. Wenn wir nicht achtgeben und das **RibbonButton**-Element außerhalb des **RibbonTab**-Elements im XAML-Code hinzufügen, geschieht tatsächlich das, was Sie in Bild 4 sehen: Das **RibbonButton**-Element wird oben neben dem Ribbon-Registerreiter angelegt. Ob und wozu man das braucht, erklärt sich nicht – wir wollen uns aber an den gewohnten Strukturen aus dem Office-Bereich orientieren. Also fügen wir das **RibbonButton**-Element auch nicht einfach in das **RibbonTab**-Element ein, sondern fügen zunächst ein **RibbonGroup**-Element hinzu. Grundsätzlich kann man sich als Access-Entwickler, der sich schon einmal mit der

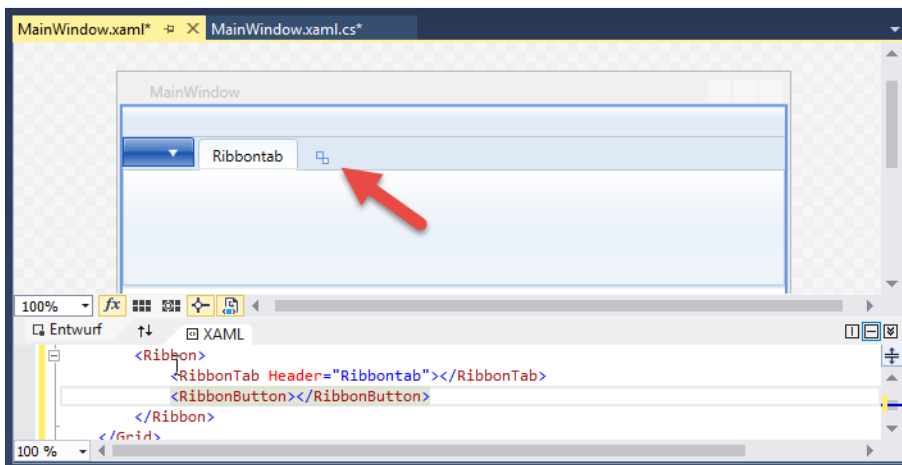


Bild 4: Ein Ribbon-Button außerhalb eines Tab-Elements

```
<Ribbon Title="Beispielribbon">
  <RibbonTab Header="Ribbontab">
    <RibbonGroup Header="Ribbongruppe 1">
      <RibbonButton Label="Beispielbutton 1"></RibbonButton>
      <RibbonButton Label="Beispielbutton 2"></RibbonButton>
      <RibbonButton Label="Beispielbutton 3"></RibbonButton>
    </RibbonGroup>
    <RibbonGroup Header="Ribbongruppe 2">
      <RibbonButton Label="Beispielbutton 4"></RibbonButton>
      <RibbonButton Label="Beispielbutton 5"></RibbonButton>
      <RibbonButton Label="Beispielbutton 6"></RibbonButton>
    </RibbonGroup>
  </RibbonTab>
</Ribbon>
```

Listing 1: Code für ein Ribbon mit zwei Gruppen und einigen Schaltflächen

Ein **RibbonTab**-Element mit zwei **RibbonGroup**- und je drei **Ribbon-Button**-Elementen definieren wir schließlich wie in Listing 1.

Das Ergebnis nach einem Klick auf die Taste **F5** zum Start des Debuggers zeigt Bild 5. Es gibt ein paar Besonderheiten, die wir an dieser Stelle klären wollen: Zunächst können Sie bereits für das **Ribbon**-Element mit dem Attribut **Title** eine Überschrift festlegen, die dann in der Abbildung beispielsweise **Beispielribbon** lautet. Dies ist unter Office so nicht möglich. Die zweite Besonderheit ist, dass es für die Beschriftungen so viele verschiedene Attribute gibt. Beim **Ribbon**-Element selbst heißt sie **Title**, beim **RibbonTab** und bei der **RibbonGroup** lautet das Attribut **Header** und bei der Schaltfläche (**RibbonButton**) heißt sie schließlich **Label**. Der Hintergrund sind die verschiedenen Basis-Elemente, von denen die Ribbon-Elemente erben. Wenn man das erstmal herausgefunden hat, gewöhnt man sich allerdings recht schnell daran.

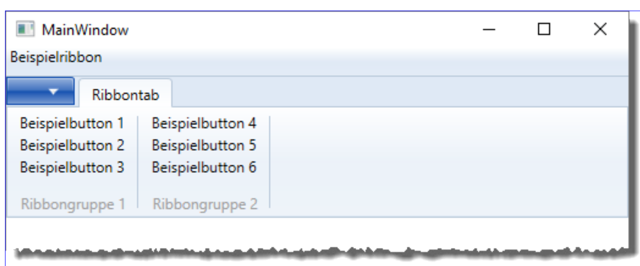


Bild 5: Ein Ribbon in einer laufenden Anwendung

Programmierung des Ribbons beschäftigt hat, merken, dass die bekannten Elemente alle mit dem Präfix **Ribbon...** ausgestattet wurden (also **RibbonTab** statt **tab**, **RibbonGroup** statt **group** et cetera).

Bilder im WPF-Ribbon

Bevor wir uns um die Funktionalität kümmern, also um die Ereignisse der einzelnen Ribbon-Elemente, wollen wir noch für eine etwas schickere Optik sorgen. Dazu gehören im Ribbon natürlich ein paar Icons. Im Gegensatz zu Access (und den übrigen Office-Anwendungen), wo einiges an Code benötigt wird, um Bilder im Ribbon anzuzeigen, gelingt dies unter WPF relativ einfach. Dazu fügen Sie zunächst die gewünschten Icons zum Projekt hinzu, indem Sie diese einfach in das Projekt im Projektmappen-Explorer ziehen. Bevor Sie damit beginnen, sollten Sie sich überlegen, ob Sie für Bild-

dateien nicht einen eigenen Ordner innerhalb des Projekts anlegen sollen. Dies erledigen Sie über den Kontextmenü-Eintrag **Hinzufügen|Neuer Ordner** des Projekt-Elements. Den neuen Ordner können Sie beispielsweise **Images** nennen. Danach ziehen wir einige Bilddateien aus dem Windows Explorer in dieses Verzeichnis. Sammlungen solcher Icons können Sie im Internet kostenlos finden, für hochwertige Kollektionen wie etwa von **iconexperience.com** müssen Sie jedoch ein paar Euro hinlegen.

Der neue Ordner **Images** mit einigen Bildern sieht dann so wie in Bild 6 aus. Wenn Sie die Einträge mit der Maus überfahren, blendet Visual Studio eine Vorschau der jeweiligen Bilder ein. Nun wollen wir die Bilder auch im Ribbon anzeigen, zunächst als Symbol für die Schaltflächen.

Dazu fügen wir den Schaltflächen eines der beiden Attribute **SmallImageSource** oder **LargeImageSource** hinzu und legen den Namen der Bilddatei inklusive Verzeichnis als Wert fest. Dies sieht beispielsweise wie folgt aus:

```
<RibbonButton Label="Beispielbutton 1"
    LargeImageSource="Images\apple.png" />
...
<RibbonButton Label="Beispielbutton 4"
   SmallImageSource="Images\ice_cream2.png" />
```

In der Anwendung erscheinen die Bilder in den veschiedenen Größen dann wie in Bild 7. Wir haben hier ausschließlich Bilder der Größe 32x32 verwendet. Üblicherweise verwendet man für das Attribut **LargeImageSource** Bilddateien dieser Größe, für das Attribut **SmallImageSource** nutzt man Bilder der Größe 16x16. Die Bilder werden aber bei Bedarf herunterskaliert. Wenn Sie über eine Icon-Sammlung mit Icons in diesen beiden Größen verfügen, sollten Sie jeweils die korrekte Größe für die entsprechenden Attribute verwenden.

Dazu müssen Sie dann zum Dateinamen der Elemente im Ordner **Images** noch ein Suffix wie etwa **_small** hinzufügen.

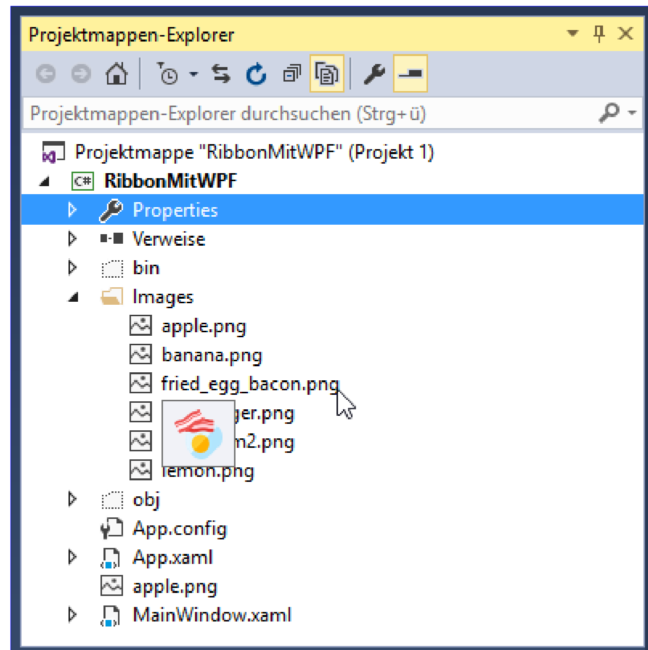


Bild 6: Bilder für die Anzeige im Ribbon im Projekt-Explorer

Ribbon-Titelzeile entfernen

Die Zeile, die durch das Ribbon-Element selbst repräsentiert wird, können Sie mit einem Trick entfernen. Dazu ändern Sie den Typ des Fenster-Objekts von **Window** auf **RibbonWindow**. Dazu sind drei Schritte nötig.

