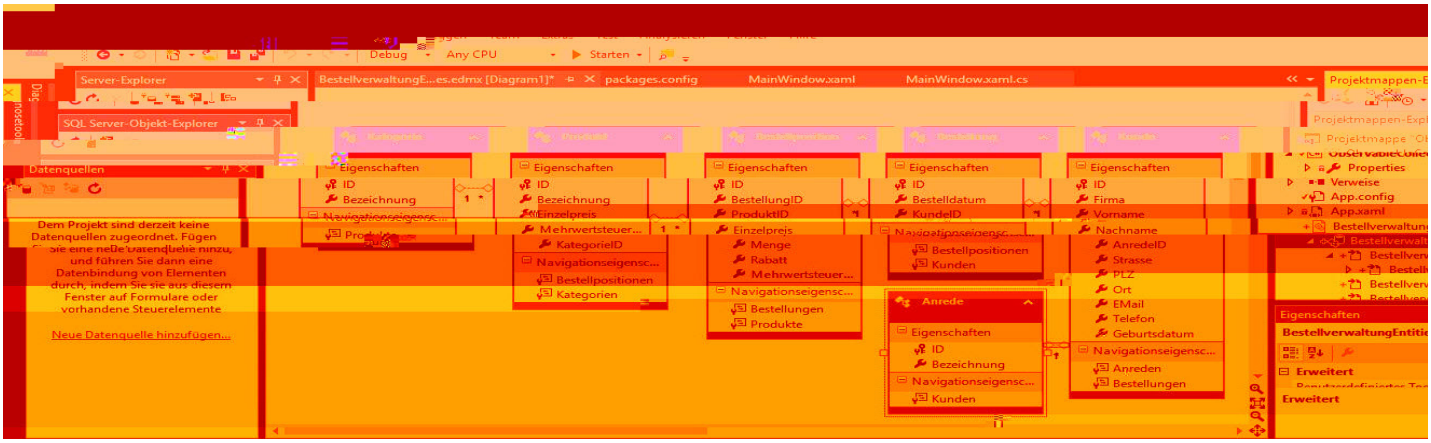
AUSGABE 03/2017

DATENBANK ENTWICKLER

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



TOP-THEMEN:

André Minhorst Verlag

Impressum

DATENBANKENTWICKLER
© 2017 André Minhorst Verlag
Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

Das ListBox-Steuerelement: Grundlagen

Wer unter Access/VBA gern das Listenfeld als Steuerelement genutzt hat, wird unter WPF ein ähnliches Steuerelement verwenden wollen. Ein einfach aufgebautes Steuerelement zur Anzeige von Texten in Listenform ist das ListBox-Steuerelement. Es lässt sich sehr leicht definieren und mit Daten füllen, sei es vordefiniert per XAML oder auch zur Laufzeit per Code. Dieser Artikel zeigt die Grundlagen zur Definition und Programmierung des ListBox-Steuerelements. Dabei erfahren Sie, wie Sie es mit einfachen Texten füllen, Einträge selektieren, die Selektion auslesen und auf Ereignisse wie die geänderte Auswahl oder einen Doppelklick reagieren können.

Programmiersprache

Die Beispiele in diesem Artikel basieren auf C#.

Hinweis

Um die verschiedenen Beispiele übersichtlich darzustellen, haben wir diese statt in verschiedenen **Window**-Elementen in **Page**-Elementen untergebracht, die dann über das Ribbon im Kopf des Fensters **MainWindow** per Mausklick auf den jeweiligen Ribbon-Button geöffnet werden können.

Die Definition des **RibbonButton**-Elements enthält dann im Attribut **tag** den Namen des zu öffnenden **Page**-Elements:

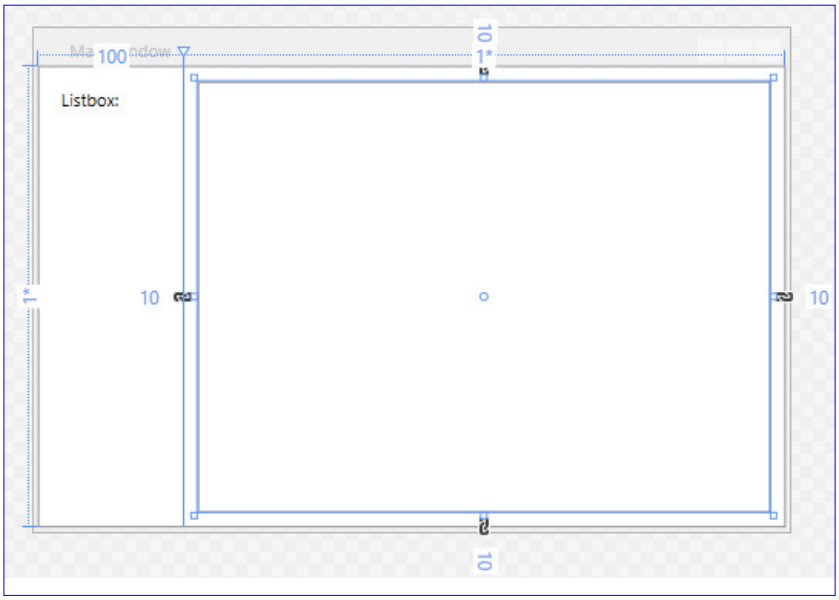


Bild 1: Entwurf unseres **ListBox**-Steuerelements

```
<RibbonButton Label="Einfache Einträge" Click="RibbonButton_Click" Tag="pgeEinfacheEintraege"></RibbonButton>
```

Die dort angegebene Ereignismethode **Ribbon_Button_Click** fragt den Wert des **Tag**-Attributs ab und öffnet das entsprechende **Page**-Element im **Frame**-Steuerelement:

```
private void RibbonButton_Click(object sender, RoutedEventArgs e) {
    RibbonButton button = (RibbonButton)sender;
    string pge = button.Tag.ToString();
    Workzone.Content = GetInstance("ListBox_Beispiele." + pge);
}
```

Listbox hinzufügen

Der erste und einfachste Schritt ist das Hinzufügen des ListBox-Elements: Dazu ziehen Sie es einfach aus der Toolbox in das WPF-Fenster und lassen es dort fallen.

Oder Sie machen es wie ein richtiger Programmierer und fügen ein **ListBox**-Element zur XAML-Definition des Fensters hinzu:

```
<ListBox x:Name="lstBeispiel" Height="100" Width="300" Margin="10,10" HorizontalAlignment="Left"
VerticalAlignment="Top"></ListBox>
```

Zusammen mit einem Label soll unsere erste Beispiel-ListBox dann wie folgt deklariert sein und im Entwurf wie in Bild 1 aussehen:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="100"></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Label Grid.Column="0" Margin="10,10,10,10">Listbox:</Label>
    <ListBox x:Name="lstBeispiel" Grid.Column="1" Margin="10,10,10,10"></ListBox>
</Grid>
```

Einträge vordefinieren

Vielleicht wissen Sie schon zur Entwurfszeit, welche Einträge die Liste anzeigen soll. Dann können Sie diese Elemente einfach per XAML festlegen. Der XAML-Code für die ListBox sieht dann wie folgt aus:

```
<ListBox x:Name="lstEintraege" Grid.Column="1" Margin="10,10,10,10">
    <ListBoxItem>Erster Eintrag</ListBoxItem>
    <ListBoxItem>Zweiter Eintrag</ListBoxItem>
    <ListBoxItem>Dritter Eintrag</ListBoxItem>
    <ListBoxItem>Vierter Eintrag</ListBoxItem>
</ListBox>
```

Diese Einträge werden dann in der ListBox wie in Bild 2 angezeigt.

Element per XAML vorselektieren

Wenn Sie die Elemente per XAML vordefinieren, können Sie auch einen der Einträge vorselektieren. Dazu fügen Sie dem betroffenen Element einfach das Attribut **IsSelected** hinzu und stellen es auf **true** ein:

```
<ListBoxItem IsSelected="true">Erster Eintrag</ListBoxItem>
```

ListBox-Element mit Texten füllen

Die erste Schaltfläche, die wir hinzufügen, soll einen Eintrag zum Listenfeld hinzufügen. Das erledigen wir mit der **Add**-Methode der **Items**-Auflistung der ListBox. Die Schaltfläche definieren wir wie folgt:

```
<Button x:Name="btnNamen" Grid.Column="0" Grid.
ColumnSpan="2" Grid.Row="1" Height="30" Mar-
gin="10,10,10,10" Click="btnNamen_Click">Namen</
Button>
```

Das **Click**-Ereignis soll diese Methode auslösen:

```
private void btnNeuerEintrag_Click(object sen-
der, RoutedEventArgs e) {
    lstEintraege.Items.Add("Fünfter Eintrag");
}
```

Das Ergebnis sieht schließlich wie in Bild 3 aus.

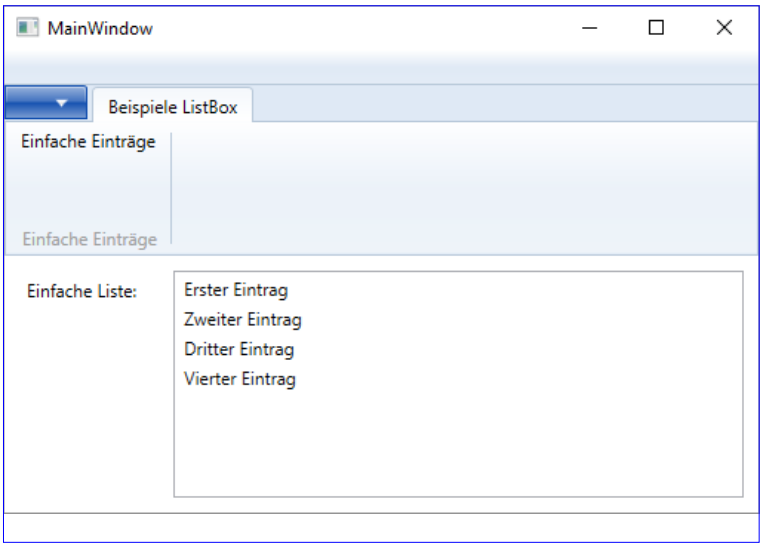


Bild 2: Feste Beispieleinträge in einem ListBox-Element

ListBox leeren

Wenn Sie alle Einträge aus der ListBox entfernen wollen, verwenden Sie die **Clear**-Methode der **Items**-Auflistung:

```
private void btnAlleLoeschen_Click(object sender, RoutedEventArgs e) {
    lstEintraege.Items.Clear();
}
```

Einfach- oder Mehrfachauswahl

Der Benutzer kann natürlich einen oder mehrere Einträge in der ListBox auswählen. Ob er einen oder mehrere auswählen kann, legen Sie mit der Eigenschaft **SelectionMode** fest. Diese kann die folgenden Werte annehmen:

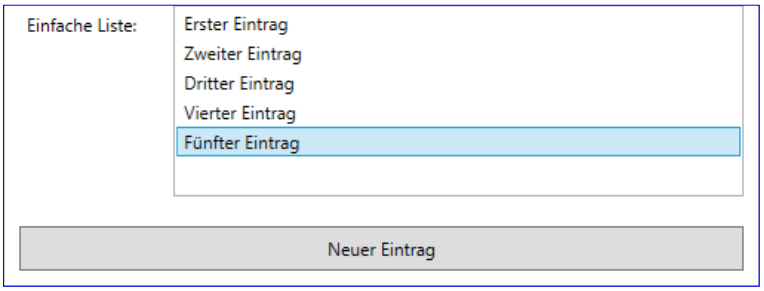


Bild 3: Einfügen einiger Einträge zu einem ListBox-Element

- **Single**: Standardeinstellung. Es kann nur ein Eintrag gleichzeitig ausgewählt sein. Allerdings kann der gewählte Eintrag auch nur noch abgewählt werden, indem man einen anderen Eintrag ausgewählt.

- **Multiple**: Der Benutzer kann einen, keinen oder mehrere Einträge auswählen. Zum Aus- oder Abwählen muss der jeweilige Eintrag angeklickt werden.

- **Extended**: Der Benutzer kann einen, keinen oder mehrere Einträge auswählen. Zum Aus- oder Abwählen gibt es mehrere Möglichkeiten: Ein einfacher Klick wählt einen Eintrag aus und alle anderen ab. Klicken bei gedrückter **Strg**-Taste arbeitet wie die Einstellung **Multiple**. Klicken bei gedrückter **Umschalt**-Taste wählt Bereiche aus.

Wir haben das in der Beispieldatenbank auf einer neuen Seite namens **pgeEintraegeAuswaehlen** abgebildet. Hier können Sie mit einer **ComboBox** einen der Modi für die Auswahl selektieren:

```
<ComboBox Grid.Column="1" Grid.Row="1" Margin="10,10,10,10" SelectionChanged="ComboBox_SelectionChanged">
    <ComboBoxItem IsSelected="true">Einfach</ComboBoxItem>
    <ComboBoxItem>Mehrfach</ComboBoxItem>
    <ComboBoxItem>Erweitert</ComboBoxItem>
</ComboBox>
```

Dies löst die folgende Ereignismethode aus:

