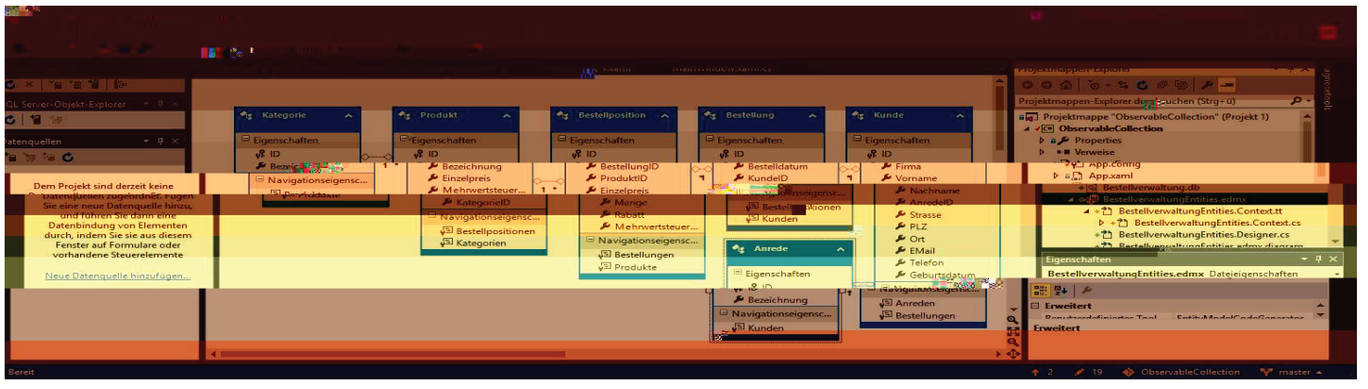
AUSGABE 05/2021

DATENBANK ENTWICKLER

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



TOP-THEMEN:

André Minhorst Verlag

Navigieren mit CollectionViewSource

Bisher haben wir oft mit Detailansichten von Datensätzen wie beispielsweise für Kunden oder Produkte gearbeitet, die zum Anlegen oder Bearbeiten eines einzelnen Datensatzes geeignet waren. Von Access kennen Sie die Möglichkeit, mit den Navigationsschaltflächen auch in solchen Detailformularen zu navigieren und von einem zum anderen Datensatz zu wechseln, ohne zwischendurch ein Übersichtsformular zu benötigen. In diesem Artikel wollen wir zeigen, wie Sie das unter WPF so abbilden können, wie es auch unter Access möglich ist. Dabei nutzen wir die Möglichkeiten der CollectionViewSource.

Ziel des Artikels

In diesem Artikel wollen wir einer Seite, die eine Detailansicht eines Datensatzes – hier eines Kunden – anzeigt, Navigationsschaltflächen wie von den Formularen von Access bekannt hinzufügen. Die WPF-Seite besteht dabei grob aus einem **Grid**-Element, das wiederum zwei weitere **Grid**-Element aufnimmt. Es definiert außerdem im Element **Window.Resources** ein **CollectionViewSource**-Element, das wir im Code behind mit den anzuzeigenden Daten füllen:

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="450" Width="800">
    <Window.Resources>
        ...
        <CollectionViewSource x:Key="kundeViewSource"></CollectionViewSource>
    </Window.Resources>
```

Danach folgt die Definition des übergeordneten **Grid**-Elements:

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

Das erste untergeordnete **Grid**-Element erhält als **DataContext** das als statische Ressource hinterlegte **CollectionViewSource**-Element sowie einige Zeilen für die Anzeige der einzelnen Felder und Spalten zur Aufteilung von Bezeichnungsfeldern und gebundenen Steuerelementen:

```
        <Grid DataContext="{StaticResource kundeViewSource}">
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto"></RowDefinition>
                ...
```

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

Danach folgen die Definitionen für Steuerelemente, die in den einzelnen Zeilen und Spalten angezeigt werden sollen – zum Beispiel für die ID und den Vornamen:

```
    <Label Content="ID:" Grid.Column="0" />
    <TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left"
        Text="{Binding ID, Mode=TwoWay}" Width="50" IsEnabled="False" BorderBrush="Transparent" />
    <Label Content="Vorname:" Grid.Column="0" Grid.Row="2" />
    <TextBox x:Name="txtVorname" Grid.Column="1" Grid.Row="2" Width="202"
        Text="{Binding Vorname, Mode=TwoWay, ValidatesOnDataErrors=True}" Margin="2,2,0,3" />
    ...
</Grid>
```

Das zweite untergeordnete **Grid**-Element enthält ein **StackPanel**-Element mit den vier Schaltflächen und dem Textfeld zur Anzeige der Position des Datensatzzeigers beziehungsweise zur Eingabe des anzuzeigenden Datensatzes, das wir im folgenden Abschnitt beschreiben.

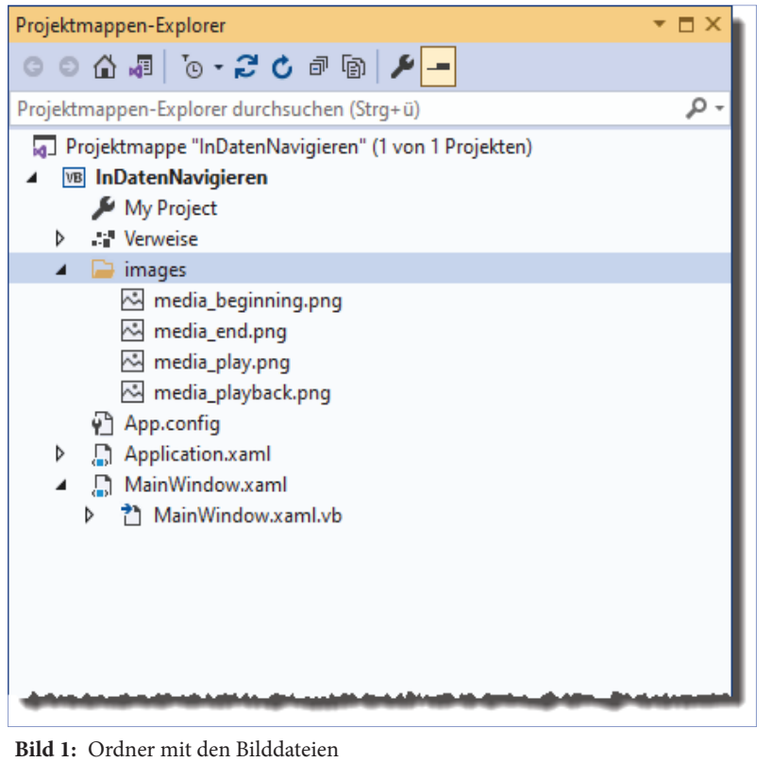


Bild 1: Ordner mit den Bilddateien

Aufbau der Navigationssteuerelemente

Die Navigationsschaltflächen wollen wir genau wie unter Access abbilden. Das heißt, dass wir von links nach rechts die folgenden Steuerelemente hinzufügen wollen:

- Schaltfläche zum Navigieren zum ersten Datensatz
- Schaltfläche zum Navigieren zum vorherigen Datensatz

- Textfeld zur Anzeige der aktuellen Position im Format **x von y**, wobei **y** die Gesamtzahl der angezeigten Datensätze enthält. In dieses Textfeld kann der Benutzer die Position eingeben, zu der er navigieren möchte.
- Schaltfläche zum Navigieren zum nächsten Datensatz
- Schaltfläche zum Navigieren zum letzten Datensatz

Die Steuerelemente organisieren wir in einem **StackPanel**-Steuerelement mit horizontaler Ausrichtung, das wir wie folgt definieren:

Die Bilddateien für die hier definierten **Image**-Elemente finden sie im Beispielprojekt im Ordner **images** (siehe Bild 1).

Der XAML-Code hierfür sieht wie folgt aus:

```
<Grid Grid.Row="1">
    <StackPanel Orientation="Horizontal" Grid.Row="10" Grid.ColumnSpan="4" >
        <Button x:Name="btnFirst" Click="btnFirst_Click" IsEnabled="{Binding FirstOrPreviousEnabled}">
            <Image Source="images/media_beginning.png" Width="24" Height="24"></Image>
        </Button>
        <Button x:Name="btnPrevious" Click="btnPrevious_Click"
            IsEnabled="{Binding FirstOrPreviousEnabled, UpdateSourceTrigger=PropertyChanged}">
            <Image Source="images/media_playback.png" Width="24" Height="24"></Image>
        </Button>
        <TextBox x:Name="txtNavigation" MinWidth="50"
            Text="{Binding MyCurrentPosition, UpdateSourceTrigger=LostFocus}"
            GotFocus="txtNavigation_GotFocus" KeyDown="txtNavigation_KeyDown"></TextBox>
        <Button x:Name="btnNext" Click="btnNext_Click"
            IsEnabled="{Binding NextOrLastEnabled, UpdateSourceTrigger=PropertyChanged}">
            <Image Source="images/media_play.png" Width="24" Height="24"></Image>
        </Button>
        <Button x:Name="btnLast" Click="btnLast_Click"
            IsEnabled="{Binding NextOrLastEnabled, UpdateSourceTrigger=PropertyChanged}">
            <Image Source="images/media_end.png" Width="24" Height="24"></Image>
        </Button>
    </StackPanel>
</Grid>
```

Für die Schaltflächen finden Sie das Attribut **IsEnabled**, das wir an die Eigenschaften **FirstOrPreviousEnabled** beziehungsweise **NextOrLastEnabled** gebunden haben. Wie diese Eigenschaften definiert sind, beschreiben wir weiter unten.

Der Entwurf des Fensters sieht anschließend wie in Bild 2 aus.

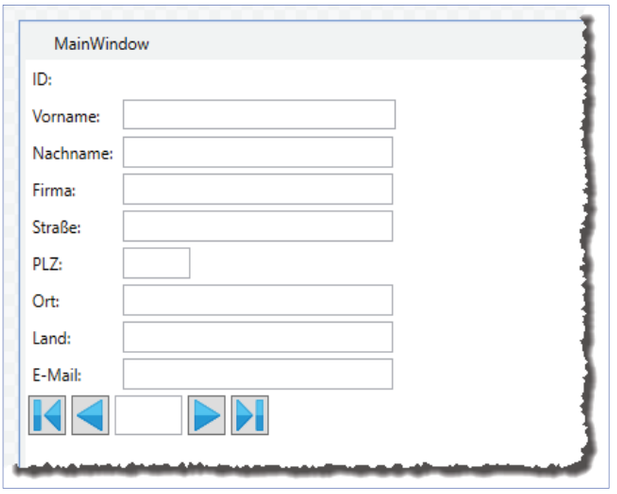


Bild 2: Entwurf des Fensters zur Anzeige der Kunden mit Navigationsschaltflächen

Anzeige der Daten

Damit überhaupt Daten im Fenster angezeigt werden, verwenden wir im Code behind-Modul eine Variable, mit der wir die **CollectionViewSource** referenzieren, die wir im XAML-Code als Ressource definiert haben:

```
Private cvsKunden As CollectionViewSource
```

Diese verbinden wir in der Konstruktor-Methode **New** mit der Ressource und stellen dann die Eigenschaft **Source** des **CollectionViewSource**-Elements auf den Wert der Funktion **GetKunden** ein:

```
Public Sub New()
    InitializeComponent()
    DataContext = Me
    cvsKunden = (Me.FindResource("kundeViewSource"))
    cvsKunden.Source = GetKunden()
End Sub
```

Die Funktion **GetKunden** erstellt eine **ObservableCollection** mit Elementen auf Basis der Klasse **Kunde**. Die Definition dieser Klasse sieht gekürzt wie folgt aus:

```
Partial Public Class Kunde
    Public Property ID As System.Int32
    Public Property Firma As System.String
    Public Property Vorname As System.String
    ...
End Class
```

Die Funktion **GetKunden** fügt mit der **Add**-Methode jeweils ein neues Element hinzu:

```
Private Function GetKunden() As ObservableCollection(Of Kunde)
    Dim Kunden As New ObservableCollection(Of Kunde)
    Kunden.Add(New Kunde() With {.ID = 1, .Firma = "Krahn GbR", .Vorname = "Adi", _
        .Nachname = "Stratmann", .Anrede = "Herr", .Strasse = "Kremser Straße 54", .PLZ = "10589", _
        .Ort = "Berlin", .Land = "Deutschland", .EMail = "adi@stratmann.de", _
        .Geburtsdatum = "20.02.1939", .Alter = 82, .Newsletter = -1})
```

```
    ... 'weitere Add-Aufrufe
    Return Kunden
End Function
```

Schaltflächen zum Navigieren in den Datensätzen

Damit können wir nun die Schaltflächen programmieren. Hier verwenden wir bereits eine Eigenschaft namens **MyCurrentPosition**, mit der wir die Position des Datensatzzeigers verwalten. Wie diese definiert ist, erläutern wir gleich im Anschluss.

Klickt der Benutzer auf die Schaltfläche **btnFirst**, sorgt die folgende Methode mit dem Aufruf von **MoveCurrentToFirst** für die Anzeige des ersten Datensatzes der aktuellen Ansicht des **CollectionViewSource**-Elements.

Außerdem stellt es den Wert der Eigenschaft **MyCurrentPosition** auf den Wert der aktuellen Position plus eins ein. Plus eins deshalb, weil **CurrentPosition** die Position 0-basiert zurückgibt – für die erste Position liefert sie also den Wert **0**:

```
Private Sub btnFirst_Click(sender As Object, e As RoutedEventArgs)
    cvsKunden.View.MoveCurrentToFirst()
    MyCurrentPosition = cvsKunden.View.CurrentPosition + 1
End Sub
```

Auf ähnliche Weise arbeitet die Methode für das **Click**-Ereignis der Schaltfläche **btnPrevious**. Diese verschiebt den Datensatzzeiger mit **MoveCurrentToPrevious** auf den vorherigen Datensatz. Die aktuelle Datensatzposition wird auf die gleiche Weise wie bei **btnFirst** in **MyCurrentPosition** gespeichert:

```
Private Sub btnPrevious_Click(sender As Object, e As RoutedEventArgs)
    cvsKunden.View.MoveCurrentToPrevious()
    MyCurrentPosition = cvsKunden.View.CurrentPosition + 1
End Sub
```

Die **Click**-Methoden für **btnNext** und **btnLast** sehen dementsprechend wie folgt aus:

```
Private Sub btnNext_Click(sender As Object, e As RoutedEventArgs)
    cvsKunden.View.MoveCurrentToNext()
    MyCurrentPosition = cvsKunden.View.CurrentPosition + 1
End Sub