Der erste Schritt ist das Hinzufügen der Klasse mit einer entsprechenden **using**-Anweisung in der Code behind-Klasse des Fensters:

```
using System.Windows.Controls.Ribbon;
```

Danach ändern Sie den Typ der Basisklasse von **MainWindow** von **Window** auf **RibbonWindow**, sodass der Inhalt der Code behind-Klasse nun wie folgt aussieht:

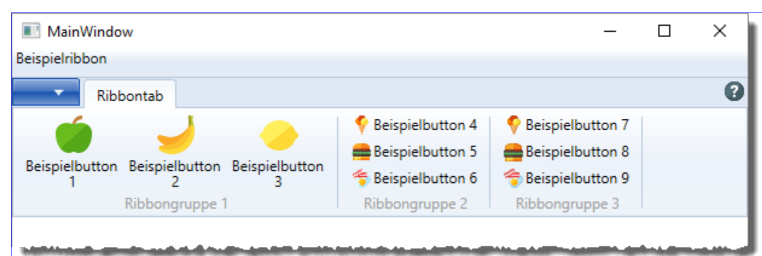


Bild 7: Ribbon-Schaltflächen mit einigen Bildern

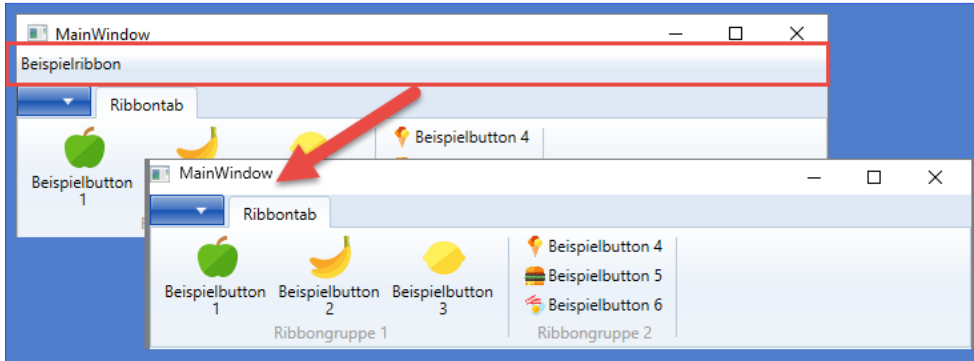


Bild 8: Ribbon mit und ohne Titelleiste

```
namespace RibbonMitWPF {
    public partial class MainWindow : RibbonWindow {
        public MainWindow() {
            InitializeComponent();
        }
    }
}
```

Schließlich ziehen Sie diese Änderung auch im XAML-Code dieser Klasse nach. Dazu ändern Sie das oberste Element ebenfalls von Window in **RibbonWindow**:

```
<RibbonWindow ... Title="MainWindow" Height="350"
Width="625">
    <Grid>
    ...
```

Wenn Sie die vorherige Ansicht und die aktuelle Version vergleichen, stellen Sie fest, dass die Zeile mit dem Text **Beispielribbon** weggefallen ist (siehe Bild 8). Übrigens: Wenn Sie das **Title**-Attribut des **RibbonWindow**-Elements in diesem Fall leer lassen, zeigt der Titel den Inhalt der Eigenschaft **Header** des **Ribbon**-Elements an.

Das Icon der Titelleiste des Fensters können Sie in diesem Fall durch Setzen des Attributs **WindowIconVisibility** auf den Wert **Collapsed** verschwinden lassen:

```
<Ribbon Title="Beispielribbon"
    WindowIconVisibility="Collapsed">
```

Schnellzugriffsleiste nutzen

Warum aber sollte man diesen Bereich überhaupt erhalten? Nun: Wenn Sie Access-Ribbons gewohnt sind, wissen Sie, dass dort oben in der Regel die Schnellzugriffsleiste untergebracht ist, die einige Standardbefehle wie **Wiederholen** oder **Rückgängig** enthält. Sie könnten also durchaus auch selbst einige solcher Befehle dort unterbringen, die immer sichtbar sind – unabhängig davon, welches **RibbonTab**-Element gerade aktiviert ist. Um Elemente zur Schnellstartleiste hinzuzufügen, legen Sie unterhalb des **Ribbon**-Elements ein Element namens **Ribbon.QuickAccessToolbar** an, dem Sie wiederum das Element **RibbonQuickAccessToolbar** unterordnen. Darin finden schließlich die gewünschten Steuerelemente Platz (siehe Bild 9):

```
<Ribbon Title="Beispielribbon">
  <Ribbon.QuickAccessToolBar>
    <RibbonQuickAccessToolBar >
      <RibbonButton SmallImageSource="Images\apple.png">
      </RibbonButton>
      <RibbonButton SmallImageSource="Images\banana.png">
      </RibbonButton>
      <RibbonButton SmallImageSource="Images\lemon.png">
      </RibbonButton>
    </RibbonQuickAccessToolBar>
  </Ribbon.QuickAccessToolBar>
  ...
</Ribbon>
```

Das Anwendungsmenü

Das blaue Element links oben im Ribbon mit dem Pfeil nach unten kennen Sie vermutlich von Office ab Version 2010. Dort können Sie damit den Backstage-Bereich öffnen, wo Sie etwa Zugriff auf die Optionen der jeweiligen Anwendung erhalten. Unter WPF können Sie dieses Element natürlich nach Ihren

Wünschen gestalten. Es heißt **RibbonApplicationMenu** und wird ebenfalls wie das **QuickAccessToolbar**-Element erst mit dem Element **Ribbon.ApplicationMenu** und dann mit **RibbonApplicationMenu** unterhalb des **Ribbon**-Elements integriert.

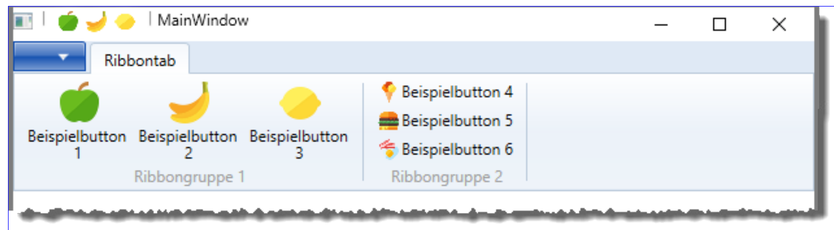


Bild 9: Ribbon mit Elementen in der Schnellzugriffsleiste

Der folgende XAML-Code liefert ein Beispiel für den Einsatz eines minimalen **RibbonApplicationMenu**-Elements. Es enthält lediglich einige Elemente des Typs **RibbonApplicationMenuItem** mit dem Text **Menu Item 1** und so weiter. Außerdem haben wir unsere im Projekt gespeicherten Bilder wieder eingesetzt – diesmal allerdings für das Attribut **ImageSource** (hier gibt es kein **SmallImageSource** oder **LargeImageSource**):

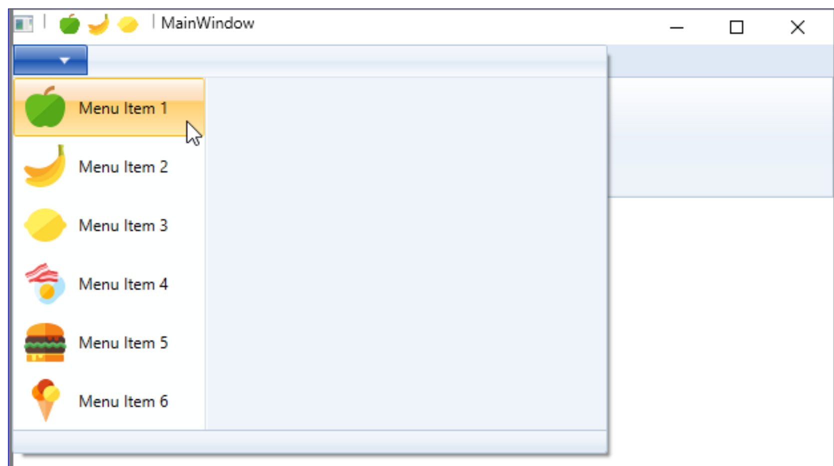


Bild 10: Ribbon mit Elementen im Anwendungsmenü

```
<Ribbon Title="Beispielribbon" >
    <Ribbon.ApplicationMenu>
        <RibbonApplicationMenu>
            <RibbonApplicationMenuItem
                Header="Menu Item 1"
                ImageSource="Images\apple.png" />
            <RibbonApplicationMenuItem
                Header="Menu Item 2"
                ImageSource="Images\banana.png" />
            <RibbonApplicationMenuItem
                Header="Menu Item 3"
                ImageSource="Images\lemon.png" />
            <RibbonApplicationMenuItem
                Header="Menu Item 4"
                ImageSource="Images\fried_egg_bacon.png" />
            <RibbonApplicationMenuItem
                Header="Menu Item 5"
                ImageSource="Images\hamburger.png" />
            <RibbonApplicationMenuItem
                Header="Menu Item 6"
                ImageSource="Images\ice_cream2.png" />
        </RibbonApplicationMenu>
    </Ribbon.ApplicationMenu>
    ...
</Ribbon>
```

Das Ergebnis sieht wie in Bild 10 aus. Wir können den einzelnen Elementen noch Untereinträge hinzufügen:

```
<RibbonApplicationMenuItem Header="Menu Item 1"
        ImageSource="Images\apple.png">
    <RibbonApplicationMenuItem Header="Menu Item 1-1"
        ImageSource="Images\banana.png"/>
    <RibbonApplicationMenuItem Header="Menu Item 1-2"
        ImageSource="Images\lemon.png" />
</RibbonApplicationMenuItem>
```

Gegebenenfalls taucht hier ein Fehler auf (**Der Index "0" befindet sich außerhalb des gültigen Bereichs der PathParameters-Liste mit der Länge "0".**). Diesen können Sie ignorieren. Der Code liefert nun das Ergebnis aus Bild 11.

EDM: Einfaches Detailfenster

Wie Sie ein DataGrid-Steuerelement mit den Daten aus einem Entity Data Model füttern, haben Sie bereits im Artikel »Einführung in das Entity Framework« erfahren. Allerdings möchten Sie gelegentlich auch in den Datensätzen einer Tabelle blättern, die enthaltenen Daten ändern oder neue Datensätze hinzufügen – ganz, wie es unter Access in einem einfachen Detailformular möglich war. Dieser Artikel zeigt, wie Sie dies mit den eingebauten Funktionen von Visual Studio einfach bewerkstelligen können.

Das Ziel dieses Artikels ist es also, ein neues Fenster zu erstellen, das über das Entity Data Model an eine Tabelle gebunden ist – in diesem Fall an die Tabelle **Kunden** unserer Beispieldatenbank namens **Bestellverwaltung** – und das die Daten jeweils eines Datensatzes in entsprechenden Steuerelementen anzeigt. Außerdem wollen wir dem Fenster einige Schaltflächen hinzufügen, mit denen der Benutzer zum ersten, vorherigen, nächsten und letzten und zu einem neuen Datensatz springen kann. Schließlich soll der Benutzer auch Änderungen durchführen und diese speichern können.

Wir verwenden dabei ein neues Projekt des Typs **Visual C#|WPF-Anwendung**. Anschließend erstellen Sie nach bewährter Art ein Entity Data Model mit den Daten der zu verwendenden Datenbank.

Fügen Sie also ein neues Element des Typs **ADO.NET Entity Data Model** zum Projekt hinzu und nennen Sie es **BestellverwaltungModel**. Im folgenden Dialog behalten Sie den Eintrag **EF Designer aus Datenbank** bei. Im nächsten Dialog klicken Sie auf **Neue Verbindung** und wählen dort **Microsoft SQL Server** aus.

Im Dialog Verbindungseigenschaften geben Sie **(localdb)\MSSQLLocalDB** als Servername und **Bestellverwaltung** unter **Mit Datenbank verbinden** ein (siehe Bild 1). Wir gehen an dieser Stelle davon aus, dass Sie die Bestellverwaltung-Datenbank bereits an den SQL Server oder LocalDB angehängt haben (dies erledigen Sie am einfachsten über den SQL Server Management Studio).