```
private void ComboBox_SelectionChanged(object sender, SelectionChangedEventArgs e) {
    ComboBox cbo = (ComboBox)sender;
    switch (cbo.SelectedIndex) {
        case 0:
            lstEintraege.SelectionMode = SelectionMode.Single;
            break;
        case 1:
            lstEintraege.SelectionMode = SelectionMode.Multiple;
            break;
        case 2:
            lstEintraege.SelectionMode = SelectionMode.Extended;
            break;
        default:
            break;
    }
}
```

Das Ergebnis sehen Sie in Bild 4. Wenn Sie etwa den dritten Eintrag **Erweitert** auswählen, stellt dies die Option **Extended** ein. Der Benutzer kann dann sowohl bei gedrückter **Umschalt**-Taste ganze Bereiche auswählen, aber auch bei gedrückter **Strg**-Taste einzelne Einträge aus- oder abwählen – also ganz wie im Windows Explorer.

Gewählte Elemente abfragen

Wenn Sie den Benutzer Elemente auswählen lassen, wollen Sie auch abfragen, welche Elemente ausgewählt sind. Dazu gibt es verschiedene Möglichkeiten, die Sie auf der Seite **pgeAuswahlErmitteln** kennen lernen.

Im ersten Beispiel wollen wir den Index des ausgewählten Elements ermitteln. Dazu legen wir eine Schaltfläche mit der folgenden **Click**-Methode an:

```
private void btnIndexErmitteln_Click(object sender, RoutedEventArgs e) {
    int index = lstEintraege.SelectedIndex;
    MessageBox.Show("Gewählter Index: " + index);
}
```

Dies liefert eine Meldung wie in Bild 5. Wenn wir den dritten Eintrag auswählen, wird also der 0-basierte Indexwert **2** ausgegeben. Wenn Sie einen der Modi zur Mehrfachauswahl aktiviert haben, was keinen Sinn macht, wenn Sie nur einen Eintrag abfragen, wird übrigens der Index des von den aktuell markierten Einträgen zuerst markierten Eintrags zurückgegeben. Wenn gar kein Eintrag markiert ist, liefert die Eigenschaft **SelectedIndex** den Wert **-1**.

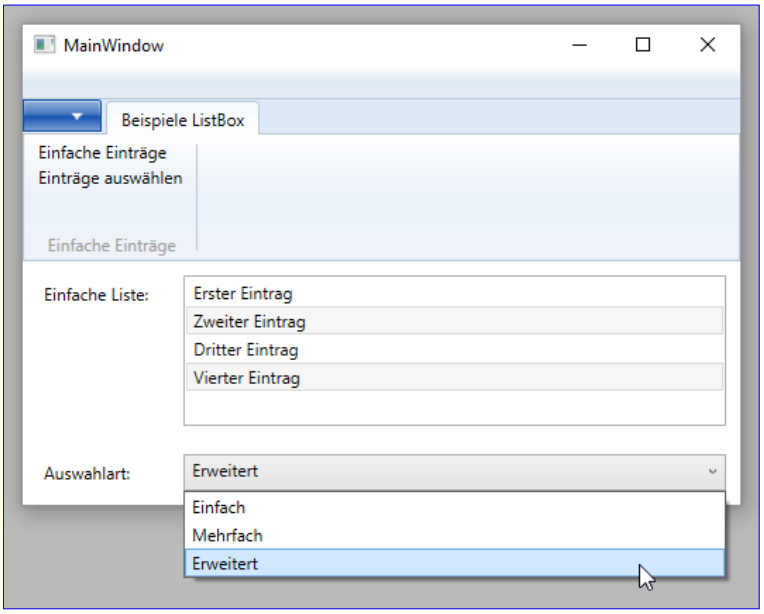


Bild 4: Auswahl der verschiedenen Auswahl-Modi

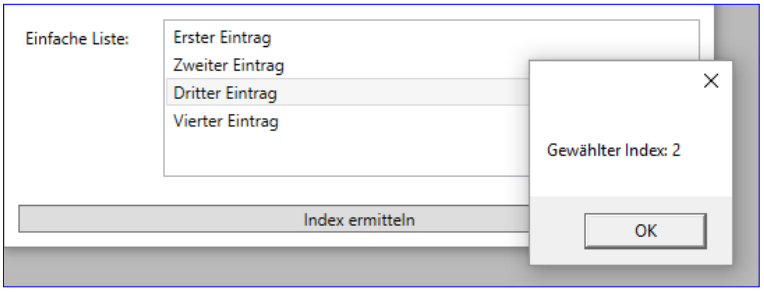


Bild 5: Anzeige des Index des gewählten Eintrags

Über den Index auf das Element zugreifen

Der Index allein hilft ja noch nicht weiter. Gegebenenfalls wollen Sie ja auch auf den angezeigten Inhalt zugreifen. Das können wir direkt erledigen, aber auch über den Index. Dies schauen wir uns zuerst an. Dazu deklarieren wir ein Element des Typs **ListBoxItem** und initialisieren es mit **null**. Dann ermitteln wir mit **SelectedIndex** zunächst den Index des ausgewählten Elements und speichern diesen in der Variablen **index**. Damit können wir nun über die **Items**-Auflistung des **ListBox**-Elements auf den markierten Eintrag zugreifen und diesen in einer Variablen des Typs **ListBoxItem** speichern. Diese gibt dann, sofern überhaupt ein Eintrag markiert ist, über die **Content**-Eigenschaft den angezeigten Inhalt preis, den wir noch in einen String konvertieren:

```
private void btnItemPerIndexErmitteln_Click(object sender, RoutedEventArgs e) {
    ListBoxItem listBoxItem = null;
    int index = -1;
    index = lstEintraege.SelectedIndex;
    if (index!=-1) {
```

EDM: Datenbindung mit dem ListBox-Steuerelement

Im Artikel Das ListBox-Steuerelement: Grundlagen haben wir die Grundlagen zum ListBox-Steuerelement erläutert. Heute gehen wir einen Schritt weiter und zeigen, wie Sie die Daten aus Objekten anzeigen, die beispielsweise aus einer Datenbank-Tabelle gewonnen oder anderweitig in die Objekte gefüllt werden. Dabei zeigen wir auch, wie Sie in der ListBox anspruchsvollere Daten wie etwa mehrspaltige Daten oder auch Bilder anzeigen.

Voraussetzung

In diesem Artikel wollen wir zeigen, wie Sie die Daten aus einer SQLite-Datenbank über das Entity Data Model in einem ListBox-Steuerelement anzeigen und die verschiedenen Möglichkeiten dazu diskutieren. Dazu greifen wir auf das gleiche Entity Data Model zu, welches wir auch in der Beispielanwendung namens **Bestellverwaltung** nutzen.

ListBox mit Elementen füllen

Wenn Sie nur eine einfache ListBox benötigen, die lediglich etwa die Namen von Kategorien anzeigt, dann durchlaufen Sie per Code eine Liste der Kategorien und fügen die Einträge zur ListBox hinzu. Das Ergebnis soll wie in Bild 1 aussehen (siehe Beispielprojekt, Seite **pgeListeEinlesenUndZuweisen.xaml**). Die Definition des **ListBox**-Elements sieht dann einfach wie folgt aus:

```
<ListBox x:Name="lstKunden" Grid.Column="1" Margin="10,10,10,10" />
```

In der Code behind-Datei benötigen wir lediglich die folgenden Codezeilen:

```
BestellverwaltungEntities dbContext;
public pgeListeEinlesenUndZuweisen() {
    InitializeComponent();
    dbContext = new BestellverwaltungEntities();
    var produkte = from d in dbContext.Produkte select d;
    foreach(var produkt in produkte) {
        lstKunden.Items.Add(produkt.Bezeichnung);
    }
    DataContext = this;
}
```

Hier lesen wir mit einer LINQ-Abfrage über die Liste der Produkte alle Elemente ein. Diese speichern wir in der Auflistung produkte, deren Elemente wir dann in einer **foreach**-Schleife durchlaufen. In der **foreach**-Schleife fügen wir der ListBox dann

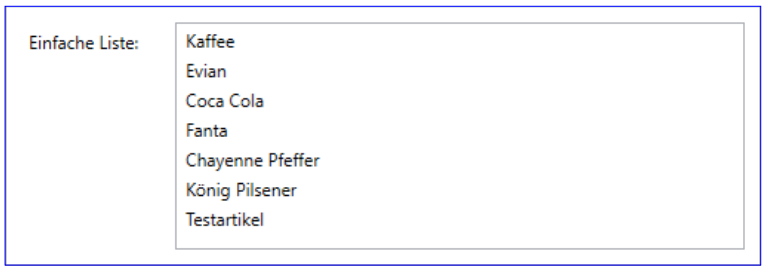


Bild 1: Anzeige der Bezeichnungen der Tabelle **Produkte**

mit der **Add**-Methode der **Items**-Auflistung die **Bezeichnung** des jeweiligen Produkts zu. Dieses Beispiel hilft uns natürlich nur weiter, wenn wir die Daten lediglich anzeigen und gegebenenfalls genau die angezeigten Werte weiterverwenden wollen. Wenn Sie über diese Einträge den Primärschlüsselwert des jeweiligen Datensatzes ermitteln wollen, ist es der falsche Ansatz. In diesem Fall benötigen Sie weitere Informationen.

ListBox an Liste binden

Im zweiten Beispiel (siehe Seite **pgeListeGebunden.xaml**) wollen wir das **ListBox**-Element an eine Liste binden, die wir aus dem Entity Data Model gefüllt haben. Die ListBox definieren wir im ersten Schritt wieder ganz einfach:

```
<ListBox x:Name="lstProdukte" Grid.Column="1" Margin="10,10,10,10" />
```

Der Code der Klasse sieht auch übersichtlich aus. Hier definieren wir zunächst den Datenbank-Kontext **dbContext**. Die Konstruktor-Methode erstellt den Datenbank-Kontext und füllt dann die Variable **produkte** mit einer Liste der Produkte, die wie wieder per LINQ-Abfrage ermitteln. Dann weisen wir die Liste **produkte** der Eigenschaft **ItemsSource** zu und stellen **DisplayMemberPath** auf das Feld **Bezeichnung** ein. Somit wird die Bezeichnung des jeweiligen Produkts angezeigt:

```
public partial class pgeListeGebunden : Page {
    BestellverwaltungEntities dbContext;
    public pgeListeGebunden() {
        InitializeComponent();
        dbContext = new BestellverwaltungEntities();
        var produkte = (from p in dbContext.Produkte select p).ToList();
        lstProdukte.ItemsSource = produkte;
        lstProdukte.DisplayMemberPath = "Bezeichnung";
    }
}
```

Das Ergebnis sieht genauso aus wie beim vorherigen Beispiel.

Binding im XAML-Code

Nun wollen wir noch ein wenig mehr mit den Bindungs-Eigenschaften der WPF-Steuerelemente arbeiten, sprich: die beiden Eigenschaften **ItemsSource** und **DisplayMemberPath** sollen im XAML-Code aufgeführt werden:

```
<ListBox x:Name="lstProdukte" ... ItemsSource="{Binding Produkte}" DisplayMemberPath="Bezeichnung"/>
```

Damit diese Eigenschaften auch die gewünschten Elemente in der Code behind-Datei finden, gestalten wir den Code dort etwas anders als beim vorhergehenden Beispiel. Nicht nur, dass wir die Zuweisung der beiden Eigenschaften **ItemsSource** und **DisplayMemberPath** dort natürlich entfernen.

Stattdessen fügen wir eine private Variable zum Speichern einer Produkte-Liste namens **produkte** hinzu. Diese machen wir über die öffentliche Eigenschaft **Produkte** von außen zugreifbar, also auch für die **ItemsSource**-Eigenschaft des **ListBox**-Elements:

```
public partial class pgeListeGebundenWPF : Page {
    BestellverwaltungEntities dbContext;
    private List<Produkt> produkte;
    public List<Produkt> Produkte {
        get { return produkte; }
        set { produkte = value; }
    }
```

Die Konstruktor-Methode muss dann natürlich dafür sorgen, dass **produkte** mit den gewünschte Einträgen gefüllt wird. Das erledigen wir, indem wir zuerst wieder den Datenbank-Kontext initialisieren und dann **produkte** mit einer Liste von **Produkt**-Elementen füllen, die wir per LINQ ermitteln:

```
    public pgeListeGebundenWPF() {
        InitializeComponent();
        dbContext = new BestellverwaltungEntities();
        produkte = new List<Produkt>((from p in dbContext.Produkte select p));
        DataContext = this;
    }
}
```

Eintrag hinzufügen

Wenn Sie der Liste etwa per Schaltfläche einen Eintrag hinzufügen wollen, sollte dieser natürlich auch direkt in der ListBox erscheinen. Einer neue Schaltfläche namens **btnNeuerEintrag** fügen wir etwa die folgende Methode für das Ereignis **Click** hinzu:

```
private void btnNeuerEintrag_Click(object sender, RoutedEventArgs e) {
    Produkt produkt = new Produkt();
    produkt.Bezeichnung = "Neues Produkt";
    produkte.Add(produkt);
}
```

Bei Anklicken wird so zwar nun ein neuer Eintrag zur Liste **produkte** hinzugefügt, was Sie durch Debuggen und Abfrage der **produkte**-Elemente prüfen können, allerdings erscheint der neue Eintrag nicht automatisch in der Liste. Das liegt daran, dass wir die Elemente in einer Auflistung des Typs **List** gespeichert haben. Sollen die Einträge nun aktualisiert werden, müssten Sie die Liste erneut der **ListBox** zuweisen.

Aktualisieren der ListBox mit einer ObservableCollection

Die Lösung ist das Austauschen der **List**-Auflistung durch eine **ObservableCollection**-Auflistung, wie wir es in **pgeListeGebundenWPFObservableCollection** gemacht haben.

Dort haben wir den Code der Code behind-Klasse wie folgt geändert. Zunächst haben wir den Typ der privaten und der öffentlichen Auflistung von **List** in **ObservableCollection** geändert:

```
public partial class pgeListeGebundenWPFObservableCollection : Page {
    BestellverwaltungEntities dbContext;
    private ObservableCollection<Produkt> produkte;
    public ObservableCollection<Produkt> Produkte {
        get { return produkte; }
        set { produkte = value; }
    }
```

Dementsprechend wird die Liste produkte mit einem neuen Objekt des Typs **ObservableCollection** gefüllt:

```
    public pgeListeGebundenWPFObservableCollection() {
        InitializeComponent();
        dbContext = new BestellverwaltungEntities();
        produkte = new ObservableCollection<Produkt>((from p in dbContext.Produkte select p));
        DataContext = this;
    }
```

Die Methode **btnNeuerEintrag_Click** arbeitet nun wie erwartet und fügt gleich den neuen Eintrag zur ListBox hinzu. Hier haben wir noch zwei Anweisungen hinzugefügt, die das neue **Produkt**-Element auch noch zur **Produkte**-Auflistung des Datenbank-Kontext hinzufügen und die Änderung speichern, damit diese auch in die Datenbank übertragen wird:

```
    private void btnNeuerEintrag_Click(object sender, RoutedEventArgs e) {
        Produkt produkt = new Produkt();
        produkt.Bezeichnung = "Neues Produkt";
        dbContext.Produkte.Add(produkt);
        produkte.Add(produkt);
        dbContext.SaveChanges();
    }
}
```

Einträge auswählen

Interessant wird es, wenn Sie die vom Benutzer getroffene Auswahl auswerten wollen. Das können Sie beispielsweise gleich nach der Auswahl erledigen oder auch zu einem späteren Zeitpunkt – etwa, wenn eine Schaltfläche betätigt wird.

Wir wollen uns im folgenden Beispiel (siehe **pgeAuswahlErmitteln**) zunächst das Ereignis **SelectionChanged** ansehen, das wir wie folgt im XAML-Code eintragen:

```
<ListBox x:Name="lstProdukte" Grid.Column="1" Margin="5" ItemsSource="{Binding Produkte}" DisplayMemberPath="Bezeichnung"
SelectionChanged="lstProdukte_SelectionChanged"/>
```

Nun benötigen wir noch die dadurch ausgelöste Methode. Diese definieren wir im Code behind-Modul wie folgt:

EDM: Bilder speichern und anzeigen

In der Bestellverwaltung haben wir bisher nur mit einfachen Daten in Datentypen wie Text, Zahl oder Datum gearbeitet. Dabei kann man doch mit WPF auch wunderbar Bilddateien anzeigen. In diesem Artikel erweitern wir die Datenbank Bestellverwaltung um das Speichern und die Anzeige von Bildern zu den Produktdatensätzen.

Speicherort: Tabellenfeld oder Dateisystem?

Eine der wichtigsten Fragen, wenn es darum geht, Bilder mit einer Datenbank zu verwalten, ist die nach dem Speicherort der Bilder. Grundsätzlich gibt es zwei einfache Varianten: Die Bilder werden im Dateisystem gespeichert und die Datenbank hält nur die Dateipfade zu den Bildern in den jeweiligen Datensätzen vor. Oder Sie speichern die Bilder direkt in einem entsprechenden Feld in der Datenbank. Wir werden an dieser Stelle, an der wir mit einer SQLite-Datenbank arbeiten, die Variante mit dem Dateisystem wählen, und zwar aus folgenden Gründen:

- Datenbanken sind dazu gemacht, referenziell verknüpfte Daten zu speichern, diese zu indizieren und schnell zugreifbar zu machen oder zu ändern. Das Speichern von Bildern oder anderen Dateien in den Tabellen der Datenbank ist da eher kontraproduktiv.

- Wenn Sie die Bilder in der Datenbankdatei speichern und die Daten sichern wollen, müssen Sie immer die komplette Datenbank inklusive der enthaltenen Bilder sichern. Wenn sich die Bilder im Dateisystem etwa in einem Ordner im gleichen Verzeichnis wie die Datenbank befinden, können diese separat gesichert werden, was nur dann nötig ist, wenn sich diese geändert haben.

- Zum Bearbeiten oder Anzeigen müssen die Bilder in der Regel entweder ohnehin im Dateisystem gespeichert oder aber anderweitig in den Speicher geladen werden. Dann kann man sie auch direkt aus dem Dateisytem heraus öffnen.

Optimalerweise speichern wir die Bilder auch noch in einem Verzeichnis, dass sich auf der gleichen Ebene oder unterhalb der Anwendung befindet. Auf diese Weise brauchen wir nur den relativen Pfad zur Bilddatei in der Datenbank zu speichern.

Tabelle Produkte erweitern

Damit wir die Bilder, die wir im Dateisystem speichern, mit den Datensätzen der Tabelle **Produkte** verknüpfen können, fügen wir dieser Tabelle ein Feld namens Bild hinzu. Es soll ein einfaches Textfeld sein, dass den Pfad zur Bilddatei speichert, und zwar relativ zur Datenbank in einem eigenen Ordner namens **Bilder**.

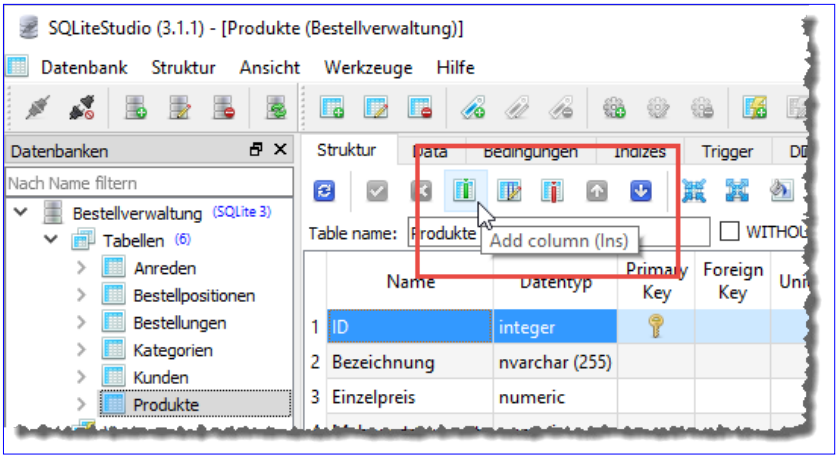


Bild 1: Anlegen eines neuen Feldes

Diese Änderung nehmen wir der Einfachheit halber über das Tool **SQLiteStudio** vor. Dort öffnen Sie die Datenbank **Bestellverwaltung** und klicken dann doppelt auf die Tabelle **Produkte**. Nun betätigen Sie die Schaltfläche **Add column (Ins)** wie in Bild 1.

Dies öffnet den Dialog aus Bild 2. Hier tragen Sie den Namen des neuen Feldes sowie den Datentyp ein. Anschließend schließen Sie den Dialog mit der **OK**-Schaltfläche.

Erst der Klick auf die Schaltfläche **Commit structure changes** öffnet den Dialog **Auszuführende Abfragen**, der die SQL-Abfragen mit den geplanten Änderungen anzeigt und per Klick auf **OK** die Änderung in die Datenbank überträgt (siehe Bild 3).

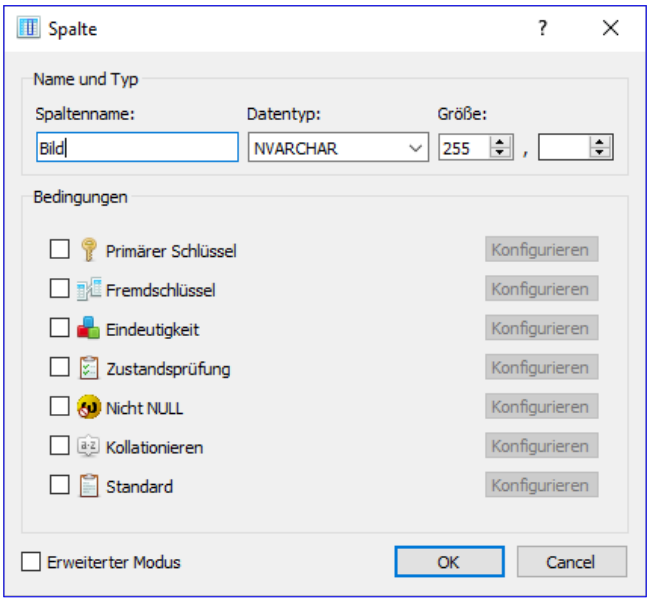


Bild 2: Anlegen des Feldes **Bild**

Nun müssen wir die Änderung in der Datenbank noch in das Entity Data Model übertragen. Dazu öffnen Sie die Ansicht **Bestellverwaltung.edmx** und wählen aus dem Kontextmenü den Eintrag **Modell aus der Datenbank aktualisieren...** aus (siehe Bild 4).

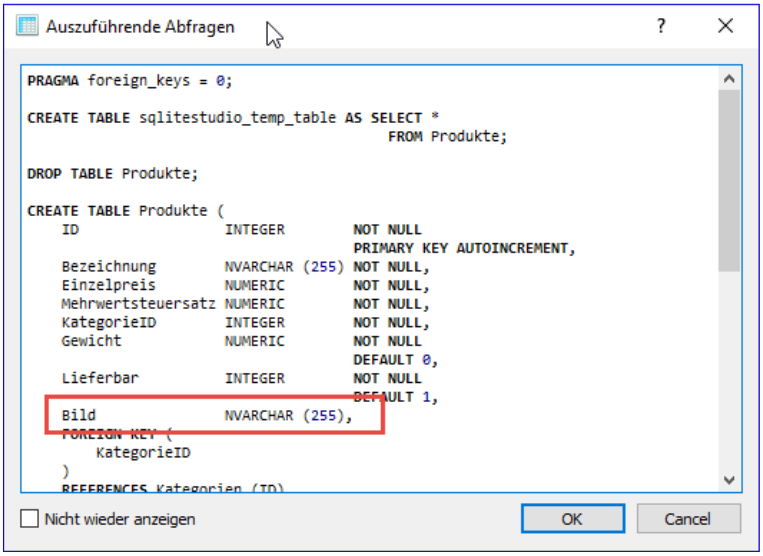


Bild 3: SQL-Anweisung zum Anlegen der Tabelle mit dem Feld **Bild**

Danach erscheint das Feld **Bild** dann auch im Element **Produkt** des Entity Data Models. Nun können wir also Bildpfade zum Feld **Bild** des Elements **Produkt** hinzufügen. Damit sind die Arbeiten am Entity Framework erledigt.

XAML-Code definieren

Nun wollen wir das Aussehen des Fensters zur Anzeige der Produktdaten mit den Bildern definieren. Dies erledigen wir diesmal gleich im Fenster **MainWindow.XAML**. Damit die Steuerelemente wie gewünschten angeordnet werden, haben wir über die Elemente **Grid.RowDefinitions** und **Grid.ColumnDefinitions** einige Zeilen und Spalten definiert. Die Definition dieser Elemente sieht wie folgt aus, wobei die Position des Elements mit dem Wert * für das Attribut **Height** eine besondere Rolle spielt:

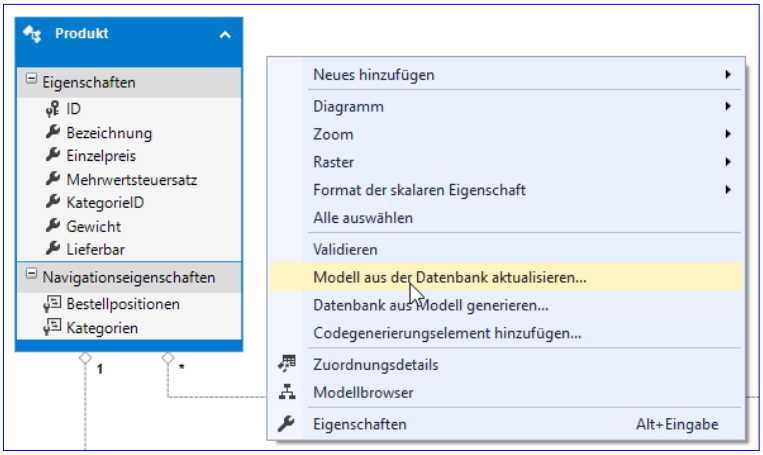


Bild 4: Aktualisieren des Entity Data Models

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition> //für Header
    <RowDefinition Height="Auto"></RowDefinition> //für ListBox
    <RowDefinition Height="Auto"></RowDefinition> //für ID des Produkts
    <RowDefinition Height="Auto"></RowDefinition> //für Bezeichnung
    <RowDefinition Height="Auto"></RowDefinition> //für Bildpfad
    <RowDefinition Height="*"></RowDefinition> //für Image-Steuerelement
    <RowDefinition Height="Auto"></RowDefinition> //für Schaltflächen
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition Width="*"></ColumnDefinition>
</Grid.ColumnDefinitions>
```