Private Sub btnLast_Click(sender As Object, e As RoutedEventArgs)
    cvsKunden.View.MoveCurrentToLast()
    MyCurrentPosition = cvsKunden.View.CurrentPosition + 1
End Sub
```

Versionsverwaltung mit Git

Wie geht eigentlich Versionsverwaltung mit Visual Studio-Projekten? Wer Software mit Access entwickelt, musste sich darüber nicht viele Gedanken machen: Man kopiert einfach die komplette .accdb-Datei unter einem anderen Namen. Mittlerweile gibt es auch einige Lösungen von Drittherstellern, mit denen die einzelnen Objekte verwaltet werden können. Unter Visual Studio ist die Versionsverwaltung zum Glück direkt integriert. Visual Studio nutzt dafür die Quellcodeverwaltung Git. Welche Einsatzmöglichkeiten es gibt und wie Sie diese für Ihre eigenen Anforderungen einsetzen können, zeigt der vorliegende Artikel.

Das die Quellcodeverwaltung **Git** in Visual Studio integriert ist, sieht man beispielsweise an der Version 2019 direkt in der Menüleiste: Hier finden Sie den Eintrag **Git** mit einigen Unterpunkten vor (siehe Bild 1). In den folgenden Abschnitten erläutern wir allerdings erst einmal, in welchen Konstellationen Sie Git für Ihre Zwecke nutzen können.

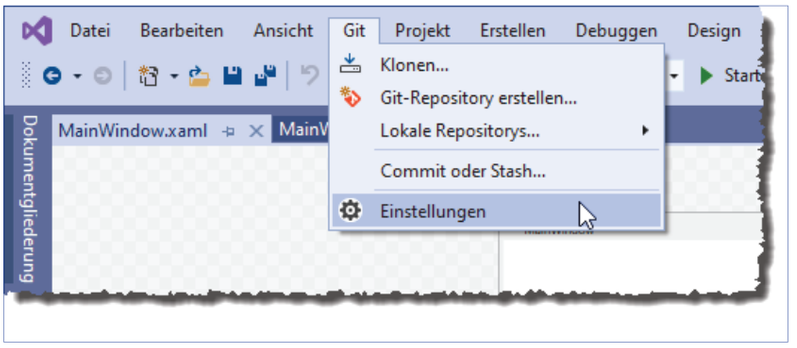


Bild 1: Git im Menüsystem von Visual Studio

Dabei verwenden wir die Begriffe **Quellcodeverwaltung** und **Versionsverwaltung** übrigens synonym.

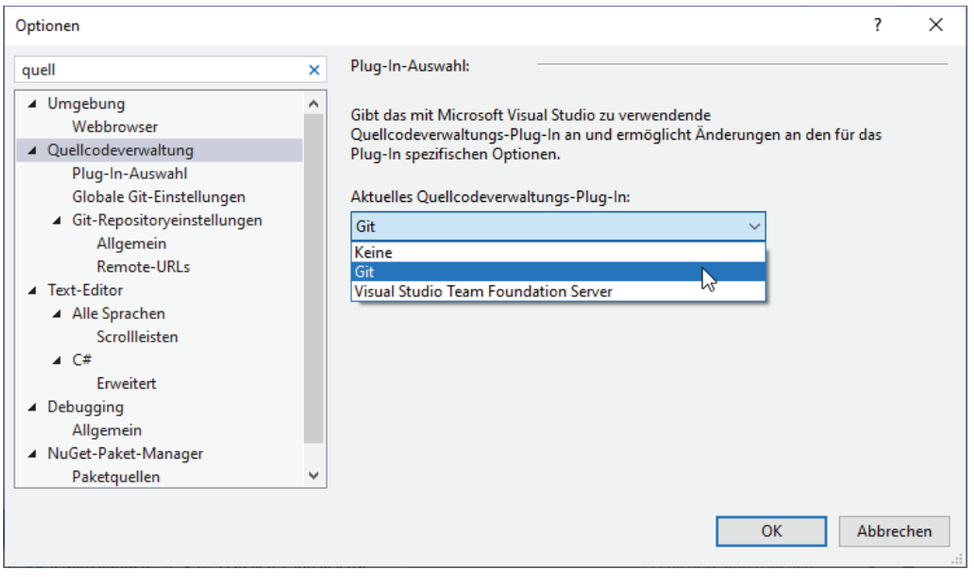


Bild 2: Quellcodeverwaltungs-Optionen

Quellcodeverwaltung festlegen

In den Optionen von Visual Studio können Sie unter Quellcodeverwaltung festlegen, welches Tool zur Quellcodeverwaltung eingesetzt werden soll (siehe Bild 2).