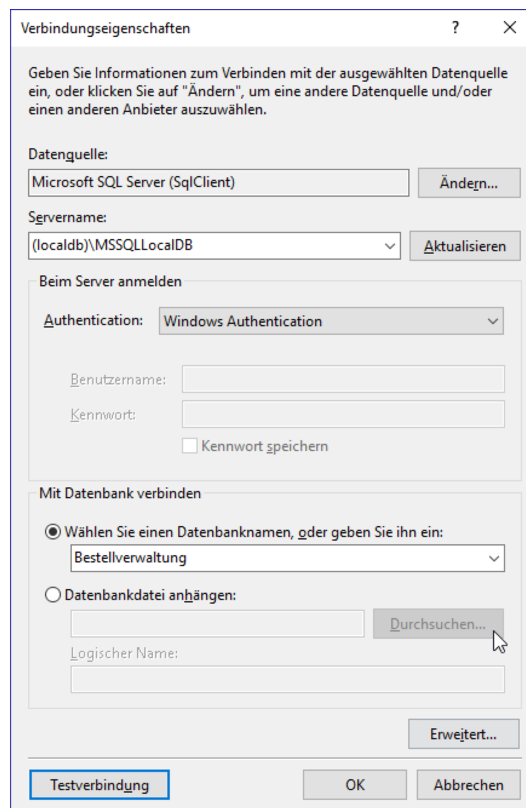


Bild 1: Hinzufügen der Verbindung

Kehren Sie zum vorherigen Dialog zurück und klicken Sie auf **Weiter**. Die Verbindungseinstellungen wollen wir unter dem Namen **BestellverwaltungEntities** speichern. Behalten Sie im folgenden Schritt den Wert **Entity Framework 6.x** bei.

Unter **Wählen Sie Ihre Datenbankobjekte und Einstellungen** klicken Sie einfach den Eintrag **Tabellen** an. Damit fügen Sie alle Tabellen zum Entity Data Model hinzu. Unter **Modelnamespace:** können Sie den Wert **BestellverwaltungModel** beibehalten.

Danach erstellt Visual Studio das Entity Data Model und zeigt dieses in einem Diagramm an. Ändern Sie hier noch die Namen der Klassen vom Plural in den Singular, also etwa von **Kategorien** in **Kategorie**.

Nun erstellen Sie das Projekt erstmalig mit dem Menüeintrag **Erstellen|<Projektname> erstellen**.

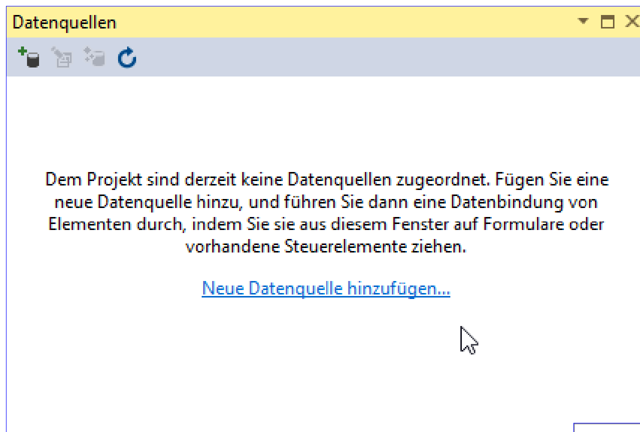


Bild 2: Noch keine Datenquellen vorhanden?

Felder zum Fenster hinzufügen

Nun schauen wir uns den einfachsten Weg an, um die Felder einer Klasse des Entity Data Models zu einem Fenster hinzuzufügen – und zwar zum Fenster **MainWindow.xaml**. Dies wollen wir dazu einfach aus einer entsprechenden Liste in das Fenster ziehen – ähnlich, wie es bei der Feldliste beim Formularentwurf unter Access geschieht.

Um dies zu erledigen, müssen wir zunächst auf Basis des Entity Data Models eine Datenquelle erstellen. Dazu aktivieren Sie, sofern noch nicht geschehen, den Bereich **Datenquellen**, und zwar über den Menübefehl **Ansicht|Weitere Fenster|Datenquellen**. Der Dialog erwartet Sie dann wie in Bild 2.

Hier benötigen wir nun eine Datenquelle. Dazu klicken Sie auf den Link **Neue Datenquelle hinzufügen...** und öffnen so den Assistenten zum Konfigurieren von Datenquellen. Dieser bietet vier mögliche Quellen für die Daten an, von denen wir den Eintrag **Objekt** auswählen (siehe Bild 3).

Im folgenden Schritt werden dann die Objekte des Projekts angezeigt. Unter dem Projektnamen finden Sie einen weiteren Eintrag gleichen Namens, der dann schließlich die Klassen des Projekts anbietet, unter anderem eben auch die Klassen des Entity Data Models, also **Kunde**, **Artikel**, **Bestellung** et cetera. Hier wählen Sie nun wie in Bild 4 den Eintrag **Kunde** aus.

Nach einem Klick auf die Schaltfläche **Fertigstellen** erscheinen die Felder der Klasse **Kunde** im **Datenquellen**-Dialog (siehe Bild 5).

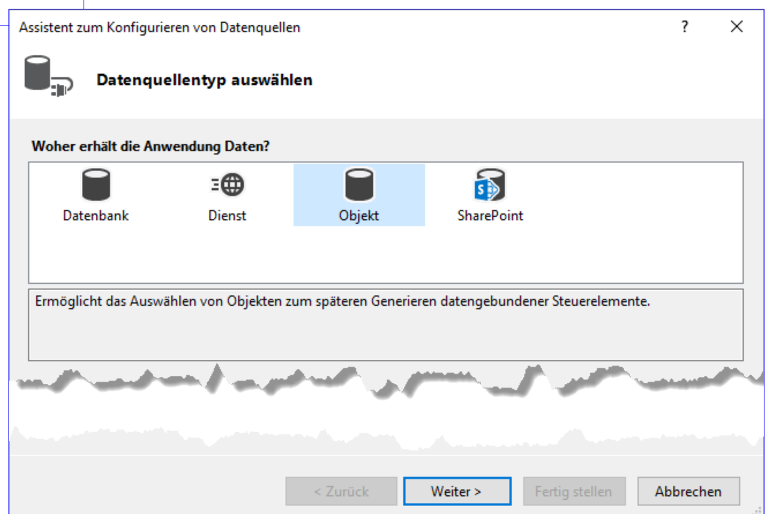


Bild 3: Die Daten sollen aus einem Objekt bezogen werden.

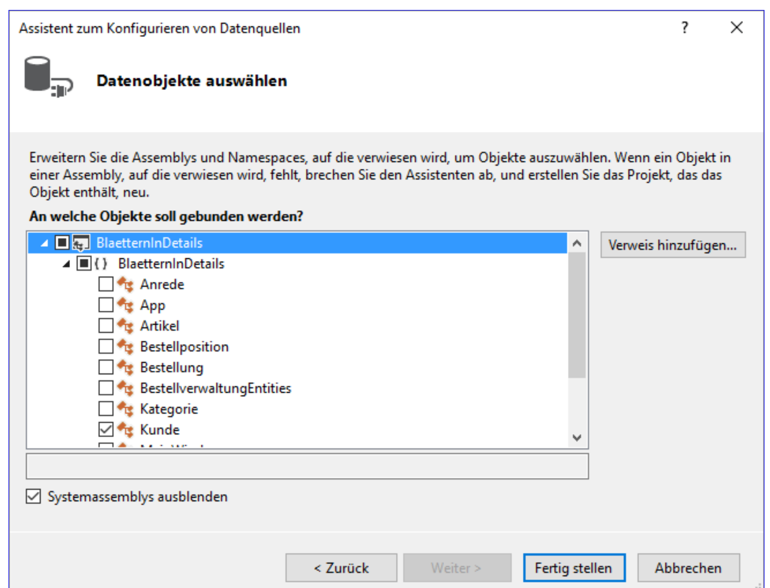


Bild 4: Auswählen des Quellobjekts, hier namens **Kunde**

Damit haben wir uns außerdem einen neuen Eintrag im neuen Ordner **DataSources** im Bereich **Properties** des Projektmappen-Explorers geholt – und zwar namens **Kunde.datasource** (siehe Bild 6).

Felder per Drag and Drop

Wenn Sie nun die Entwurfsansicht des Fensters **MainWindow.xaml** aktivieren, ändert sich die Ansicht im Bereich **Datenquellen** ein wenig (siehe Bild 7). Sie können nun per Mausklick auf die einzelnen Einträge jeweils ein Kombinationsfeld aktivieren, über das sich Feineinstellungen für das Hinzufügen der einzelnen Felder zum Fenster vornehmen lassen.

Die für unseren Fall wichtigste Feineinstellung, wenn wir in einem Rutsch alle Felder in das Fenster **MainWindows.xaml** ziehen wollen, ist die Auswahl der zu erstellenden Steuerelemente (siehe Bild 8). Mit **DataGrid** erstellen Sie ein DataGrid mit allen gewünschten Feldern. **ComboBox** erstellt ein Kombinationsfeld, was in diesem Szenario jedoch Nacharbeiten erfordert. **Liste** erstellt ein Listenfeld mit den Daten der Datenquelle. Und **Detail** erstellt schließlich die gewünschte Ansicht – daher wählen wir diesen Eintrag aus, was dazu führt, dass links neben dem Eintrag **Kunde** ein entsprechendes Icon erscheint. Nun ziehen Sie einfach den kompletten **Kunde**-Knoten in den Entwurf des Fensters **MainWindow.xaml**. Das Ergebnis sieht wie in Bild 9 aus.

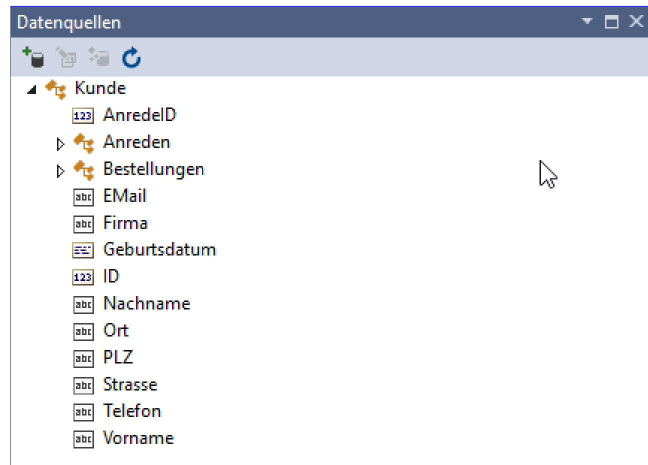


Bild 5: Nun steht eine Datenquelle zur Verfügung.