Hier hinterlegen wir nun das **ListBox**-Steuerelement, das in der zweiten Zeile über zwei Spalten angeordnet werden soll. Dieses fügen wir in der folgenden Form zum XAML-Code hinzu:

```
<ListBox x:Name="lstProdukte" Grid.Row="1" Grid.ColumnSpan="2" Margin="5" ItemsSource="{Binding Produkte}"
DisplayMemberPath="Bezeichnung" SelectionChanged="lstProdukte_SelectionChanged"/>
```

Das **ListBox**-Element heißt **lstProdukte** und ist an die Auflistung **Produkte** als **ItemsSource** gebunden. Angezeigt soll der Inhalt der Eigenschaft **Bezeichnung** des gebundenen Objekts. Wenn sich die Auswahl ändert, etwa weil der Benutzer ein anderes Element anklickt, soll dies die Ereignisprozedur **lstProdukte_SelectionChanged** auslösen. Damit das **ListBox**-Element die Elemente der Produkte auflistet, fügen wir der Code behind-Klasse **MainWindow.xaml.cs** einige Codezeilen hinzu. Den Beginn macht die klassenweite Definition des **DBContext**-Elements:

```
BestellverwaltungEntities dbContext;
```

Damit wir die Produkte als Datenherkunft für die ListBox haben und das einzelne Produkt-Objekt als Datenherkunft für die Steuerelemente des Fensters, definieren wir außerdem noch eine private Variable für die Liste der Produkte, und zwar für eine ObservableCollection mit Elementen des Typs **Produkt**:

```
private ObservableCollection<Produkt> produkte;
```

Den Inhalt dieser Variablen wollen wir auch öffentlich zugänglich machen, damit wir die Elemente des XAML-Codes daran binden können. Dies erledigen wir wie folgt:

```
public ObservableCollection<Produkt> Produkte {
    get { return produkte; }
    set { produkte = value; }
}
```

Auch für das jeweils mit den Textfeldern anzuzeigende Produkt, dessen Bild wir ja auch laden wollen, deklarieren wir ein Objekt namens **produkt**, dass eine Instanz des Typs **Produkt** speichern kann:

```
private Produkt produkt;
```

Die öffentliche Eigenschaft kann über den Namen **Produkt** gelesen und geschrieben werden. Damit beim Füllen der Variablen **produkt** gleich die daran gebundenen Steuerelemente aktualisiert werden, implemtieren wir die Schnittstelle **INotifyProperty-Changed** für die aktuelle Klasse. Der Teil in der öffentlichen Eigenschaft Produkt erhält dazu im Setter einen Aufruf der Methode **OnPropertyChanged**:

```
public Produkt Produkt {
    get { return produkt; }
    set {
        produkt = value;
        OnPropertyChanged(new PropertyChangedEventArgs("Produkt"));
    }
}
```

Für eine ordnungsgemäße Implementierung der Schnittstelle muss diese natürlich im Kopf der Klasse angegeben werden:

```
public partial class MainWindow : Window, INotifyPropertyChanged {
...
}
```

Außerdem benötigen wir noch diese beiden Elemente:

```
protected virtual void OnPropertyChanged(PropertyChangedEventArgs e) {
    if (PropertyChanged != null) {
        PropertyChanged(this, e);
    }
}
public event PropertyChangedEventHandler PropertyChanged;
```

Danach wenden wir uns der Konstruktor-Methode der Klasse zu. Diese initialisiert das Objekt auf Basis der XAML-Definition, initialisiert das DBContext-Objekt, füllt die ObservableCollection namens **produkte** und weist dem Fenster die Code behind-Klasse als Datenquelle zu. Damit das **ListBox**-Element das Element markiert, das beim Anzeigen des Fensters in den übrigen Steuerelementen angezeigt wird (und somit der privaten Variablen **produkt** zugewiesen ist), stellen wir produkt auf das mit **First** ermittelte erste Element der **produkte**-Auflistung ein und markieren gleich auch noch diesen Eintrag in der **ListBox**:

```
public MainWindow() {
    InitializeComponent();
```

```
    dbContext = new BestellverwaltungEntities();
    produkte = new ObservableCollection<Produkt>(dbContext.Produkte);
    DataContext = this;
    produkt = produkte.First();
    lstProdukte.SelectedItem = produkt;
}
```

Anzeigen der Detaildaten eines Produkts

Bevor wir uns dem Bild zuwenden, schauen wir uns noch die Steuerelemente zur Anzeige der Produktdetails an. Diese definieren wir in XAML wie folgt:

```
<Label Grid.Row="2" Grid.Column="0" Margin="5">ID:</Label>
<TextBox Grid.Row="2" Grid.Column="1" x:Name="txtID" Text="{Binding Path=Produkt.ID}" Margin="5" TextAlignment="Left"
Width="40" HorizontalAlignment="Left"></TextBox>
<Label Grid.Row="3" Grid.Column="0" Margin="5">Bezeichnung:</Label>
<TextBox  Grid.Row="3" Grid.Column="1" x:Name="txtBezeichnung" Text="{Binding Path=Produkt.Bezeichnung}" Margin="5"
TextAlignment="Left"></TextBox>
<Label Grid.Row="4" Grid.Column="0" Margin="5">Bildpfad:</Label>
<TextBox x:Name="txtBild" Text="{Binding Path=Produkt.Bild}" Grid.Row="4" Grid.Column="1" Margin="5"
TextAlignment="Left"></TextBox>
```

Damit erhalten wir drei **TextBox**-Elemente, die über die **Text**-Eigenschaft an die Eigenschaften **Produkt.ID**, **Produkt.Bezeichnung** und **Produkt.Bild** gebunden werden. Diese Syntax rührt daher, dass wir in der Code behind-Klasse, die wir als **DataContext** des Fensters angegeben haben, ein **Produkt**-Element über die öffentliche Eigenschaft **Produkt** bereitstellen, dass wiederum die Eigenschaften der Klasse **Produkt** liefert. Nun fügen wir noch ein **Image**-Element hinzu, mit dem wir gleich ein in der Eigenschaft **Bild** angegebenes Bild anzeigen wollen. Dieses und das entsprechende Bezeichnungselement definieren wir mit den folgenden beiden Zeilen (das **Source**-Attribut erläutern wir weiter unten):

```
<Label Grid.Row="5" Grid.Column="0" Margin="5">Bild:</Label>
<Image x:Name="imgBild" Grid.Row="5" Grid.Column="1" HorizontalAlignment="Left" Source="{Binding Produkt.Bild,
Converter={StaticResource ImagePathConverter}}"></Image>
```

Schließlich fügen wir noch zwei Schaltflächen zum Fenster hinzu. Die erste soll die Auswahl eines neuen Bildes für ein Produkt ermöglichen, die zweite soll geänderte Daten speichern. Die Definition dieser beiden Schaltflächen packen wir in **StackPanel**-Element, dass sich wieder über zwei Spalten erstreckt:

```
<StackPanel Orientation="Horizontal" Grid.Row="10" Grid.ColumnSpan="2">
    <Button x:Name="btnBildAuswaehlen" Margin="5" Click="btnBildAuswaehlen_Click">Bild auswählen</Button>
    <Button x:Name="btnSpeichern" Margin="5" Click="btnSpeichern_Click">Speichern</Button>
</StackPanel>
```

SQLite-Datenmodellierung per C#

Wenn Sie eine SQLite-Datenbank von einer WPF/C#-Anwendung aus nutzen, möchten Sie gegebenenfalls einmal Tabellen in dieser Datenbank anlegen, ändern oder löschen oder gar neue Datenbanken mit C# erstellen. Dies kann beispielsweise interessant werden, wenn Sie eine neue Version einer Anwendung ausliefern, aber der Benutzer die Daten in der Datenbank der vorherigen Version weiter nutzen möchte. Dieser Artikel stellt die grundlegenden Vorgehensweisen für diese Arbeitsschritte vor.

NuGet-Paket für SQLite hinzufügen

Wenn Sie von einer C#-Anwendung aus Änderungen an einer SQLite-Datenbank vornehmen oder diese sogar erstellen wollen, benötigen Sie das NuGet-Paket mit den entsprechenden Erweiterungen. Dazu klicken Sie mit der rechten Maustaste auf das Projekt im Projektmappen-Explorer und wählen den Kontextmenü-Eintrag **NuGet-Pakete verwalten...** aus. Im nun erscheinenden Bereich wechseln Sie auf Durchsuchen und suchen nach **SQLite**. Den nun auftauchenden Eintrag **System.Data.SQLite** können Sie dann auswählen und installieren.

SQLite-Befehle verfügbar machen

In der Klasse, in der Sie die Befehle zum Erstellen und Anpassen von SQLite-Datenbanken unterbringen wollen, benötigen Sie zunächst einen Verweis auf den Namespace **System.Data.SQLite**, der erst nach dem Hinzufügen des oben genannten NuGet-Pakets vorfinden:

```
using System.Data.SQLite;
```

Leere Datenbank erstellen

Um eine leere Datenbank zu erstellen, fügen wir dem Fenster **MainWindow.xaml** ein Textfeld zur Eingabe des Datenbanknamens sowie eine Schaltfläche zum Erstellen der leeren Datenbankdatei hinzu. Diese Schaltfläche soll die folgende Methode auslösen:

```
private void btnCreateDatabase_Click(object sender, RoutedEventArgs e) {
    string databaseFile = AppDomain.CurrentDomain.BaseDirectory + txtDatenbankname.Text;
    File.Delete(databaseFile);
    SQLiteConnection.CreateFile(databaseFile);
    if (File.Exists(databaseFile)) {
        MessageBox.Show("Leere Datenbank \n\n'" + databaseFile + "'\n\n wurde erstellt.");
    }
}
```

Die Methode trägt den Pfad der **.exe**-Datei plus den vom Benutzer gewählten Datenbanknamen in die Variable **databaseFile** ein. Diese Datei wird, sofern vorhanden, gelöscht und dann mit der Methode **CreateFile** des Objekts **SQLiteConnection** neu erstellt. Dabei übergeben Sie die Variable mit dem Datenbankpfad als Parameter. Die folgende **if**-Bedingung prüft, ob nach

diesem Vorgang eine Datei mit dem angegebenen Namen vorhanden ist und zeigt in diesem Fall eine entsprechende Meldung an.

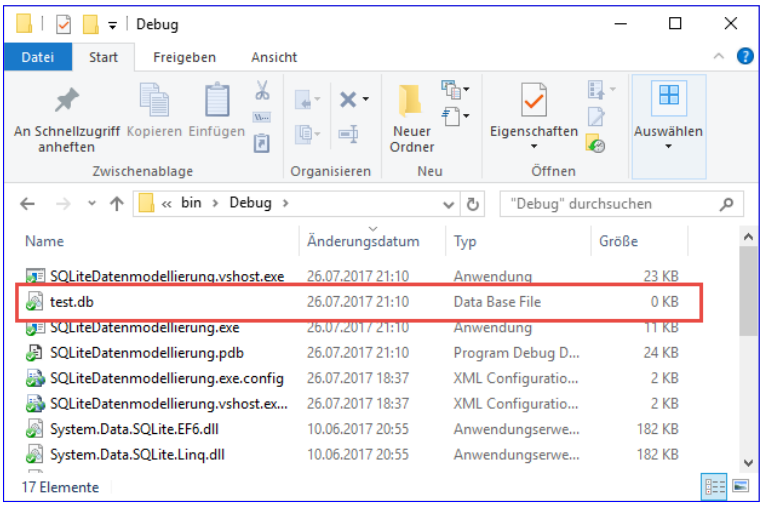


Bild 1: Frisch angelegte Datenbankdatei

Neue Tabelle erstellen

Nun wollen wir der frisch angelegten Datenbank eine erste Tabelle hinzufügen. Das ist dann auch tatsächlich der erste Inhalt der Datenbankdatei – diese weist nach dem Anlegen nämlich die Dateigröße **0** auf (siehe Bild 1). Danach rufen wir mit der Tasten **btnCreateTable** die folgende Methode auf, die wieder den Pfad der **.exe**-Datei und den Dateinamen aus dem Textfeld in der Variablen **databaseFile** zusammensetzt. Dann erstellt sie ein neues Objekt des Typs **SQLiteConnection**, der sie als Konstruktor-Parameter eine Zeichenkette übergibt, die aus **Data Source=**, dem Datenbankpfad und der zu verwenden Version besteht.