Unter Visual Studio 2019 ist auf unserem Testsystem **Git** bereits voreingestellt.

Einsatzzwecke der Versionsverwaltung mit Git

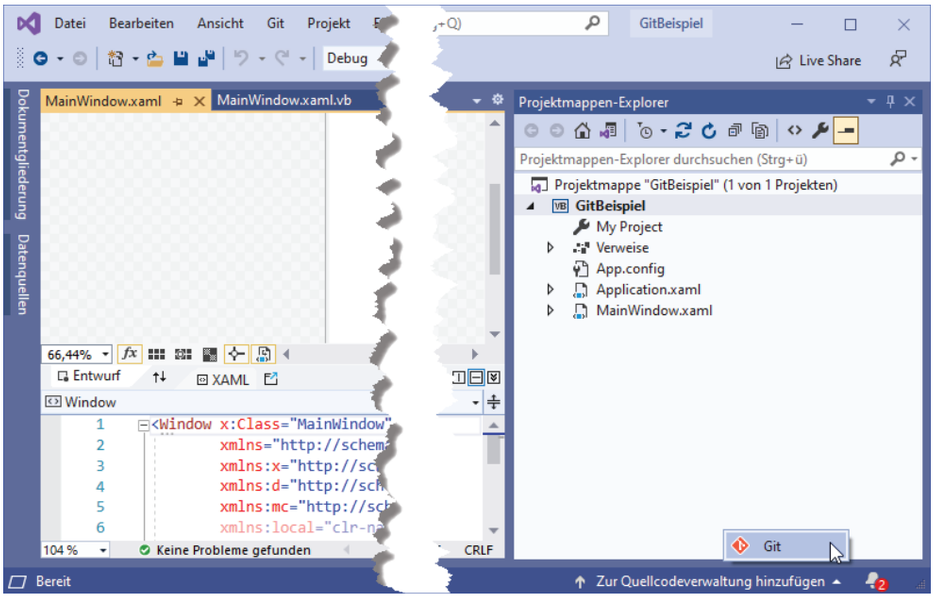


Bild 3: Hinzufügen zu Git

Git wurde 2005 vom Erfinder von Linux, Linus Thorvalds, entwickelt. Die eigentliche Idee war damals, dass man die unterschiedlichen Versionen einer Software auf einem Server speichert, damit alle an dem Projekt beteiligten Entwickler darauf zugreifen konnten. Das unter anderem zu dem Zweck, jederzeit eine aktuelle Version des Projekts herunterladen zu können und somit immer auch die Änderungen der anderen Entwickler präsent zu haben, aber auch dazu, die eigenen Änderungen mit den anderen Entwicklern zu teilen. Gegebenenfalls folgt hier noch ein Zwischenschritt, in dem die Änderungen noch geprüft werden, bevor sie in die Hauptversion übernommen werden.

Sie können Git aber auch einfach nutzen, um die verschiedenen Versionsstände Ihrer eigenen Projekte auf dem lokalen Rechner zu verwalten. Weiter oben haben wir bereits das Beispiel von Access genannt: Hier konnte man eigentlich nur die komplette Datenbank sichern, um gegebenenfalls zu einem bestimmten Versionsstand zurückspringen zu können. Bei Visual Studio-Produkten ist es selbst ohne Quellcodeverwaltung bereits möglich, einfach den Ordner mit dem Projekt zu kopieren und sich nach Bedarf eine ältere Version einer oder mehrerer Dateien in den aktuellen Projektordner zu holen. Mit Git wird das alles noch viel einfacher – wie das gelingt, zeigen die folgenden Abschnitte.

Projekt zu Git hinzufügen