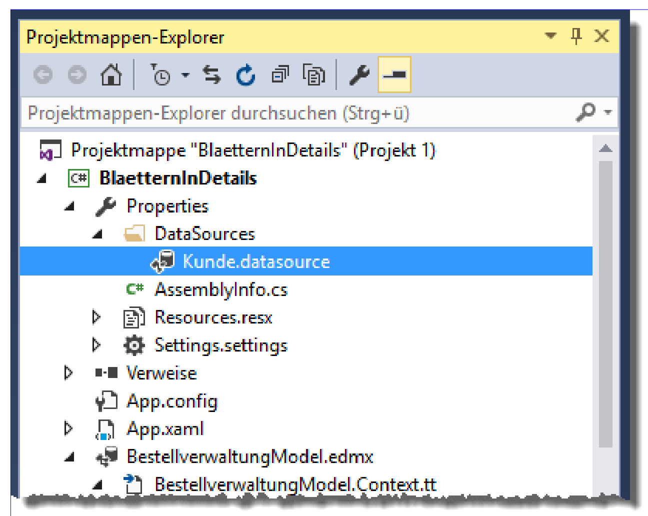


Bild 6: Die Datenquelle im Projektmappen-Explorer

Bild 7: Geänderte Ansicht bei gleichzeitiger Anzeige einer **.xaml**-Datei

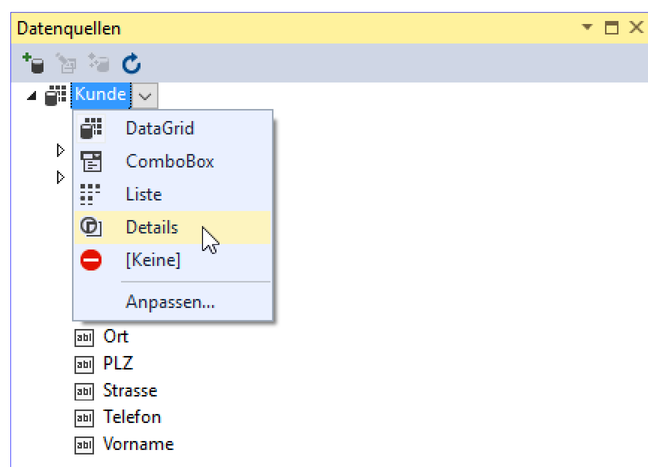


Bild 8: Festlegen der Ansicht

Wenn wir nun das Projekt starten, erhalten wir zwar das Fenster mit den Feldern in der noch unsortieren Anordnung und ohne Kombinationsfelder etwa für das Feld **AnredeID**, aber es zeigt keine Daten an (siehe Bild 10). Was nun?

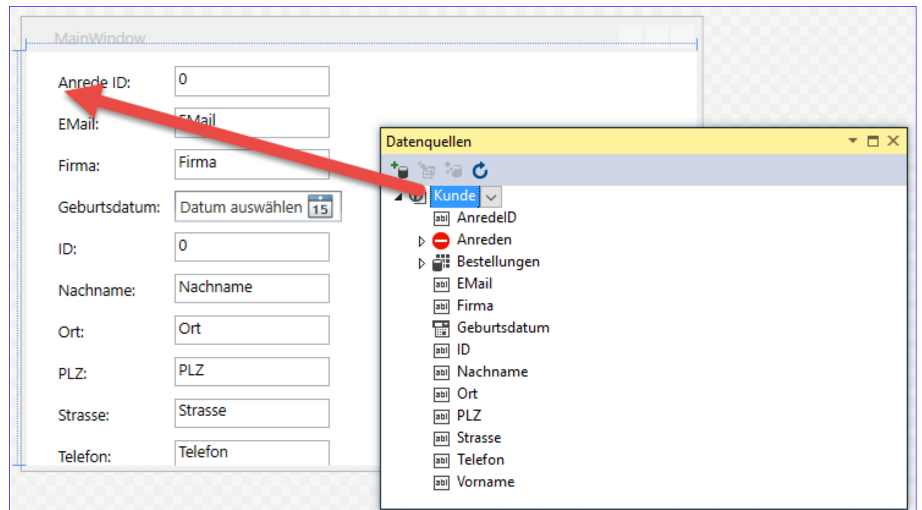


Bild 9: Detailfenster per Drag and Drop

Schauen wir uns zunächst an, was wir durch das Hinzufügen der Steuerelemente per Drag and Drop überhaupt für Änderungen durchgeführt haben – beziehungsweise welche Änderungen automatisch erledigt wurden. Dass für jedes Feld der Datenquelle jeweils ein Steuerelement zum Fenster hinzugefügt wurde, haben Sie ja bereits gesehen. Dementsprechend hat auch eine Menge Code Einzug in das **.xaml**-Dokument **MainWindow.xaml** genommen.

Im oberen Teil wurde ein Eigenschaftselement namens **Windows.Resources** mit folgendem Inhalt eingefügt:

```
<Window.Resources>
    <CollectionViewSource x:Key="kundeViewSource" d:DesignSource="{d:DesignInstance {x:Type local:Kunde}, CreateList=True}"/>
</Window.Resources>
```

Das **CollectionViewSource**-Element ist eine Verbindung zwischen der Benutzeroberfläche und den Daten. Es erhält automatisch über das Attribut **key** einen Schlüssel namens **kundeViewSource**, damit man es später in den Elementen der Benutzeroberfläche referenzieren kann. Der hintere Teil enthält Informationen über das Aussehen der gebundenen Steuerelemente zur Entwurfszeit.

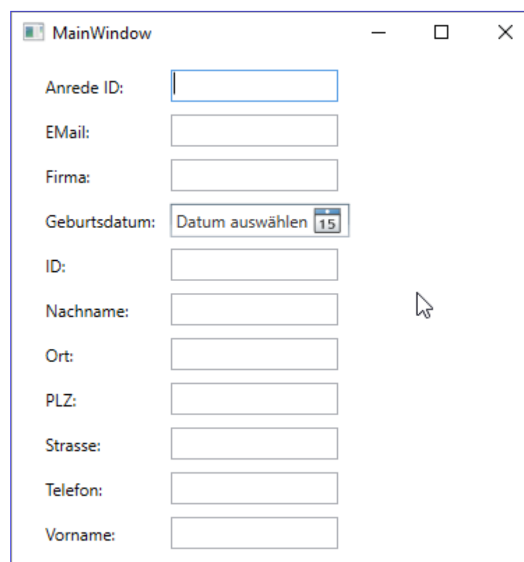


Bild 10: Detailfenster noch ohne Daten

Wenn Sie den Teil **d:DesignSource="{d:DesignInstance {x:Type local:Kunde}, CreateList=True}"** entfernen, werden Sie den Unterschied erkennen.

Schauen wir weiter nach unten, sehen wir ein **Grid**-Element, dessen wichtigste Eigenschaft in diesem Zusammenhang das Attribut **DataContext** ist.

Dieses enthält den Wert **{StaticResource kundeViewSource}** und legt somit fest, dass es die Daten aus der weiter oben festgelegten statischen Ressource beziehen soll (siehe Listing 1). Dies ist die Voraussetzung dafür, dass wir für die im **Grid**-Element enthaltenen Steuerelemente Bindungen festlegen können, über die sie dann auf die Daten der **CollectionViewSource** zugreifen können. Dies gilt für alle Elemente, die dem **Grid**-Element untergeordnet sind.

Im Grid folgen nun einige Zeilen Code, die das Aussehen des Grids festlegen, also wie viele Zeilen und Spalten es enthält und welche Breite beziehungsweise Höhe diese haben (hier in der Regel auf den Wert **Auto** eingestellt).

```
<Grid x:Name="grid1" DataContext="{StaticResource kundeViewSource}" HorizontalAlignment="Left" Margin="204,55,0,0"
        VerticalAlignment="Top">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        ...
    </Grid.RowDefinitions>
    <Label Content="Anrede ID:" Grid.Column="0" HorizontalAlignment="Left" Margin="3" Grid.Row="0"
        VerticalAlignment="Center"/>
    <TextBox x:Name="anredeIDTextBox" Grid.Column="1" HorizontalAlignment="Left" Height="23" Margin="3" Grid.Row="0"
        Text="{Binding AnredeID, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}"
        VerticalAlignment="Center" Width="120"/>
    <Label Content="EMail:" ... />
        <TextBox x:Name="eMailTextBox" ...
            Text="{Binding EMail, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Firma:" ... />
    <TextBox x:Name="firmaTextBox"
        Text="{Binding Firma, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Geburtsdatum:" ... />
    <DatePicker x:Name="geburtsdatumDatePicker" ...
        SelectedDate="{Binding Geburtsdatum, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="ID:" Grid.Column="0" HorizontalAlignment="Left" Margin="3" Grid.Row="4" VerticalAlignment="Center"/>
    <TextBox x:Name="iDTextBox" ...
        Text="{Binding ID, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Nachname:" ... />
    <TextBox x:Name="nachnameTextBox" ...
        Text="{Binding Nachname, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Ort:" ... />
    <TextBox x:Name="ortTextBox" ...
    Text="{Binding Ort, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="PLZ:" ... />
    <TextBox x:Name="pLZTextBox" ...
        Text="{Binding PLZ, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Strasse:" ... />
    <TextBox x:Name="strasseTextBox" ...
        Text="{Binding Strasse, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Telefon:" ... />
    <TextBox x:Name="telefonTextBox" ...
        Grid.Row="9" Text="{Binding Telefon, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Vorname:" ... />
    <TextBox x:Name="vornameTextBox" ...
    Text="{Binding Vorname, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" />
</Grid>
```

Listing 1: Die per Drag and Drop erstellten Steuerelemente samt Datenbindung

Dies erfolgt in den Eigenschaftselementen **Grid.ColumnDefinitions** und **Grid.RowDefinitions**, die wiederum entsprechende **ColumnDefinition**- und **RowDefinition**-Elemente enthalten.

Für jedes Feld der Datenquelle hat Visual Studio jeweils ein **Label**-Element angelegt sowie ein entsprechendes Steuerelement. In diesem Fall handelt es sich nur um **TextBox**- und **DatePicker**-Steuerelemente.

Die **TextBox**-Steuerelemente und auch das **DatePicker**-Steuerelement wurden mit jeweils einem Attribut ausgestattet, das sich auf die Datenbindung bezieht. Für das Feld **EMail** lautet der Code des Attributs **Text** beispielsweise so:

```
Text="{Binding EMail, Mode=TwoWay,
NotifyOnValidationError=true, ValidatesOnExceptions=true}"
```

Die Angabe von **EMail** direkt hinter dem **Binding**-Schlüsselwort heißt, dass das Steuerelement als **Text** den Inhalt des Feldes **EMail** der Datenquelle anzeigen soll. **Mode=TwoWay**: legt fest, dass Änderungen an den Daten an das zugrunde liegende Objekt zurückgegeben werden. Würde hier der Wert **OneWay** stehen, würden sich Änderungen nicht auf die Datenquelle auswirken.