Die **Open**-Methode öffnet die Verbindung zu dieser Datenbank. Dann schreibt die Methode eine **CREATE TABLE**-Anweisung, die eine neue Tabelle namens **Beispieltabelle** mit den beiden Feldern **id** und **Beispielstring** enthält. Nun erstellt die Methode ein neues **SQLiteCommand**-Objekt und übergibt die SQL-Anweisung aus der String-Variablen **sql** sowie das Verbindungsobjekt aus **connection** als Parameter. Die **ExecuteNonQuery**-Methode führt die **CREATE TABLE**-Anweisung schließlich aus. Mit der **Close**-Methode schließen wir die Verbindung:

```
private void btnCreateTable_Click(object sender, RoutedEventArgs e) {
    string databaseFile = AppDomain.CurrentDomain.BaseDirectory + txtDatenbankname.Text;
    SQLiteConnection connection = new SQLiteConnection("Data Source=" + databaseFile + ";Version=3;");
    connection.Open();
    string sql = "CREATE TABLE Beispieltabelle (id INT, Beispielstring VARCHAR(255))";
    SQLiteCommand command = new SQLiteCommand(sql, connection);
    command.ExecuteNonQuery();
    connection.Close();
}
```

Ein Blick in den Windows Explorer zeigt, dass dies nicht spurlos an der Datenbankdatei vorübergegangen ist: Diese hat nun immerhin eine Größe von acht Kilobytes. Aber befindet sich auch die neue Tabelle darin? Dies prüfen wir mit dem Tool **SQLiteStudio**, das Sie kostenlos über das Internet herunterladen können. Mit dem Menüeintrag **Datenbank|Datenbank hinzufügen** öffnen Sie den Dialog aus Bild 2. Hier wählen Sie die soeben angelegte Datei aus und klicken nach dem Betätigen von **Verbindung testen** auf die Schaltlfäche **OK**.

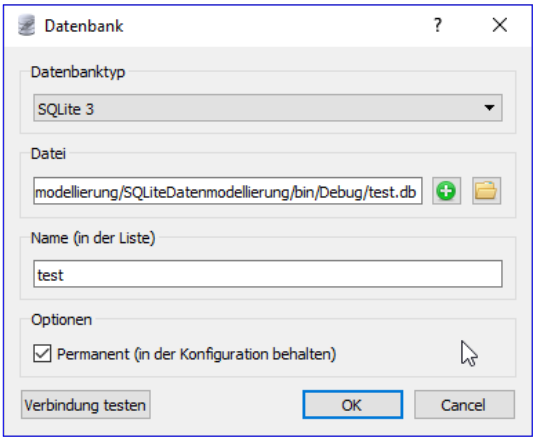


Bild 2: Hinzufügen der Datenbank zum **SQLiteStudio**

Danach finden Sie im Hauptfenster von **SQLiteStudio** bereits die Datenbank mit der ersten Tabelle im Datenbank-Explorer vor (siehe Bild 3). Allerdings verfügt das Feld **id** noch nicht über eine Primärschlüssel, aber das können wir ja gleich nachholen – indem wir die neue Tabelle verändern.

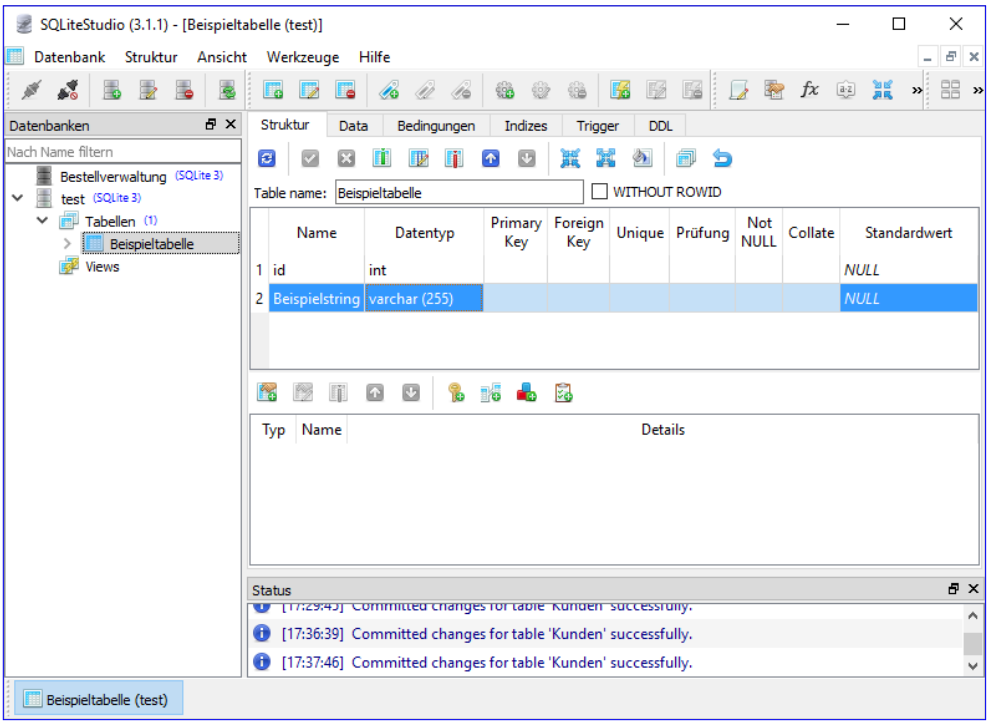


Bild 3: Die neue Tabelle im **SQLiteStudio**

Tabelle mit Primärschlüsselfeld erstellen

Im nächsten Beispiel wollen wir eine Tabelle erstellen und das **ID**-Feld direkt als Primärschlüsselfeld kennzeichnen. Dazu fügen wir der **CREATE TABLE**-Methode gleich das Schlüsselwort **PRIMARY KEY** für das Feld **ID** hinzu. Dazu nutzen wir prinzipiell die gleiche Methode wie für das vorherige Beispiel:

```
private void btnTabelleMitPKAnlegen_Click(object sender, RoutedEventArgs e) {
    string databaseFile = AppDomain.CurrentDomain.BaseDirectory + txtDatenbankname.Text;
    string connectionString = "Data Source=" + databaseFile + ";Version=3;";
    MessageBox.Show(connectionString);
    SQLiteConnection connection = new SQLiteConnection(connectionString);
    connection.Open();
    string sql = "CREATE TABLE BeispieltabelleMitPK (id INT PRIMARY KEY, Beispielstring VARCHAR(255))";
    SQLiteCommand command = new SQLiteCommand(sql, connection);
    command = new SQLiteCommand(sql, connection);
    command.ExecuteNonQuery();
    connection.Close();
}
```

Auch das liefert das gewünschte Ergebnis, wie ein Blick in **SQLiteStudio** beweist (siehe Bild 4).

Nun habe ich allerdings beim Erstellen der neuen Tabelle bei Experimentieren vergessen, das Textfeld **txtDatenbankname** zu füllen. Es gibt allerdings beim Durchlaufen der obigen Methode keinerlei Fehlermeldung – wie kann das sein? Theoretisch dürfte der Zugriff auf die Datenbank mit der **Open**-Methode doch gar nicht funktionieren, da die Datenbankdatei gar nicht vor-

handen ist? Doch das ist ein Fehlschluss: Dadurch, dass wir **txtDatenbankname** nicht gefüllt haben, übergeben wir nämlich einen String wie den folgenden mit **connectionString** an die **Open**-Methode:

```
Data Source=C:\Users\User\...\bin\
Debug\;Version=3;
```

Wenn wir nun in das Verzeichnis **...\bin\Debug** schauen, finden wir dort auch keine neue Datenbank etwa mit einem beim Öffnen temporär vergebenen Namen. Wenn wir allerdings eine Ebene nach oben gehen, also in das Verzeichnis **...\bin**, finden wir die Situation aus Bild 5 vor. SQLite legt, wenn die in der Verbindungszeichenfolge für die **Open**-Methode angegebene Datenbankdatei noch nicht vorhanden ist, eine neue Datenbank an. In diesem Fall hat SQLite **Debug\;Version=3;** als Dateinamen interpretiert und eine Datei namens **Debug;Version=3** angelegt (das Backslash-Zeichen **** wurde dabei als Escape-Zeichen für das Semikolon interpretiert). Die Größe von **20 KB** zeigt an, dass hier auch gleich die gewünschte Tabelle angelegt wurde.

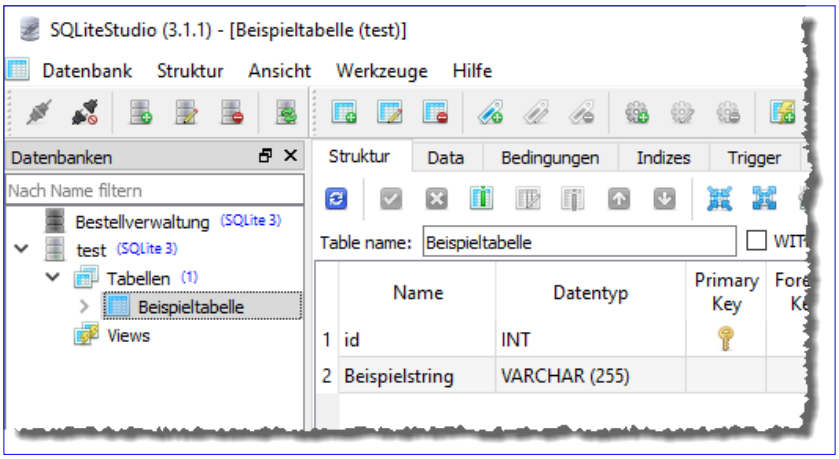


Bild 4: Tabelle mit Primärschlüssel im **SQLiteStudio**

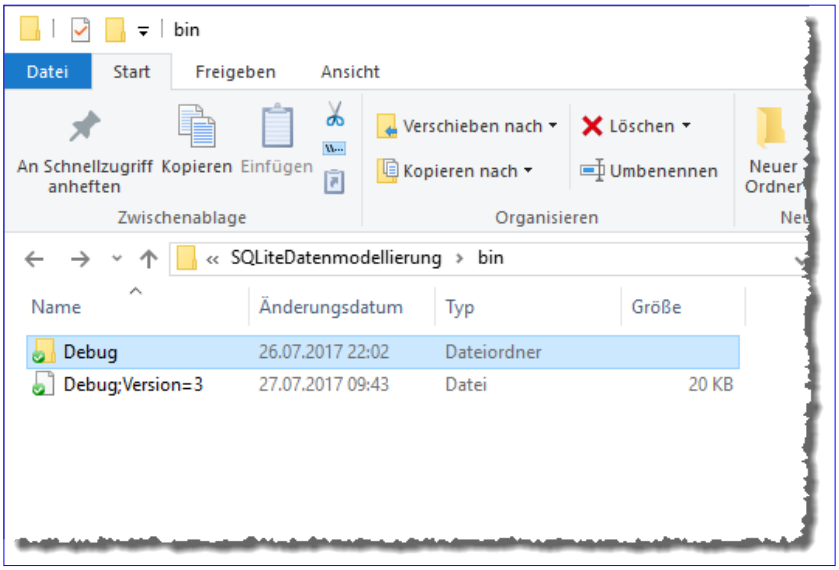


Bild 5: Eine Datenbankdatei namens **Debug;Version=3**

Autowert definieren

Beim Einfügen von Datensätzen müssen wir uns Gedanken darum machen, auf welche Weise der Wert des Primärschlüsselfeldes definiert wird. Unter Access wurde das immer einfach durch Festlegen der Eigenschaft **Autowert** für das betroffene Feld erledigt. Unter SQLite ist das auch nicht viel anders: Hier fügen Sie hinter dem **PRIMARY KEY**-Schlüsselwort einfach noch **AUTOINCREMENT** ein (siehe Methode **btnTabelleMitAutowertAnlegen_Click**):

```
CREATE TABLE BeispieltabelleMitAutowert(ID INTEGER PRIMARY KEY AUTOINCREMENT, ...)
```

SQLite hat hier allerdings eine etwas andere Philosophie als andere Datenbanksysteme: Sie müssen gar nicht unbedingt eine Autoincrement-Funktion definieren, sondern können das Primärschlüsselfeld auch einfach nur als **INTEGER PRIMARY KEY** definieren. Wenn Sie dann beim Neuanlegen eines Datensatzes keinen Wert für das Primärschlüsselfeld angeben, fügt SQLite automatisch einen entsprechend ermittelten Wert ein. Gegenüber anderen Datenbanksystemen haben Sie so die Wahl, ob Sie

das System einen Primärschlüsselwert auswählen lassen wollen oder ob Sie diesen selbst vergeben wollen. Mehr dazu erfahren Sie weiter unten unter **Datensätze einfügen**.

Prüfen, ob Datenbank vorhanden ist

Wir wollen an dieser Stelle den Benutzer darauf hinweisen, dass die Datenbank noch nicht vorhanden ist und die Methode abbrechen. Das erledigen wir mit einer entsprechenden **if**-Bedingung:

```
private void btnTabelleMitPKAnlegen_Click(object sender, RoutedEventArgs e) {
    string databaseFile = AppDomain.CurrentDomain.BaseDirectory + txtDatenbankname.Text;
    if (!File.Exists(databaseFile)) {
        MessageBox.Show("Die Datenbank \n\n'" + databaseFile + "'\n\nist nicht vorhanden.");
    }
    else {
       //Tabelle anlegen
    }
}
```

Herstellen einer Verbindung in eigener Methode

In den bisherigen Beispielen haben wir jeweils einige Anweisungen eingebaut, welche die Verbindung herstellen und mit der Variablen **connection** referenzieren. Damit wir diese Zeilen nicht in jeder neuen Methode erneut schreiben müssen, gliedern wir diese in eine eigene Methode aus und erweitern deren Funktionsumfang gleich ein wenig.