Wenn Sie ein bestehendes Projekt haben und dieses mit der Quellcodeverwaltung **Git** verwalten möchten, finden Sie den notwendigen Befehl direkt unten rechts im Hauptfenster von Visual Studio (siehe Bild 3). Klicken Sie auf **Zur Quellcodeverwaltung hinzufügen**, erscheint eine Liste der verfügbaren Quellcodeverwaltungen – im Screenshot also nur **Git**.

Wählen Sie diesen Eintrag aus, erscheint als Nächstes der Dialog **GIT-Repository erstellen** aus Bild 4.

Hier wollen wir fürs Erste nur ein lokales Repository anlegen. Daher aktivieren wir links den Eintrag **Nur lokal**. Dies vereinfacht den Dialog stark, sodass wir nur noch den Pfad für das lokale Git-Repository angeben müssen. Hier wollen wir einfach das Hauptverzeichnis des Projekts nutzen.

Ein Klick auf die Schaltfläche **Erstellen** legt das Repository an und fügt bereits alle Dateien des Projekts hinzu. Der Projektmappen-Explorer präsentiert sich nun leicht verändert und auch in dem Bereich des Hauptfensters von Visual Studio, wo sich zuvor der Befehl **Zur Quellcodeverwaltung hinzufügen** befunden hat, zeigen sich einige neue Elemente (siehe Bild 5).

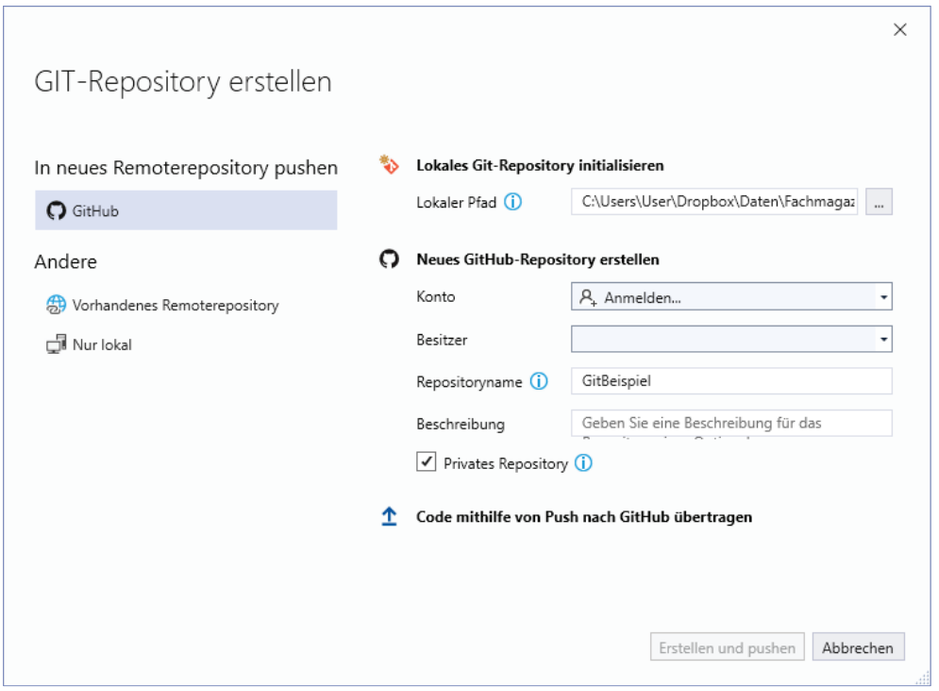


Bild 4: Erstellen eines Git-Repositorys

Hier sehen Sie zum Beispiel, wie das Repository heißt und dass Sie sich aktuell in einem sogenannten Branch namens **master** befinden (mehr dazu weiter unten). Klicken Sie auf master und zeigen für den nun erscheinenden Eintrag master das Kontextmenü an, finden Sie unter anderem einen Befehl namens Verlauf anzeigen (siehe Bild 6). Mit einem Klick auf diesen Befehl zeigen Sie alle bisherigen Versionen des ganzen Projekts an. Später werden Sie sehen, dass Sie auch die Versionen für einzelne Dateien