Das hinter **Binding** angegebene Feld bezieht sich auf den **DataContext** im übergeordneten Objekt, der wiederum die statische Ressource **kundeViewSource** referenziert. Diese ist wiederum im Element **Window.Resources** als **CollectionViewSource**-Element definiert, wie wir ja weiter oben bereits gesehen haben.

Daten bereitstellen

Nun wissen wir aber noch nicht, woher diese **CollectionViewSource**, deren **Key**-Attribut den Wert **kundeViewSource** hat, die Daten bezieht. Und genau das erledigen wir in der Code behind-Klasse. Beziehungsweise sollte das beim Drag and Drop der Entität in das Fenster erfolgt sein – was aber nicht wie gewünscht geschehen ist, denn sonst hätten wir ja weiter oben beim ersten Start der Anwendung keine leeren Steuerelemente vorgefunden.

Schauen wir uns also an, was uns der Assistent beschert hat. Das ist gar nicht so viel, wie Listing 2 zeigt. Zunächst einmal: Damit die hier definierte Ereignismethode **Window_Loaded** ausgelöst wird, hat Visual Studio das Attribut **Loaded** mit dem Namen der Methode für das **Window**-Element hinterlegt:

```
<Window x:Class="BlaetternInDetails.MainWindow"
... Title="MainWindow" Height="350" Width="632"
Loaded="Window_Loaded">
```

Die Ereignismethode selbst instanziert dann lediglich ein Objekt namens **kundeViewSource** mit dem Typ **CollectionViewSource** und verbindet es mit der für die Benutzeroberfläche festgelegten Ressource, indem sie diese mit **FindResource** ermittelt und nach einer Konvertierung in den richtigen Datentyp zuweist.

Darunter finden wir dann zwei auskommentierte Zeilen, von denen die zweite der Ansatzpunkt zum Füllen der Benutzeroberfläche mit den gewünschten Daten ist. Wir müssen nur noch **[generische Datenquelle]** durch den richtigen Ausdruck ersetzen.

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    System.Windows.Data.CollectionViewSource kundeViewSource = ((System.Windows.Data.CollectionViewSource)
                                                    (this.FindResource("kundeViewSource")));
    // Laden Sie Daten durch Festlegen der CollectionViewSource.Source-Eigenschaft:
    // kundeViewSource.Source = [generische Datenquelle]
}
```

Listing 2: Automatisch generierter Code

EDM: Lookup-Kombinationsfelder

Im Artikel »Einfaches Detailfenster« sehen wir uns an, wie Sie per Drag and Drop die Daten einer Kundentabelle über ein Entity Data Model zu einem Fenster hinzufügen und dieses mit Navigationsschaltflächen ausstatten. Eine Kleinigkeit fehlt dort noch: Die Umwandlung des Textfeldes mit den Anreden in ein entsprechendes Kombinationsfeld, mit dem sich die Anreden auswählen lassen. Dies reichen wir im vorliegenden Artikel nach.

In dem genannten Artikel haben wir ein Entity Data Model auf Basis unserer Beispieldatenbank **Bestellverwaltung** erstellt, das unter anderem die Entitäten **Kunde** und **Anrede** enthält. Per Drag and Drop haben wir aus dem Datenquellen-Fenster alle Felder der Entität **Kunde** in Form einer Detailansicht in das Fenster gezogen. Das Fenster haben wir außerdem mit Navigationsschaltflächen und mit Schaltflächen zum Erstellen neuer Datensätze und zum Speichern geänderter und neuer Datensätze ergänzt. Nun wollen wir noch aus dem Textfeld, das bisher statt einer Anrede nur den in der Entität **Kunde** gespeicherten Zahlenwert des Fremdschlüsselfeldes der entsprechenden Tabelle namens **Kunden** anzeigt, ein Kombinationsfeld machen. Dieses soll zwar den Wert des Fremdschlüsselfeldes als gebundenen Wert enthalten, aber den Wert des entsprechenden Feldes der Lookup-Tabelle anzeigen und die übrigen Datensätze zur Auswahl anbieten.

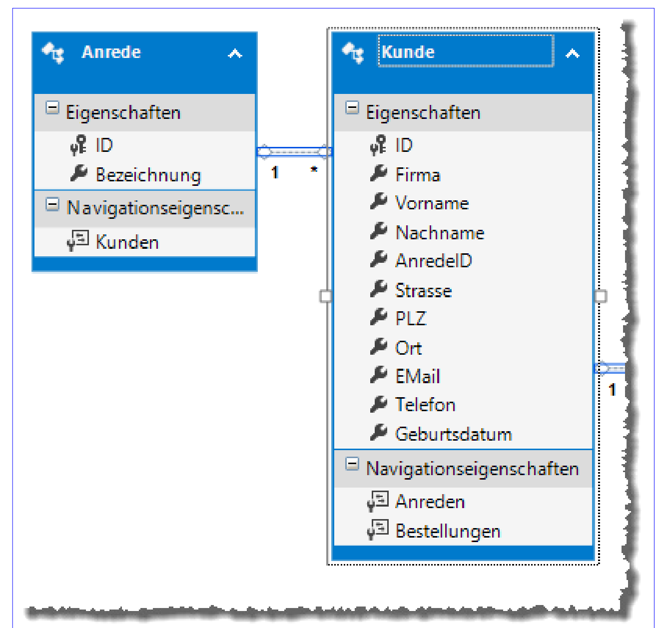


Bild 1: Betroffene Tabellen des Entity Data Models

Die beiden relevanten Tabellen des Entity Data Models sehen Sie in Bild 1. Das Fenster selbst enthält die Felder der Entität **Kunde**. Wir wollen das Steuerelement, das beim Einfügen der Felder per Drag and Drop aus dem Datenquellen-Bereich für das Feld **AnredeID** eingefügt wurde, entfernen und dafür ein Kombinationsfeld hinterlegen.

Wenn wir die Felder wie im Artikel **EDM: Einfaches Detailfenster** einfügen, können wir zumindest

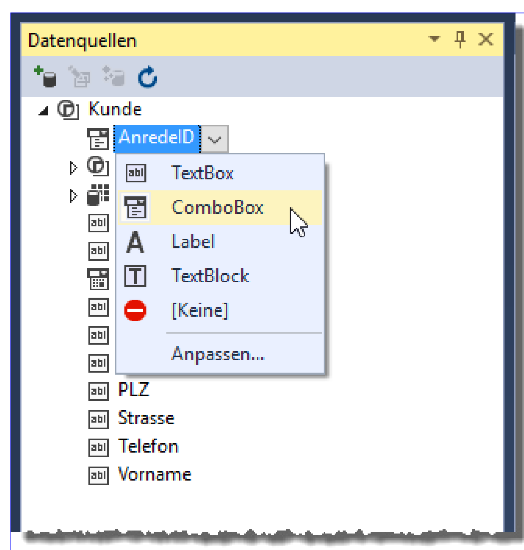


Bild 2: Vorkonfigurieren des Feldes **AnredeID** als Kombinationsfeld

schon einmal den Steuerelementtyp über das entsprechende Kombinationsfeld im Bereich **Datenquellen** einstellen. Dazu wählen Sie dort den Eintrag **ComboBox** aus (siehe Bild 2). Nach dem Sortieren der Steuerelemente wie im oben genannten Artikel beschrieben, fügen Sie noch den Bereich mit den Navigationsschaltflächen hinzu – ebenfalls nach der Vorgehensweise aus dem oben genannten Artikel. Natürlich können Sie auch die

dortigen Beispiele weiterverwenden (die Beispielfenster finden Sie ohnehin in einem einzigen Beispielprojekt).

Erster Versuch

Nachdem wir das Fenster soweit eingerichtet haben, dass es die Daten der zugrunde liegenden Tabelle anzeigt, starten wir das Projekt und schauen uns an, was das Kombinationsfeld uns liefert. Wie Bild 3 zeigt, enthält es wohl die gleiche Datenquelle wie das Fenster selbst und zeigt die Werte des Feldes **AnredeID** für alle Datensätze der Tabelle **Kunden** an.

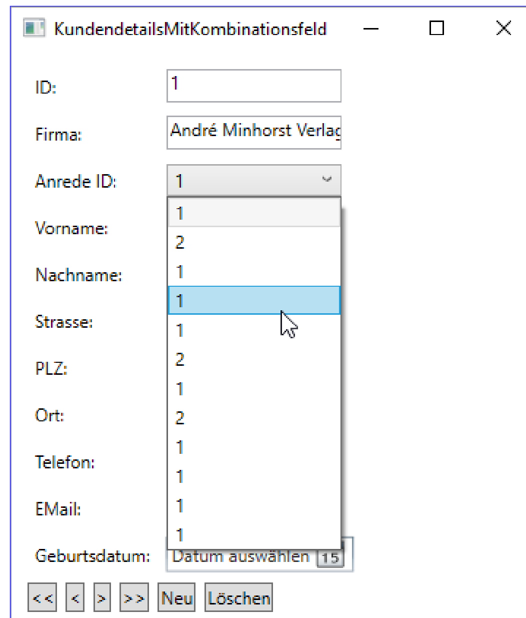


Bild 3: Das Kombinationfeld zeigt lediglich alle Werte, die für die **Kunde**-Entitäten vergeben wurden.

hat (siehe Listing 1). Hier kommen nun drei Anweisungen hinzu: Die erste weist der Variablen **anredenViewSource** die gleichnamige Ressource hinzu, die wir gleich noch im XAML-Code anlegen müssen. Die zweite lädt mit der **Load**-Methode des **DBSet**-Objekts **Anreden** die Daten der Tabelle **Anreden** aus der Datenbank in das Entity Data Model. Die dritte weist der **Source**-Eigenschaft von **anredenViewSource** die aktuell im Speicher befindlichen Daten aus **DBContext.Anreden** über die **Local**-Eigenschaft zu.

Eigene Datenquelle für das Kombinationsfeld

Der nächste Schritt ist also zunächst, dem Fenster eine eigene Datenquelle für die Einträge der Tabelle **Anrede**, welche das Kombinationsfeld anzeigen soll, hinzuzufügen. Analog zur Variablen **kundeViewSource** des Typs **CollectionViewSource** legen wir eine weitere **CollectionViewSource** an, diesmal namens **anredeViewSource** – dies alles im Kopf der partiellen Klasse im Code behind-Modul des Fensters:

```
CollectionViewSource anredenViewSource;
```

Einige weitere notwendige Schritte nehmen wir in der Ereignismethode **Window_Loaded** vor, die auch schon das **CollectionViewSource**-Objekt **kundeViewSource** gefüllt

XAML-Code anpassen

Nun fehlen noch die Anpassungen im XAML-Code des Fensters. Als Erstes kümmern wir uns um die Ressource.

Wir haben zwar im Code behind-Modul eingestellt, dass eine CollectionViewSource-Klasse erzeugt und über das Entity Data Model mit den Daten der Tabelle **Anreden** gefüllt wird und angegeben, dass diese mit einer Ressource verknüpft wird, aber diese Ressource ist noch nicht vorhanden.

Diese fügen wir wie in Listing 2 zum Eigenschaftselement **Window.Resources** hinzu, und zwar als **CollectionViewSource**-Element wie das bereits vorhandene Element mit dem Key **kundeViewSource**. Hier verwenden wir für das Attribut **local** jedoch die Entität **Anrede**.

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    anredenViewSource = ((CollectionViewSource)(this.FindResource("anredenViewSource")));
    kundeViewSource = ((CollectionViewSource)(this.FindResource("kundeViewSource")));
    DBContext.Kunden.Load();
    kundeViewSource.Source = DBContext.Kunden.Local;
    DBContext.Anreden.Load();
    anredenViewSource.Source = DBContext.Anreden.Local;
}
```

Listing 1: Verbinden und füllen des zweiten **CollectionViewSource**-Objekts **anredeViewSource**

EDM: Kombinationsfelder erweitern

Im Artikel »Lookup-Kombinationsfelder« zeigen wir, wie Sie eine per Drag and Drop aus dem Datenquellen-Bereich hinzugefügte Entität mit einem Kombinationsfeld versehen, mit dem Sie die Daten einer Lookup-Tabelle auswählen können. Dies wollen wir nun noch erweitern, und zwar um die Möglichkeit der Eingabe neuer Werte für die Lookup-Tabelle direkt über das Kombinationsfeld. Dazu nutzen wir ein neues Fenster, das die Daten der Tabelle Artikel anzeigt – und deren Kategorie per Kombinationsfeld auswählt.