Die neue Methode heißt VerbindungOeffnen und erwartet den Pfad zur Datenbankdatei als Parameter (**datenbankpfad**) sowie einen Boolean-Wert namens **datenbankErstellen**, der angibt, ob die Datenbank erstellt werden soll, wenn Sie noch nicht vorhanden ist. Die Methode stellt die Boolean-Variable **datenbankExistiert** zunächst auf **true** ein. Dann prüft sie, ob die mit **datenbankpfad** angegebene Datei existiert. Falls nicht, wird **datenbankExistiert** auf **false** eingestellt und eine Meldung ausgegeben, dass die Datenbank nicht vorhanden ist. Falls die Datenbank existiert, baut die Methode in der zweiten **if**-Bedingung eine Verbindungszeichenfolge zusammen, erstellt ein **SQLiteConnectionString**-Objekt auf Basis dieser Verbindung und öffnet diese mit der **Open**-Methode. Die Verbindung wird dann als Rückgabewert zurückgeliefert. Sollte **datenbankExistiert** den Wert **false** enthalten, liefert die Methode den Wert **null** zurück:

```
private SQLiteConnection VerbindungOeffnen(string datenbankpfad, bool datenbankErstellen = false) {
    bool datenbankExistiert = true;
    if (!File.Exists(datenbankpfad)) {
        if (!datenbankErstellen) {
            datenbankExistiert = false;
            MessageBox.Show("Die Datenbank \n\n'" + datenbankpfad+ "'\n\nist nicht vorhanden.");
        }
    }
    if (datenbankExistiert) {
        string connectionString = "Data Source=" + datenbankpfad + ";Version=3;";
```

```
        SQLiteConnection connection = new SQLiteConnection(connectionString);
        connection.Open();
        return connection;
    }
    else {
        return null;
    }
}
```

Prüfen, ob Tabelle vorhanden ist

Der nächste Schritt beim Anlegen einer Tabelle ist eine vorherige Prüfung, ob diese Tabelle bereits vorhanden ist. Wenn wir eine Connection haben und den Namen der zu prüfenden Tabelle, können Sie die folgende Methode verwenden, um zu prüfen, ob die Tabelle bereits vorhanden ist. Die Methode erwartet die Verbindung (**connection**) und den Tabellennamen (**tabellenname**) als Parameter. Sie fügt in sql eine SQL-Abfrage zusammen, welche die Tabelle **sqlite_master** nach einem Datensatz durchsucht, dessen Feld **type** den Wert **table** und dessen Feld **name** den übergebenen Tabellennamen enthält. Diese Abfrage übergeben wir wieder mit der Connection an ein neues **SQLiteCommand**-Objekt. Wir führen dieses allerdings diesmal nicht mit der Methode **ExecuteNonQuery** aus, sondern mit **ExecuteScalar**. Dies liefert die erste Spalte der ersten Zeile des Ergebnisses zurück. Wir haben für die erste Spalte den Wert **1** angegeben. Findet **ExecuteScalar** keinen passenden Datensatz, ist die Tabelle nicht vorhanden und es wird der Wert **null** zurückgeliefert. Als Rückgabewert nutzen wir einen Boolean-Wert, der angibt, ob die Variable **tabelleVorhanden**, die nun entweder den Wert **1** oder **null** enthält, nicht den Wert **null** aufweist – also den Wert **true**, wenn die Tabelle gefunden wurde und **false**, falls nicht:

```
private bool TabelleVorhanden(SQLiteConnection connection, string tabellenname) {
    string sql = "SELECT 1 FROM sqlite_master WHERE type = 'table' AND name = '" + tabellenname + "'";
    SQLiteCommand command = new SQLiteCommand(sql, connection);
    var tabelleVorhanden = command.ExecuteScalar();
    return (tabelleVorhanden != null);
}
```

Tabelle nur anlegen, wenn noch nicht vorhanden

Mit den beiden zuvor erstellten Methoden können wir den Code für das Anlegen einer Tabelle nur in dem Fall, dass diese noch nicht vorhanden ist, reduzieren. Dies sieht dann wie folgt aus:

```
private void btnPruefenObTabelleVorhanden_Click(object sender, RoutedEventArgs e) {
    string databaseFile = AppDomain.CurrentDomain.BaseDirectory + txtDatenbankname.Text;
    SQLiteConnection connection = VerbindungOeffnen(databaseFile, false);
    if (connection != null) {
        bool tabelleVorhanden = TabelleVorhanden(connection, "BeispieltabelleMitPK");
        if (!tabelleVorhanden) {
            string sql = "CREATE TABLE BeispieltabelleMitPK (id INT PRIMARY KEY, Beispielstring VARCHAR(255))";
            SQLiteCommand command = new SQLiteCommand(sql, connection);
```

Entity Framework: SQLite verknüpfen

In den letzten Artikel haben wir oft die SQLite-Datenbank als Beispieldatenbank verwendet. Nun kann es allerdings durch einen falschen Connection-String oder eine verschobene, gelöschte oder leere Datenbank dazu kommen, dass die Anwendung nicht über das Entity Framework auf die Datenbank zugreifen kann. Dieser Artikel beschreibt einige Szenarien, in denen das der Fall ist und zeigt, wie Sie beim Start der Anwendung prüfen können, ob die Quell-Datenbank erreichbar ist und diese gegebenenfalls neu auswählen.

Die SQLite-Datenbank ist eine sehr praktische Datenquelle für einfache Anwendungen, die keinen SQL Server benötigen. Sie kommt als eine einzige Datei daher und die Treiber können bequem in ein Projekt integriert werden – auch die Weitergabe ist überhaupt kein Problem. Was aber ein Problem sein könnte, wissen Access-Entwickler, die Datenbanken in Front- und Backend aufgeteilt haben, nur zu gut: Die Datenbankdatei könnte sich nicht an dem Ort befinden, der in der Verbindungszeichenfolge des Frontends angegeben ist.

Mit einem .NET-WPF-Projekt und einer SQLite-Datenbank haben Sie ein klassisches Frontend-Backend-Szenario, wie es auch bei aufgeteilten Access-Datenbanken vorliegt. Im Gegensatz zur Konstellation mit einem SQL Server haben Sie keine Server-Angabe im Connection-String, sondern Sie müssen den Namen der Datenbank, gegebenenfalls mit Angabe des Pfades zu dieser Datenbank, angeben. Wenn Sie ein Entity Data Model zu Ihrem Projekt hinzugefügt und die Datenbankdatei, zum Beispiel **Bestellverwaltung.db**, als Datenquelle ausgewählt haben, schreibt der Entity Data Model-Assistent eine Verbindungszeichenfolge in die Datei **App.config**, die etwa wie folgt aussieht:

```
<configuration>
  ...
  <connectionStrings>
    <add name="BestellverwaltungEntities" connectionString="...;provider=System.Data.SQLite.EF6;provider connection
        string=&amp;quot;data source=c:\Beispiel\Bestellverwaltung.db&amp;quot;" providerName="System.Data.EntityClient" />
  </connectionStrings>
</configuration>
```

Hier sehen Sie, dass der Assistent zum Erstellen des Entity Data Models den absoluten Pfad zur Datenbankdatei **Bestellverwaltung.db** angegeben hat. Das geht solange gut, wie Sie die Anwendung auf dem Entwicklungsrechner debuggen oder einsetzen und die Datenbank sich an Ort und Stelle befindet.

Wenn Sie jedoch die Anwendung auf einen anderen Rechner verschieben, ist die Wahrscheinlichkeit hoch, dass die Datenbankdatei unter diesem Pfad nicht mehr zu erreichen ist. Ich selbst programmiere beispielsweise auf zwei verschiedenen Rechnern, wobei die Dateien per DropBox jeweils auf den aktuellsten Stand gebracht werden. Und sobald das Projekt samt Datenbankdatei auf dem einen Rechner nicht auf dem gleichen Pfad wie auf dem anderen Rechner liegt, findet eines der beiden Projekte die Datenbankdatei nicht mehr.

Relativ statt absolut

In diesem Fall ist die erste Maßnahme, den Pfad aus der Angabe der Quelldatenbank zu entfernen und nur noch den Dateinamen anzugeben. Das sieht dann in der Datei **App.config** wie folgt aus:

```
<configuration>
  ...
  <connectionStrings>
    <add name="BestellverwaltungEntities" connectionString="...;provider=System.Data.SQLite.EF6;provider connection
        string=&quot;data source=Bestellverwaltung.db&quot;" providerName="System.Data.EntityClient" />
  </connectionStrings>
</configuration>
```

Dies bedeutet nun nichts anderes, als dass die Datenbankdatei nun immer dann erreichbar ist, wenn sich diese im gleichen Verzeichnis wie die **.exe**-Datei des Projekts befindet.

Mehrbenutzer-Szenario

Das wird in einem Mehrbenutzer-Szenario natürlich nicht mehr funktionieren, denn schon wenn die Anwendung auf zwei Rechnern installiert ist, müsste man auf einem Rechner den absoluten Pfad zu der zu nutzenden Datenbankdatei hinterlegen.

Wir benötigen also einen Mechanismus, der beim Start der Anwendung ausgelöst wird und prüft, ob die zu verwendende Datenbankdatei verfügbar ist. Wenn dies nicht der Fall ist, soll ein Dialog erscheinen, mit dem der Benutzer die Datenbankdatei auswählen soll. Diese Änderung sollte dann optimalerweise irgendwo in den Einstellungen der Datenbank gespeichert und von nun an beim Start abgerufen werden können. Wir beginnen jedoch mit der Prüfung, ob die Datenbankdatei überhaupt an Ort und Stelle ist. Dies erledigen wir beim Start der Anwendung, und zwar beim Aufruf des Hauptfensters.

Beispielkonstellation

Wir erstellen im Fenster MainWindow eine minimale Konfiguration, um die Kunden aus der Datenbank **Bestellverwaltung.db** in einem DataGrid anzuzeigen. Dazu fügen Sie **MainWindow.xaml** folgenden Code hinzu:

```
<Grid>
    <DataGrid ItemsSource="{Binding Kunden}"></DataGrid>
</Grid>
```

Das Code behind-Modul MainWindow.xaml.cs statten wir so aus:

```
using System.Windows;
using System.Collections.ObjectModel;
namespace TestDatenbankVorhanden {
    public partial class MainWindow : Window {
        BestellverwaltungEntities dbContext = null;
        private ObservableCollection<Kunde> kunden;
```

Weitergabe von WPF/C#/SQLite-Anwendungen

Die bisher bekannte Methode im Rahmen der Entwicklung von Anwendung in diesem Magazin war das Kopieren des bin/debug-Verzeichnisses auf den Zielrechner. In dieser Ausgabe wollen wir uns nun ansehen, wie Sie Anwendungen mit den Bordmitteln von Visual Studio 2015 in der Community-Edition weitergeben können. Die Möglichkeiten sind beschränkt, aber für die meisten Fälle durchaus ausreichend.

Voraussetzungen

Für Testzwecke wollen wir unser bisher in diesem Magazin besprochene Anwendung **Bestellverwaltung** weitergeben. Dazu benötigen Sie die Erweiterung **Visual Studio Installer Projects**, die Sie über den Dialog **Extensions und Updates** installieren können (Menüeintrag **Extras|Extensions und Updates...**). Klicken Sie dort auf **Online** und geben Sie **Install** als Suchbegriff ein. Sie finden dann den Eintrag **Microsoft Visual Studio 2015 Installer Projects**, den Sie per Mausklick herunterladen und anschließend installieren können.

Weitergabe-Projekte

Visual Studio bietet, wenn Sie ein neues Projekt erstellen, einige Einträge im Bereich **Vorlagen|Andere Projekttypen|Visual Studio Installer** (siehe Bild 1).

Von diesen Projekttypen ist für unsere Zwecke nur einer interessant, nämlich **Setup Project**: Dieser erstellt eine Datei mit dem Namen **Setup.exe** sowie eine weitere namens **<Projektname>.msi**.