Neuen Eintrag hinzufügen

Unter Access war es relativ einfach, ein Kombinationsfeld mit einer Funktion auszustatten, die dafür sorgte, dass neu eingegebene Einträge, die noch nicht in der Datensatzherkunft vorhanden waren, zur zugrunde liegenden Tabelle hinzugefügt werden und dann direkt im Kombinationsfeld angezeigt werden konnten. Wir wollen uns anschauen, wie dies unter WPF unter Verwendung unseres Entity Data Models auf Basis der Tabellen der Beispieldatenbank **Bestellverwaltung** funktioniert. Ein passendes Beispiel sind die Tabellen **Artikel** und **Kategorien**. Die Tabelle **Artikel** enthält ein Fremdschlüsselfeld, und zwar zur Auswahl der Kategorie des Artikels. Die Kategorien stammen wiederum aus der Tabelle **Kategorien**.

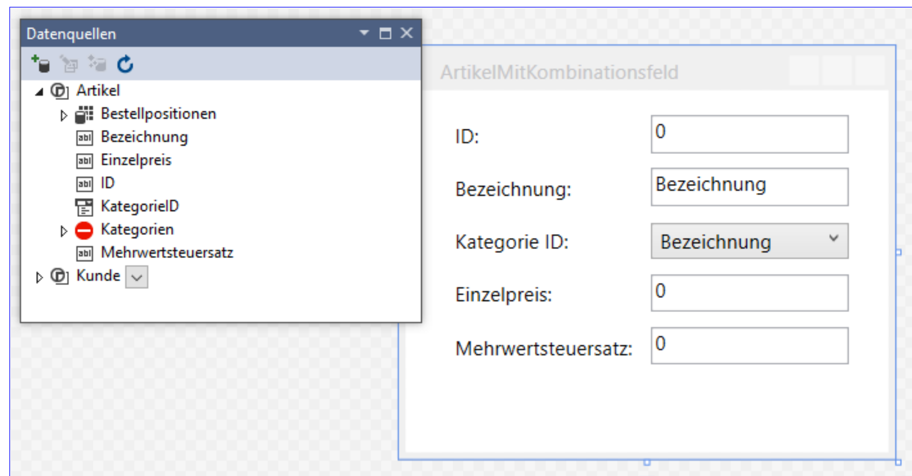


Bild 1: Erstellen des Beispielfensters

Für das neue Beispiel fügen wir der Datenquelle des Projekts die Entität **Artikel** hinzu. Ein neues Fenster soll die Daten der gleichnamigen Tabelle anzeigen. Dazu ändern wir den Typ der Entität **Artikel** per Kombinationsfeld auf **Details** und den Typ des Steuerelements **KategorieID** auf **ComboBox**. Dann ziehen Sie die Entität **Artikel** in das neue Fenster **ArtikelMitKombinationsfeld** und sortieren die enthaltenen Steuerelemente so wie in Bild 1 um. Damit die Daten auch im Fenster angezeigt werden, sind in aller Kürze die folgenden Schritte nötig:

- Als Erstes ändern Sie den Entwurf im XAML-Code wie in Listing 1 dargestellt. Hier kommen vor allem die neue **CollectionViewSource** für die Kategorien und die Änderungen am **ComboBox**-Element zum Tragen.

- Dann erweitern Sie den Code des Code behind-Moduls wie in Listing 2. Hier ist die Deklaration der beiden **CollectionViewSource**-Elemente und des Datenbankkontexts sowie deren Füllung im Ereignis **Window_Loaded**, das übrigens auch im XAML-Code noch hinzugefügt werden muss. Details hierzu erfahren Sie im Artikel **EDM: Einfaches Detailfenster**.

Auf die Beschreibung der Navigationsschaltflächen verzichten wir an dieser Stelle – wie dies funktioniert, haben wir ebenfalls im Artikel **EDM: Einfaches Detailfenster** erläutert.

ComboBox zum Hinzufügen von Artikeln?

Nun schauen wir uns an, wie wir dem Kombinationsfeld einen neuen Eintrag unterjubeln können, der dann optimaler-

```
<Window x:Class="BlaetternInDetails.ArtikelMitKombinationsfeld" ...
        Title="ArtikelMitKombinationsfeld" Height="250" Width="300" Loaded="Window_Loaded">
    <Window.Resources>
        <CollectionViewSource x:Key="artikelViewSource"
            d:DesignSource="{d:DesignInstance {x:Type local:Artikel}, CreateList=True}"/>
        <CollectionViewSource x:Key="kategorienViewSource"
            d:DesignSource="{d:DesignInstance {x:Type local:Kategorie}, CreateList=True}"/>
    </Window.Resources>
    <Grid DataContext="{StaticResource artikelViewSource}">
        <Grid x:Name="grid1" HorizontalAlignment="Left" Margin="22,11,0,0" VerticalAlignment="Top">
            ...
            <ComboBox x:Name="kategorieIDComboBox" ItemsSource="{Binding Source={StaticResource kategorienViewSource}}"
                DisplayMemberPath="Bezeichnung" SelectedValuePath="ID" SelectedValue="{Binding Path=KategorieID}">
            </ComboBox>
            ...
        </Grid>
    </Grid>
</Window>
```

Listing 1: Notwendige Änderungen am Fenster zur Anzeige von Artikeln mit Kategorien im Kombinationsfeld

weise direkt in der zugrunde liegenden Tabelle **Kategorien** gespeichert wird. Wenn wir die Anwendung starten und in unserem Beispielfenster versuchen, wie etwa unter Access eine neue Kategorie einzugeben, stellen wir allerdings fest, dass ein **ComboBox**-Steuerelement wohl nicht zum Hinzufügen von Daten gemacht ist (siehe Bild 2). Was nun? Ein anderes Steuerelement wählen? Oder gibt es Einstellungen für das **ComboBox**-Steuerelement, die das Hinzufügen von eigenen Texten ermöglichen? Schauen wir uns doch zunächst eine einfache und auch für den Benutzer intuitive Variante an, bei der wir neben dem Kombinationsfeld eine Schaltfläche zum Hinzufügen eines Eintrags platzieren. Anschließend betrachten wir die direkte Eingabe neuer Elemente direkt in das Kombinationsfeld.

```
public partial class ArtikelMitKombinationsfeld : Window {
    public ArtikelMitKombinationsfeld() {
        InitializeComponent();
    }
    CollectionViewSource artikelViewSource;
    CollectionViewSource kategorienViewSource;
    BestellverwaltungEntities DBContext = new BestellverwaltungEntities();
    private void Window_Loaded(object sender, RoutedEventArgs e) {
        artikelViewSource = ((CollectionViewSource)(this.FindResource("artikelViewSource")));
        DBContext.Artikel.Load();
        artikelViewSource.Source = DBContext.Artikel.Local;
        kategorienViewSource = ((CollectionViewSource)(this.FindResource("kategorienViewSource")));
        DBContext.Kategorien.Load();
        kategorienViewSource.Source = DBContext.Kategorien.Local;
    }
}
```

Listing 2: Anpassung der Code behind-Klasse des Fensters

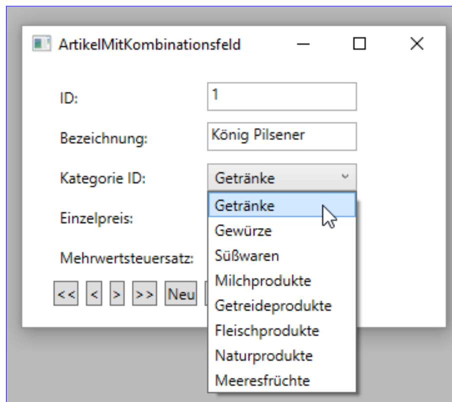


Bild 2: Das **ComboBox**-Steuerelement erlaubt nur die Auswahl, aber kein Anfügen von Einträgen.

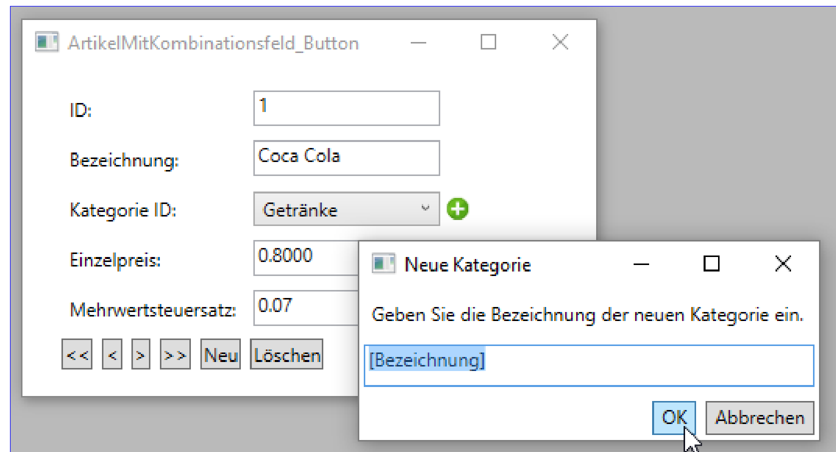


Bild 3: Kombinationsfeld mit Schaltfläche zum Hinzufügen neuer Einträge

Einträge hinzufügen per Schaltfläche

Das Anbieten einer Schaltfläche mit einem Plus-Symbol scheint die offensichtlichere und intuitivere Variante zu sein, um Elemente zu einem Lookup-Kombinationsfeld hinzuzufügen. Nach einem Klick auf diese Schaltfläche soll eine Inputbox erscheinen, die den neuen Eintrag ermittelt (siehe Bild 3). Nach der Eingabe soll die Datenherkunft des Kombinationsfeldes direkt aktualisiert und der neue Eintrag übernommen werden, sodass das Kombinationsfeld diesen direkt anzeigt.

Im Entwurf des Fensters **ArtikelMitKombinationsfeld_Button** sieht dies wie in Bild 4 aus. Hier haben wir neben dem Kombinationsfeld eine neue Schaltfläche hinzugefügt, die wir mit einigen neuen Attributwerten ausgestattet haben.

Die Definition sehen Sie in Listing 3. Wie Sie hier erkennen, haben wir nicht nur einfach eine Schaltfläche hinzugefügt, sondern diese noch zusammen mit dem **ComboBox**-Element in einem **StackPanel** zusammengefasst. Dieses weist für die Eigenschaft **Orientation** den Wert **Horizontal** auf und soll in der gleichen Grid-Zeile landen wie zuvor das Kombinationsfeld. Zu den Ressourcen des Grids haben wir außerdem noch einen Verweis auf die Datei **add.png** hinzugefügt, welche wir natürlich auch im Projekt gespeichert haben, und mit dem **Key**-Wert **add** versehen.

Die Schaltfläche **btnAdd** verwendet diese Ressource als Wert der **Content**-Eigenschaft (**Content="{StaticResource add}"**). Außerdem haben wir hier den Hintergrund mit der **Background**-Eigenschaft auf **Transparent** eingestellt und den Rahmen mit **BorderThickness="0"** ausgeblendet. Außerdem hinterlegen wir natürlich eine Ereignismethode für den Button, die wie in Listing 4 aussieht.

Die Methode speichert einen Verweis auf das Kombinationsfeld **cboKategorien** in der Variablen **cbo** des Typs **ComboBox**. Dann verwendet sie die im Artikel **Inputbox im Eigenbau** vorgestellte **InputBox**-Klasse, um eine Inputbox zur Eingabe des neuen Eintrags anzuzeigen, und speichert das Ergebnis in der Variablen **strNeu**. Als Standardwert übergeben wir der **Show**-Methode der Klasse dabei den Wert **[Bezeichnung]**. Die folgende **if**-Bedingung prüft dann, ob die Inputbox noch den Wert **[Bezeichnung]** enthält oder

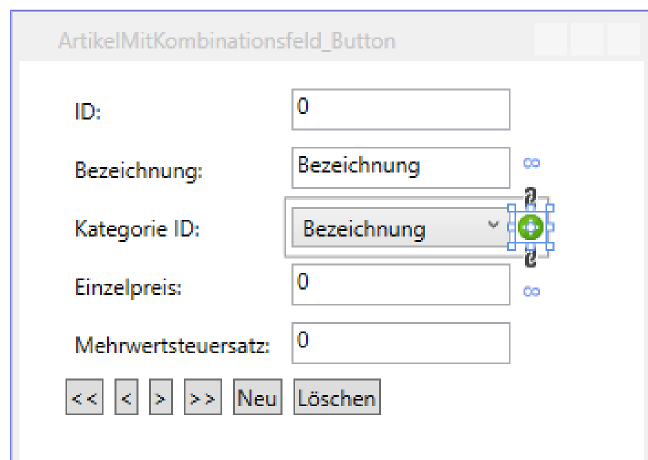


Bild 4: Schaltfläche zum Hinzufügen neuer Einträge

```
<Window x:Class="BlaetternInDetails.ArtikelMitKombinationsfeld_Button"
        Title="ArtikelMitKombinationsfeld_Button" Height="250" Width="350" Loaded="Window_Loaded" Closing="Window_Closing">
    ...
    <Grid DataContext="{StaticResource artikelViewSource}">
        <Grid.Resources>
            <Image x:Key="add" Source="add.png"></Image>
        </Grid.Resources>
        <Grid x:Name="grid1" HorizontalAlignment="Left" Margin="22,11,0,0" VerticalAlignment="Top" Width="282">
            ...
            <StackPanel Grid.Column="1" Grid.Row="2" Orientation="Horizontal">
                <ComboBox x:Name="cboKategorien"
                    HorizontalAlignment="Left" Height="Auto"
                    ItemsSource="{Binding Source={StaticResource kategorienViewSource}}"
                    DisplayMemberPath="Bezeichnung" SelectedValuePath="ID"
                    SelectedValue="{Binding Path=KategorieID}"
                    Margin="3" VerticalAlignment="Center" Width="120">
                </ComboBox>
                <Button x:Name="btnAdd" Content="{StaticResource add}" Grid.Row="2" Grid.Column="1"
                    Width="16" Height="16" HorizontalAlignment="Right" Background="Transparent"
                    BorderThickness="0" Click="btnAdd_Click"></Button>
            </StackPanel>
    ...
    </Grid>
</Window>
```

Listing 3: Definition des Fenstes mit einem Kombinationsfeld und einer Schaltfläche zum Hinzufügen von Einträgen

leer ist. Ersteres würde darauf hindeuten, dass der Benutzer entweder keinen neuen Eintrag eingegeben und die **OK**-Schaltfläche gedrückt hat.

Zweiteres heißt, dass der Benutzer entweder das Textfeld geleert und **OK** gedrückt hat oder dass er direkt die Schaltfläche **Abbrechen** betätigt hat. In all diesen Fällen gehen wir davon aus, dass der Benutzer keinen neuen Eintrag eingefügt hat, und brechen die Methode an dieser Stelle ab. Andernfalls legen wir ein neues Objekt des Typs **Kategorie** an und weisen seiner Eigenschaft **Bezeichnung** den durch den Benutzer eingegebenen Text aus der Variablen **strNeu** zu. Damit ist das neue **Kategorie**-Element bereits mit allen notwendigen Informationen ausgestattet.

```
private void btnAdd_Click(object sender, RoutedEventArgs e) {
    ComboBox cbo = this.cboKategorien;
    string strNeu = InputBoxCode.Show("Geben Sie die Bezeichnung der
        neuen Kategorie ein.", "Neue Kategorie", "[Bezeichnung]");
    if (!(strNeu=="") & !(strNeu=="[Bezeichnung]")) {
        Kategorie kategorie = new Kategorie();
        kategorie.Bezeichnung = strNeu;
        DBContext.Kategorien.Add(kategorie);
        DBContext.SaveChanges();
        DBContext.Kategorien.Load();
        kategorienViewSource.Source = DBContext.Kategorien.Local;
        cbo.SelectedValue = kategorie.ID;
    }
}
```

Listing 4: Ereignisprozedur beim Anklicken des **Hinzufügen**-Buttons des Kombinationsfeldes

Die folgende **Add**-Methode der **Kategorien**-Auflistung des **DBContext**-Objekts fügt die neue Kategorie zur Auflistung hinzu. Die **SaveChanges**-Methode überträgt die Ände-

EDM: 1:n-Beziehung als Parent-Child-Ansicht

Eine der unter Access ganz einfach abzubildenden Hierarchien zweier Tabellen ist die 1:n-Beziehung in einem Haupt- und einem Unterformular. Auch unter WPF mit den Daten aus einem Entity Data Model lässt sich dies gut abbilden. Wir schauen uns in diesem Artikel an, wie es funktioniert und was Sie beim Anzeigen der Daten beachten müssen. Als Beispiel nutzen wir die Kategorien der Datenbank im Hauptfenster und die der aktuellen Kategorie zugeordneten Artikel in einer Listenansicht.

Als Beispiele verwenden wir die beispielsweise im Artikel **EDM: Einfaches Detailfenster** vorgestellten Tabellen der Datenbank namens **Bestellverwaltung**, in diesem Fall speziell die Tabelle **Kategorien** für das Hauptfenster, die Tabelle **Artikel** für die Liste sowie die Tabelle **Lieferanten** als Lookuptabelle der Tabelle **Artikel** zur Auswahl der Lieferanten.

Hauptfenster anlegen

Für das Hauptfenster fügen Sie dem Projekt ein neues WPF-Fenster namens **KategorienUndArtikel** hinzu. Nun wollen wir in diesem die Kategorien in der Detailansicht anzeigen, also jeweils nur eine Kategorie gleichzeitig. Darüber hinaus benötigen wir dann natürlich noch Navigationsschaltflächen.

Da wir im Artikel **EDM: Einfaches Detailfenster** ja prinzipiell schon die gleiche Aufgabe erledigt haben, können wir uns dort bedienen (im Beispielprojekt finden Sie auch die in diesem Artikel behandelten Fenster). Als Erstes kümmern wir uns um die zugrunde liegenden Daten. Dazu fügen wir dem XAML-Code unterhalb des **Window**-Elements gleich die Referenz auf die Datenquelle hinzu:

```
<Window.Resources>
    <CollectionViewSource x:Key="vsKategorien"
        d:DesignSource="{d:DesignInstance {x:Type
        local:Kategorie}, CreateList=True}"/>
</Window.Resources>
```

Im Code behind-Modul legen wir die Variable für die **CollectionViewSource** an, diesmal im Gegensatz zu den vom Visual Studio automatisch per Drag and Drop hinzugefügten CollectionViewSources aus den anderen Beispielen mit dem Präfix **vs**, also unter dem Namen **vsKategorien**. Außerdem deklarieren und initialisieren wir auch hier den Datenbankkontext namens **DBContext** auf Basis der Klasse **BestellverwaltungEntities**:

```
CollectionViewSource vsKategorien;
BestellverwaltungEntities DBContext =
    new BestellverwaltungEntities();
```

Dann statten wir die Konstruktor-Methode, die beim Initialisieren des Fensters **KategorienUndArtikel** ausgelöst wird, mit ein paar zusätzlichen Anweisungen aus, die unsere **CollectionViewSource** mit den Daten der Entitätsliste **Kategorien** des **DBContext**-Objekts füllen:

```
public KategorienUndArtikel() {
    InitializeComponent();
    vsKategorien= ((CollectionViewSource)
        (this.FindResource("vsKategorien")));
    DBContext.Kategorien.Load();
    vsKategorien.Source = DBContext.Kategorien.Local;
}
```

Nun folgt die Definition der Steuerelemente im Fenster. Dazu fügen wir dem bereits vorhandenen **Grid**-Element die statische Ressource von oben als **DataContext** hinzu:

```
<Grid DataContext="{StaticResource vsKategorien}">
```

Danach definieren wir für das Grid zwei Spalten und vier Zeilen. Die ersten beiden Spalten sollen die Felder **ID** und **Bezeichnung** der Entität **Kategorie** anzeigen. Dazu stellen

```
<Grid DataContext="{StaticResource vsKategorien}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Label Content="ID:" Grid.Column="0" HorizontalAlignment="Left" Margin="3" VerticalAlignment="Center"/>
    <TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left" Height="22" Margin="3,4,0,6" Text="{Binding ID,
        Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" VerticalAlignment="Center" Width="120"/>
    <Label Content="Bezeichnung:" Grid.Column="0" HorizontalAlignment="Left" Margin="3" Grid.Row="1"
        VerticalAlignment="Center"/>
    <TextBox x:Name="txtBezeichnung" Grid.Column="1" HorizontalAlignment="Left" Height="23" Margin="3"
        Grid.Row="1" Text="{Binding Bezeichnung, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}"
        VerticalAlignment="Center" Width="120"/>
    ...
</Grid>
```

Listing 1: Definition der Steuerelemente des Hauptfensters

wir das Attribut **Text** auf einen entsprechenden **Binding**-Ausdruck ein, der die Eigenschaft **ID** referenziert. Gleiches erledigen wir für das Textfeld **txtBezeichnung**, welches auf die Eigenschaft **Bezeichnung** zeigt (siehe Listing 1).