Aus zwei mach eins

Nun werden Sie sich vermutlich fragen, wie Sie die Anwendung auf Basis des weiterzugebenden Projekts und das Setup-Projekt zusammenbringen sollen. Das ist gar nicht so schwer: Wir öffnen einfach die Projektmappe mit dem weiterzugebenden Projekt (also die **.sln**-Datei) und fügen der Projektmappe das Setup-Projekt hinzu. Dazu klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf das Projektmappen-Element und wählen aus dem

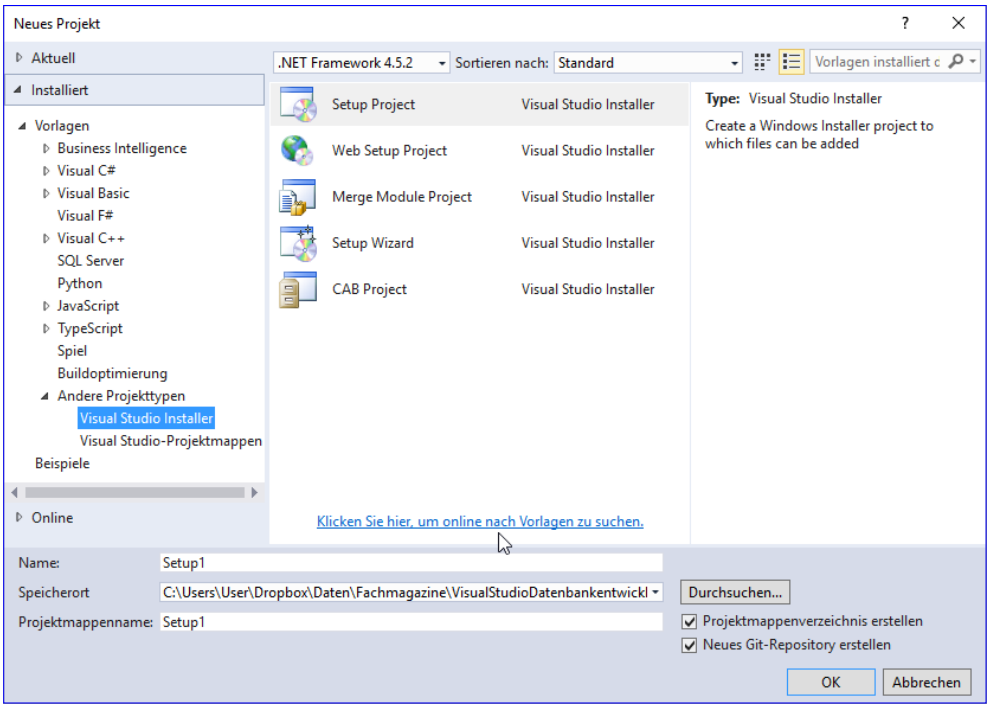


Bild 1: Weitergabe-Projekte

Kontextmenü den Eintrag **Hinzufügen|Neues Projekt** aus (siehe Bild 2). Im nun erscheinenden Dialog **Neues Projekt** wählen Sie dann die bereits weiter oben genannte Projektvorlage **Setup Project** aus und erstellen diese unter dem Namen **InstallBestellverwaltung** im Verzeichnis der Projektmappe, welches hier bereits voreingestellt sein dürfte (diese Bezeichnung wird als Name für die **.msi**-Datei verwendet und im Setup-Dialog als Titelzeile eingeblendet, also wählen Sie diesen sorgfältig aus). Im Gegensatz zu dem oben abgebildeten Dialog **Neues Projekt** enthält der vom Projektmappen-Explorer aus gestartete Dialog nicht das Textfeld **Projektmappenname**. Logisch, denn wir fügen ja einer Projektmappe ein neues Projekt hinzu.

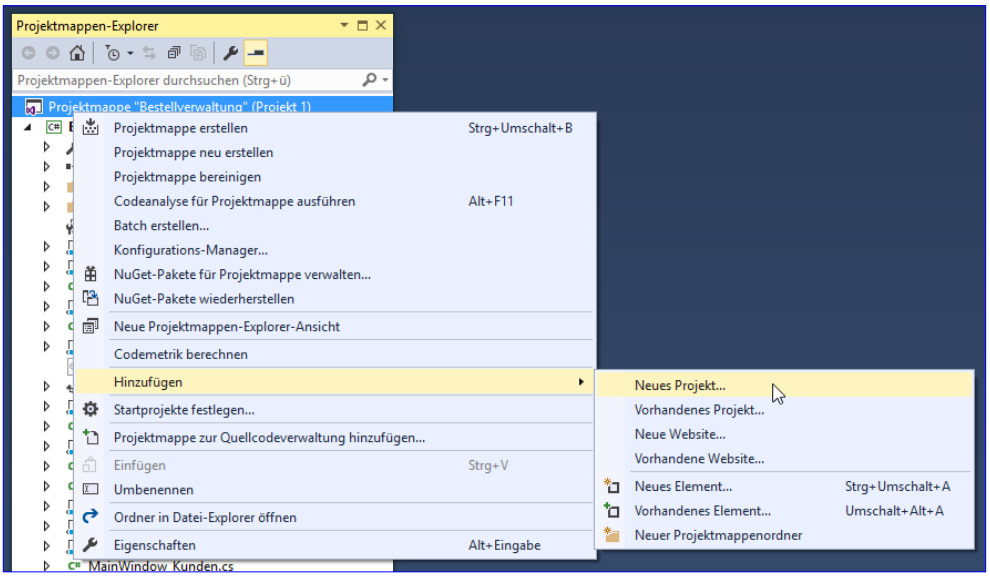


Bild 2: Hinzufügen eines Setup-Projekts

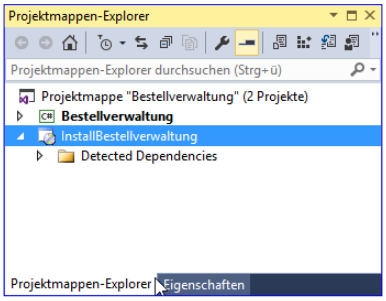


Bild 3: Zwei Projekte in einer Projektmappe

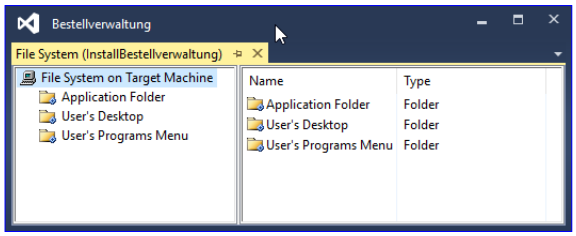


Bild 4: Der nach dem Erstellen des Setup-Projekts eingeblendete **FileSystem**-Editor.

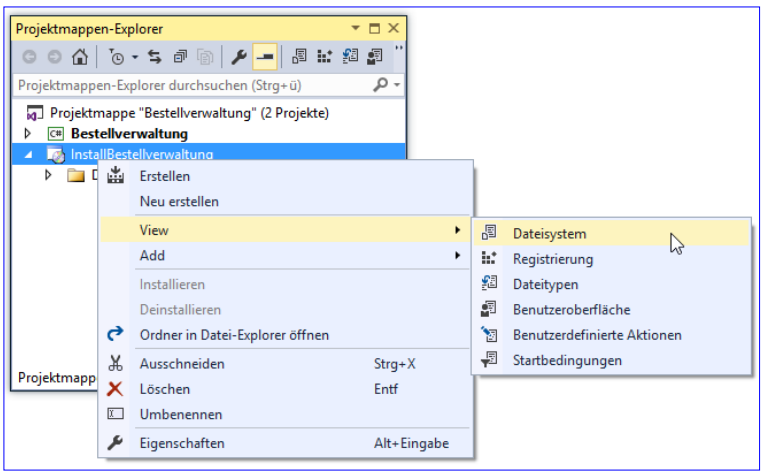


Bild 5: Aufruf der verschiedenen Editoren des Setups

Nach dem Erstellen des neuen Projekts erscheint dieses im Projektmappen-Explorer in der gleichen Ebene wie das Projekt selbst (siehe Bild 3).

Dateisystem-Editor

Nun erscheint der **FileSystem**-Editor (siehe Bild 4).

Sollten Sie diesen einmal versehentlich schließen, können Sie ihn über den Kontextmenü-Eintrag **View|Dateisystem** des Setup-Projekts wieder einblenden (siehe Bild 5). Dies gilt auch für die anderen Editoren des Setups, auf die wir später zu sprechen kommen.

Quelle hinzufügen

Nun fügen wir allerdings erst einmal die Information über die Dateien hinzu, die im Setup landen sollen. Dazu klicken Sie mit der rechten Maustaste auf den Eintrag **Application Folder** im Dateisystem-Editor und wählen dort den Kontextmenü-Befehl **AddlProjektausgabe** aus (siehe Bild 6).

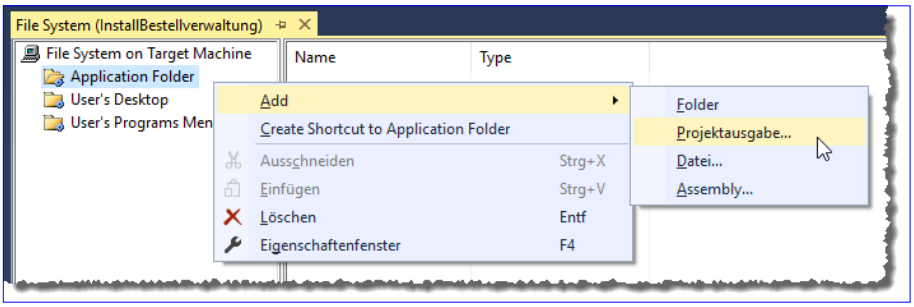


Bild 6: Hinzufügen der Projektausgabe

Damit öffnen Sie den Dialog aus Bild 7, der im oberen Kombinationsfeld den Namen des Projekts enthalten sollte, dessen Ausgabe in das Setup aufgenommen werden soll – hier also **Bestellverwaltung**. In der Liste darunter soll der Eintrag **Primäre Ausgabe** markiert sein. Klicken Sie auf **OK**, um den Dialog zu schließen.

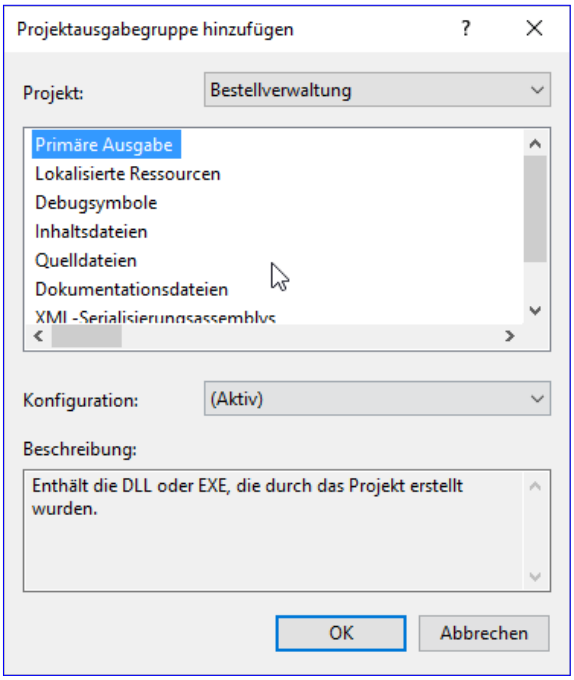


Bild 7: Projektausgabegruppe festlegen

Damit kommt Leben in die Sache. Sowohl im Projektmappen-Explorer, der nun einige Einträge unterhalb des bis dahin leeren Elements **Detected Dependencies** enthält, landen diese auch im Ordner **Application Folders** des **FileSystem**-Editors – nebst dem Element **Primäre Ausgabe from Bestellverwaltung (Active)** (siehe Bild 8).

Zielverzeichnis einstellen

Nun legen wir fest, wo auf dem Zielrechner die Anwendung landen soll. Dazu klicken Sie wiederum mit der rechten Maustaste auf den Eintrag **Application Folder** und wählen diesmal den Kontextmenü-

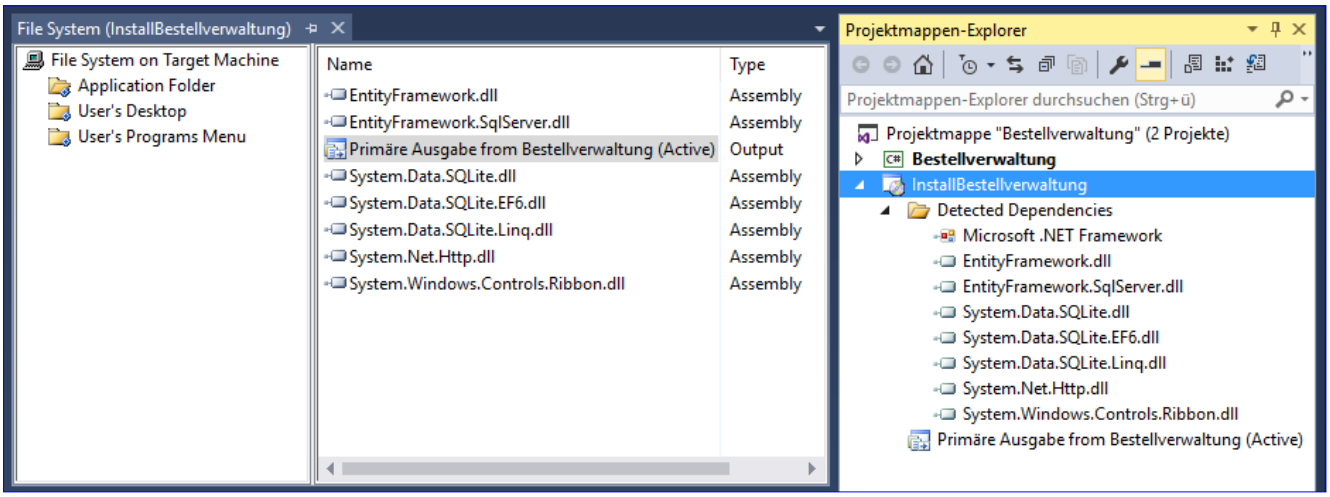


Bild 8: Gefüllter **Application Folder**

Eintrag **Eigenschaftenfenster** aus. Hier finden Sie die Eigenschaft **DefaultLocation** vor, die in unserem Beispiel standardmäßig den Wert **[ProgramFilesFolder][Manufacturer]\[ProductName]** enthält (siehe Bild 9). Die einzelnen Elemente werden bei der Installation ersetzt. Den Wert für **ProgramFilesFolder** ermittelt das Setup abhängig vom Zielsystem. Die beiden anderen Einstellungen, **Manufacturer** und **ProductName**, können Sie selbst in den Eigenschaften des Projekts festlegen.