Nun fehlen allerdings noch die Navigationsschaltflächen. Diese können Sie einfach in einem Stück aus dem XAML-Code des Fensters **ArtikelMitKombinationsfeld** als Grid in die dritte Zeile des Hauptgrids hineinkopieren. Am XAML-Code brauchen Sie keine Änderungen vorzunehmen. Das Fenster sieht nun im Entwurf wie in Bild 1 aus.

Code für Navigationsschaltflächen kopieren und anpassen

Danach kopieren Sie die Ereignismethoden aus der Code behind-Klasse des Fensters **ArtikelMitKombinationsfeld** in die Code behind-Klasse des neuen Fensters **KategorienUnd-Artikel**. Die Methoden passen Sie derart an, dass Sie den Namen der **CollectionViewSource** austauschen sowie an allen Stellen die Liste **Artikel** durch die Liste **Kategorien** und die Entität **Artikel** durch die Entität **Kategorie** ersetzen. Das

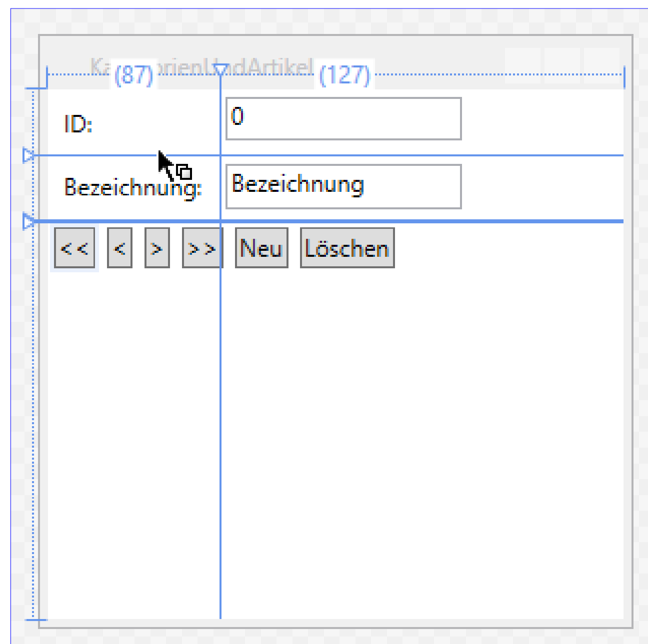


Bild 1: Aktuelle Entwurfsansicht des Fensters

Ergebnis sieht wie in Listing 2 aus. Danach können Sie das Projekt schon starten und die Datensätze des neuen Fensters mit den Navigationsschaltflächen durchlaufen.

```
private void btnErster_Click(object sender, RoutedEventArgs e) {
    Speichern();
    if (vsKategorien.View.CurrentPosition > 0) {
        vsKategorien.View.MoveCurrentToFirst();
    }
}
private void btnVorheriger_Click(object sender, RoutedEventArgs e) {
    Speichern();
    if (vsKategorien.View.CurrentPosition > 0) {
        vsKategorien.View.MoveCurrentToPrevious();
    }
}
private void btnNaechster_Click(object sender, RoutedEventArgs e) {
    Speichern();
    if (vsKategorien.View.CurrentPosition < ((CollectionView)vsKategorien.View).Count) {
        vsKategorien.View.MoveCurrentToNext();
    }
}
private void btnLetzter_Click(object sender, RoutedEventArgs e) {
    Speichern();
    if (vsKategorien.View.CurrentPosition < ((CollectionView)vsKategorien.View).Count) {
        vsKategorien.View.MoveCurrentToLast();
    }
}
private void Speichern() {
    if (DBContext.ChangeTracker.HasChanges()) {
        DBContext.SaveChanges();
    }
}
private void btnNeu_Click(object sender, RoutedEventArgs e) {
    DBContext.Kategorien.Add(DBContext.Kategorien.Create());
    vsKategorien.View.MoveCurrentToLast();
}
private void btnLoeschen_Click(object sender, RoutedEventArgs e) {
    Kategorie kategorie = (Kategorie)vsKategorien.View.CurrentItem;
    if (kategorie != null) {
        DBContext.Kategorien.Remove(kategorie);
        Speichern();
    }
}
```

Listing 2: Ereignismethoden für die Navigationsschaltflächen

Artikel-DataGrid hinzufügen

Nun fehlen aber natürlich noch die Artikel zu der jeweils angezeigten Kategorie. Wie geht man dies an, wenn man keine Anleitung hat? Tendenziell würde man vermutlich davon ausgehen, dass wie etwa für das Füllen eines Kombinationsfeldes eine weitere **CollectionViewSource** nötig ist, um die Daten für die per 1:n-Beziehung verknüpften Daten der Tabelle **Artikel** in einem **DataGrid** oder in einem ähnlichen Steuerelement bereitzustellen. Schauen wir uns also an, wie das aussehen kann.

Tipps und Tricks

Kleine Tipps und Tricks rund um die Programmierung mit Visual Studio, die keinen eigenen Artikel benötigen, landen regelmäßig in dieser Kategorie.

Virtuelle Maschine auf die Schnelle

Normalerweise sträube ich mich immer, virtuelle Maschinen zu installieren. Obwohl man damit toll Sachen testen kann. Der Grund ist einfach: Ich brauche diese Maschinen immer kurzfristig und habe dann keine Zeit für die Installation! Und ich schätze, ich bin nicht der Einzige, der sich dann sagt: Das mache ich nächstes Mal, jetzt komme ich auch so klar. Nur, um dann ein paar Tage oder Wochen später festzustellen, dass man mal wieder eine frische virtuelle Maschine brauchen könnte, aber in der Zwischenzeit vergessen hat, eine solche einzurichten.

Nun gibt es allerdings Abhilfe, und zwar von Microsoft selbst: Dort bietet man nämlich fertige virtuelle Maschinen an, die genau zu unserem Anwendungszweck passen – nämlich mit Windows 10 und der aktuellen Version von Visual Studio. Zum Zeitpunkt der Erstellung dieses Artikels finden Sie einen entsprechenden Download etwa unter dem folgenden Link:

```
https://developer.microsoft.com/de-de/windows/downloads/virtual-machines
```

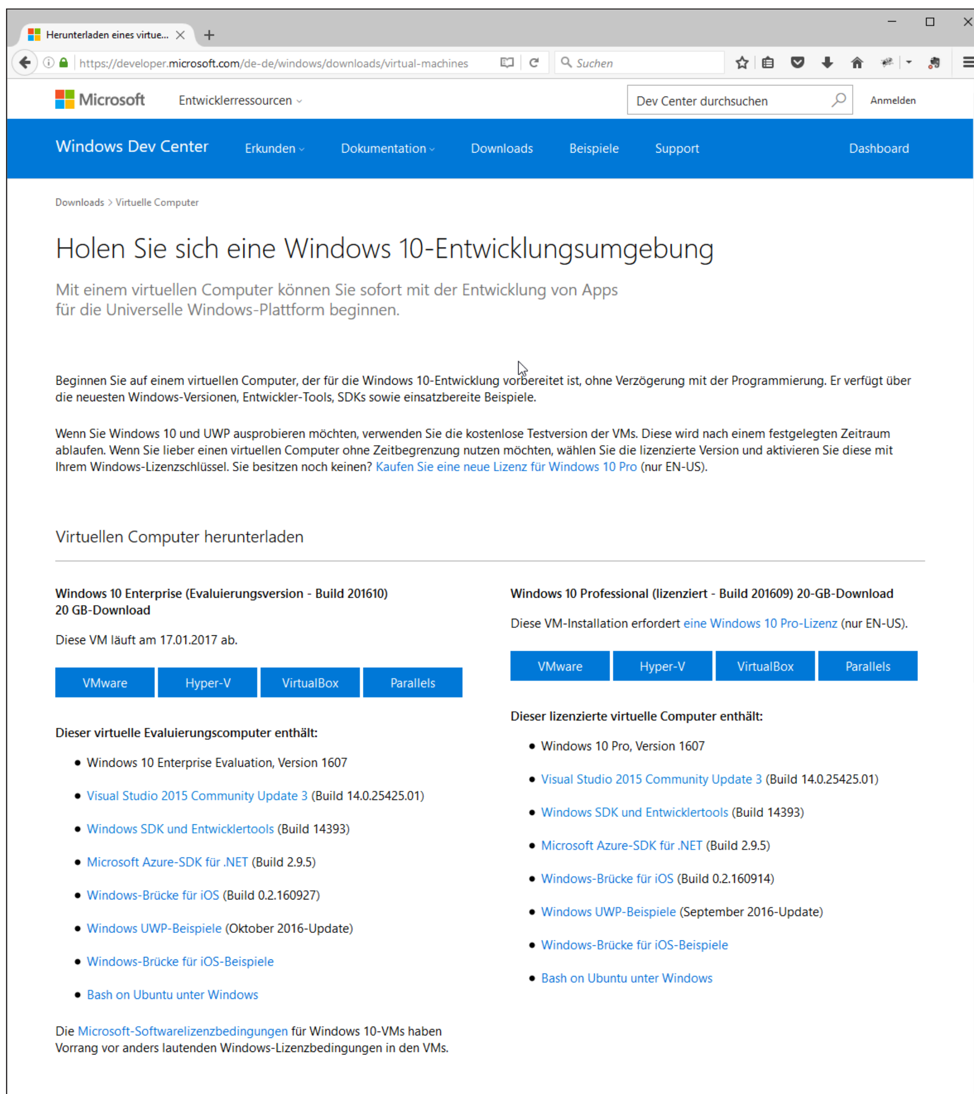


Bild 1: Download einer virtuellen Maschine für verschiedene Clients

Hier finden Sie virtuelle Maschinen für verschiedene Clients wie etwa **VMWare**, **Hyper-V**, **VirtualBox** oder **Parallels**. Die Downloads kommen in zeitlich beschränkten Versionen oder in unbegrenzten Versionen, für die Sie jedoch noch einen Registrierungsschlüssel benötigen (siehe Bild 1).

Nach dem Download etwa für VMWare extrahieren Sie die im .zip-Archiv enthaltenen Dateien und öffnen diese mit VMWare. Hier findet gegebenenfalls noch eine Konvertierung statt, was