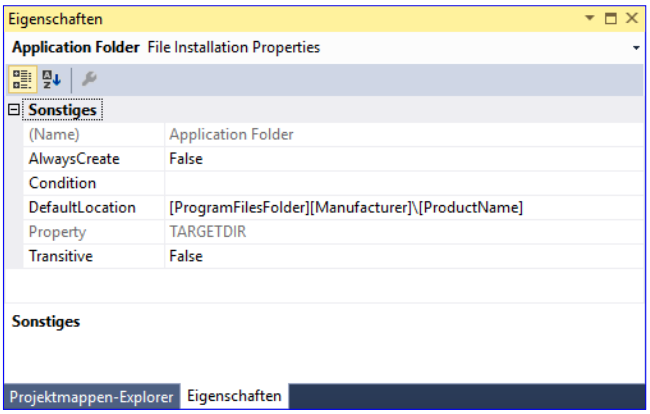


Bild 9: Zielverzeichnis festlegen

Dazu wechseln Sie im Projektmappen-Explorer zu dem Eintrag für das Install-Projekt (hier **InstallBestellverwaltung**). Aktivieren Sie mit **F4** die Anzeige der Eigenschaften, erscheint der Dialog aus Bild 10. Hier stellen Sie dann etwa die Eigenschaften **Manufacturer**, **ProductName** oder **Title** ein. Wir haben hier Werte verwendet, mit denen wir bei einem ersten Test schnell erkennen können, welche Eigenschaft wo im Setup landet – also etwa **Eigenschaften_Manufacturer** für **Manufacturer**.

Setup erstellen

Danach erstellen wir das Setup. Dazu speichern Sie vorsichtshalber alle Änderungen (Menüeintrag **Datei|Alles speichern**). Anschließend öffnen Sie mit dem Menüeintrag **Erstellen|Konfigurations-Manager...** den **Konfigurations-Manager**. Dieser zeigt die beiden Projekte der aktuellen Projektmappe an. Hier stellen Sie nun für beide Einträge unter **Konfiguration** den Wert **Release** ein und markieren jeweils die Option **Erstellen** (siehe Bild 11). Schließen Sie den Dialog dann wieder.

Wählen Sie dann den Menüeintrag **Erstellen|Projektmappe erstellen** aus. Nach dem Erstellen des Setups finden Sie im Ordner **...InstallBestellverwaltung\Release** die beiden Dateien **InstallBestellverwaltung.msi** und **setup.exe** (siehe Bild 12).

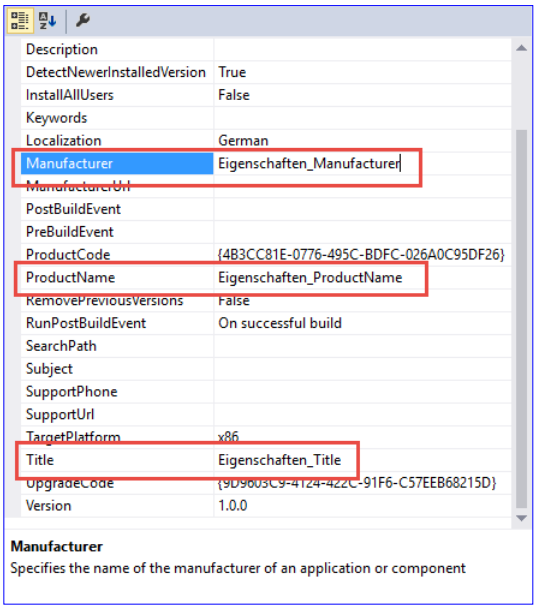


Bild 10: Einstellen der Setup-Eigenschaften

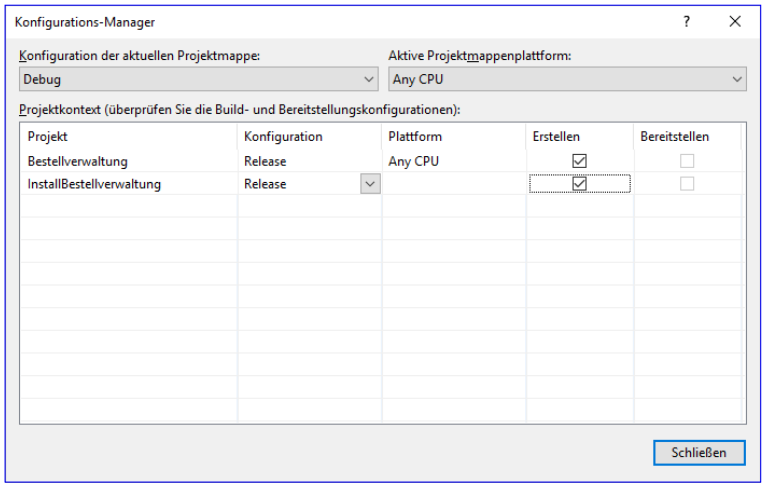


Bild 11: Konfigurieren der Erstellung

Unterschied .msi und setup.exe

Wozu benötigen Sie diese beiden Dateien? Aktuelle Windows-Systeme sind mit dem Windows Installer ausgestattet. Klicken Sie doppelt auf die **.msi**-Datei, wird diese mit dem Windows Installer geöffnet. Die **.msi**-Datei ist eine Datenbankdatei, welche Informationen über die zu installierende Anwendung und gegebenenfalls die Anwendung und weitere benötigte Dateien selbst enthält. Die Datei **setup.exe** ist eine Bootstrapper-Datei. Diese führt nicht die Installation durch, sondern prüft, ob die korrekte Version des Windows Installers auf dem Zielsystem installiert ist und ruft diese dann für die **.msi**-Datei auf.

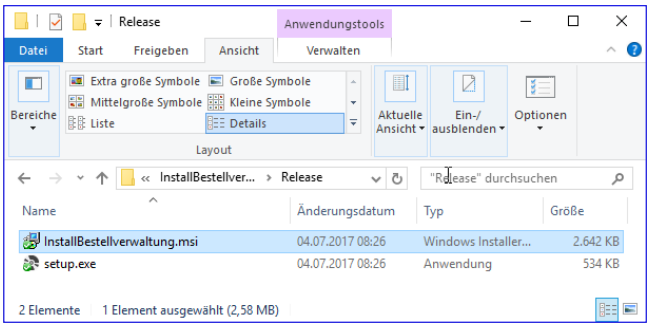


Bild 12: Setup-Dateien

Test der Installation

Nun schauen wir uns an, ob die Installation funktioniert und klicken dazu doppelt auf die **.msi**-Datei. Diese zeigt nun einen typischen Installationsdialog an (siehe Bild 13).

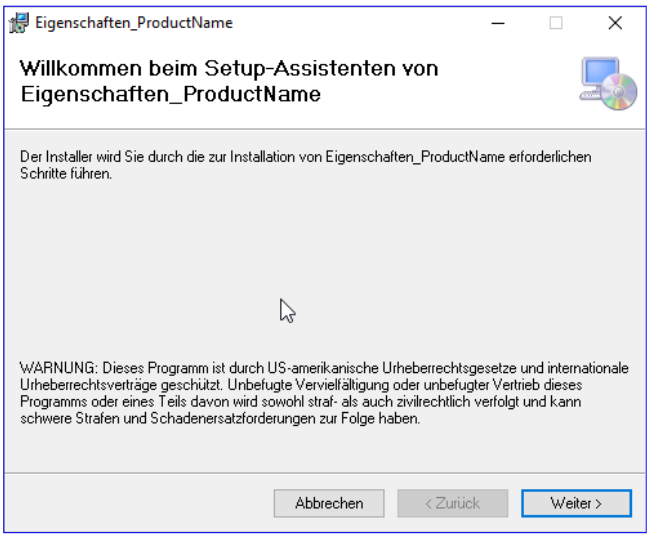


Bild 13: Erster Installationsschritt

Nach diesem ersten allgemeinen Schritt folgt gleich der Schritt mit den meisten auszuwählenden Optionen (siehe Bild 14). Hier legen Sie fest, in welches Verzeichnis die Anwendung installiert werden soll, können den Speicherbedarf abfragen und festlegen, ob die Anwendung für alle oder nur für den aktuellen Benutzer installiert werden soll.

Schließlich starten Sie im dritten Dialog die Installation (siehe Bild 15). Im Anschluss erscheinen dann noch die Abfrage der

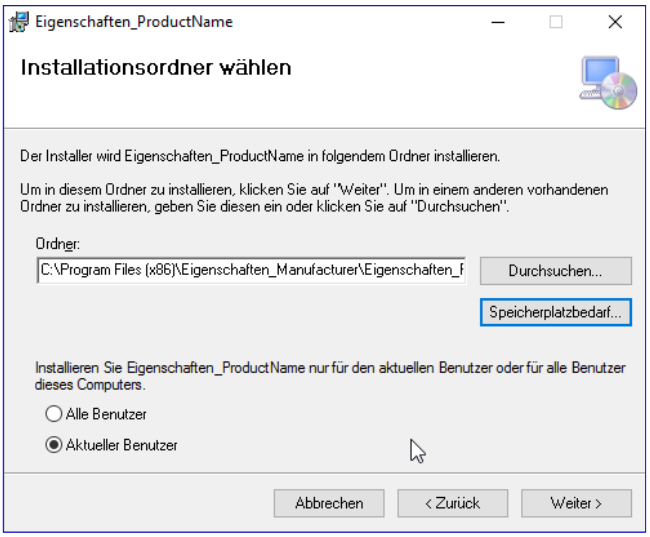


Bild 14: Auswahl des Zielverzeichnisses

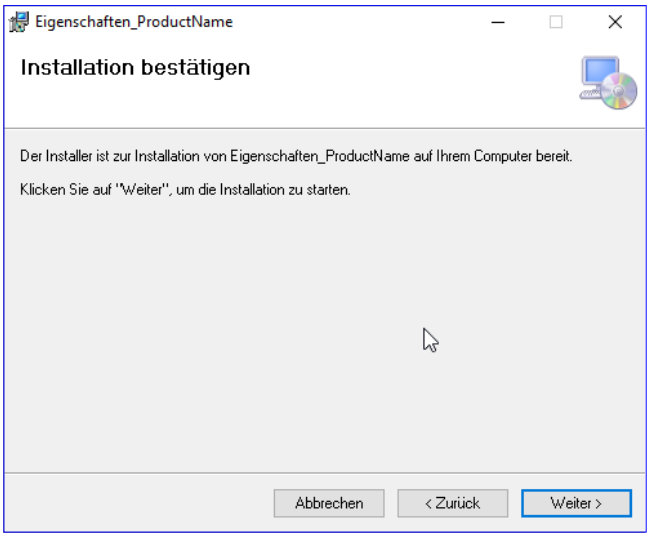


Bild 15: Abschluss der Installation

Tipps und Tricks

In den übrigen Artikeln lernen Sie immer wieder kleine Helferlein kennen, mit denen Sie gängige Probleme lösen. Schade nur, wenn Sie bei der Programmierarbeit mal wieder vor einem solchen Problem stehen und sich nicht mehr an die einfache Lösung erinnern können. Damit Sie diese bei Bedarf schnell wiederfinden, fassen wir diese Helferlein in der Reihe Tipps und Tricks nochmal für Sie zusammen.

WPF-Anwendung direkt wieder schließen

Eine WPF-Anwendung zeigt normalerweise direkt nach dem Start das Hauptfenster an, das in der Regel **MainWindow.xaml** heißt. Manchmal möchten Sie aber vor dem Anzeigen des Hauptfensters noch Prüfungen durchführen, die für den ordnungsgemäßen Ablauf der Anwendung nötig sind – zum Beispiel das Vorhandensein einer bestimmten Datei oder der Datenbank. Dies können Sie beispielsweise in der Konstruktor-Methode des Hauptfensters erledigen, also in der Code behind-Klasse **MainWindow.xaml.cs**. Wenn Sie hier feststellen, dass die Bedingungen für das Starten der Anwendung nicht gegeben sind, können Sie die Anwendung mit folgendem Befehl beenden:

```
App.Current.Shutdown();
```

Mehrzeilige Zeichenfolgen komfortabel eingeben

Wer unter VBA einmal versucht hat, eine mehrzeilige Zeichenfolge in einer String-Variablen zu speichern, musste beispielswiese diesen Weg gehen:

```
Dim strMehrzeilig As String
strMehrzeilig = "Erste Zeile" & vbCrLf & "Zweite Zeile"
```

Unter C# ist das viel einfacher. Sie nutzen einfach ein führendes Klammeraffe-Symbol (@) vor dem öffnenden Anführungszeichen und geben dann einfach den mehrzeiligen Text mit Zeilenumbrüchen ein:

```
string mehrzeilig = @"Erste Zeile
Zweite Zeile
Dritte Zeile";
```

C#-Methode vorzeitig verlassen

Wenn Sie zu Beginn oder mitten in einer Methode feststellen, dass beispielsweise bestimmte Bedingungen für das Weiterlaufen der Methode nicht erfüllt sind, werden Sie die Methode beenden wollen. Unter Access/VBA ging das ganz einfach mit **Exit Sub**/**Function**. Unter C# gibt es eine möglicherweise etwas unerwartete Variante: Dort verlassen Sie die Methode vorzeitig mit der Anweisung **return**. Ja, genau das **return**, mit dem Sie sonst den Wert zurückgeben, den eine Methode mit Rückgabewert liefern soll – nur diesmal ohne Rückgabewert.