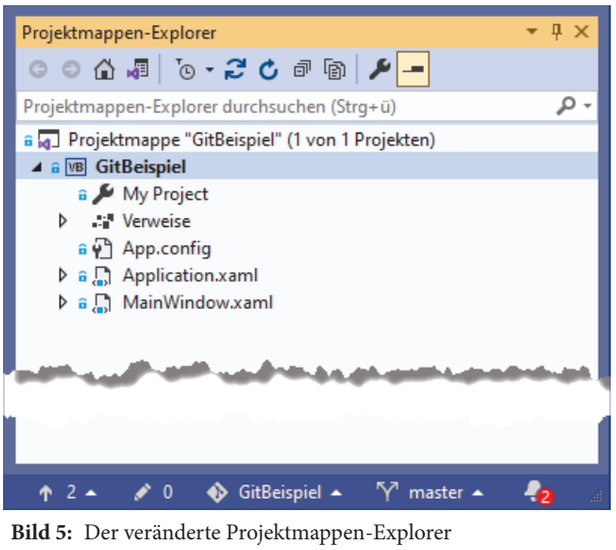


Bild 5: Der veränderte Projektmappen-Explorer

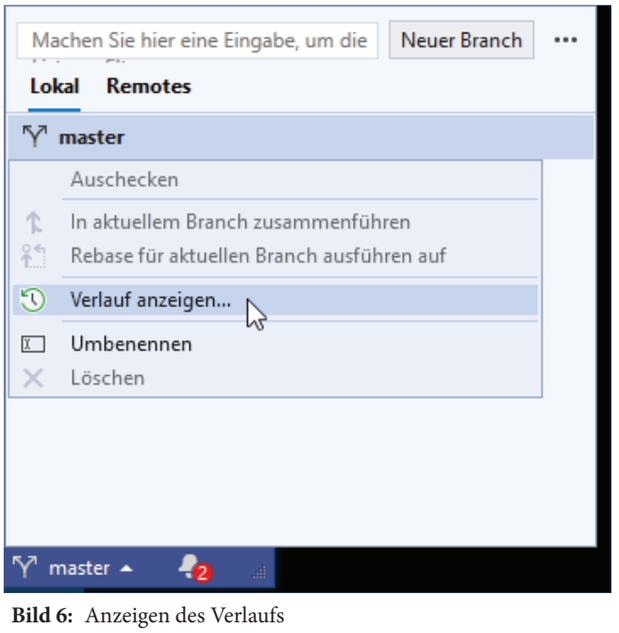


Bild 6: Anzeigen des Verlaufs

anzeigen können. An diesem Beispiel zeigen wir dann auch die Liste der bisher erstellten Versionen.

Nun zu den Einträgen der Projektmappe: Hier finden wir jeweils ein Schloss-Symbol vor, das beim Überfahren mit der Maus den Text **Eingecheckt** anzeigt. Ist dieses Symbol vorhanden, sehen Sie im Projektmappen-Explorer die Version des Elements, wie es auch im Repository eingecheckt ist. Eingecheckt heißt in diesem Zusammenhang, dass die aktuelle Version dort gespeichert ist.

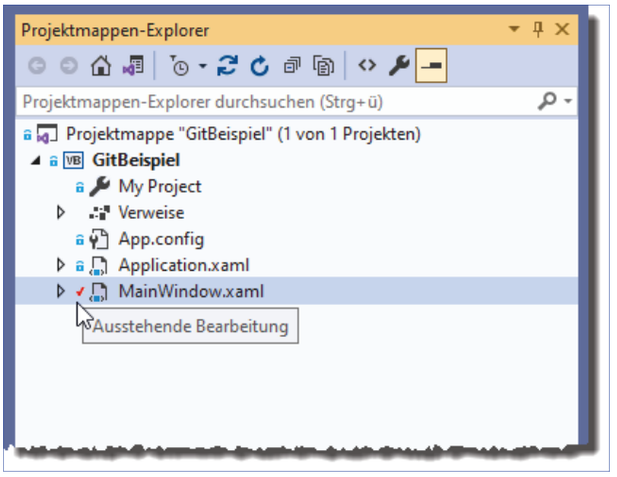


Bild 7: Ein Element mit ausstehender Bearbeitung

Eine Änderung des Symbols eines Elements erhalten wir beispielsweise, wenn wir seinen Code ändern. Dann wandelt Visual Studio das Symbol in ein rotes Häkchen um und der angezeigte Text lautet **Ausstehende Bearbeitung** (siehe Bild 7).

Wenn Sie die neue Version nun einchecken wollen, wählen Sie den Kontextmenü-Eintrag **Git|Commit oder Stash** für dieses Element aus (siehe Bild 8).

Dies ruft den Bereich aus Bild 10 hervor. Hier geben Sie in das Textfeld mit dem Text **Nachricht eingeben** einen Text ein, der die aktuellen Änderungen beschreibt. Auf diese Weise können Sie später nachvollziehen, welche Änderungen dieses Commit enthält. Klicken Sie dann auf **Commit für alle**, wird der Bereich geschlossen und das geänderte Element erhält im Projektmappen-Explorer das Icon **Eingecheckt**.

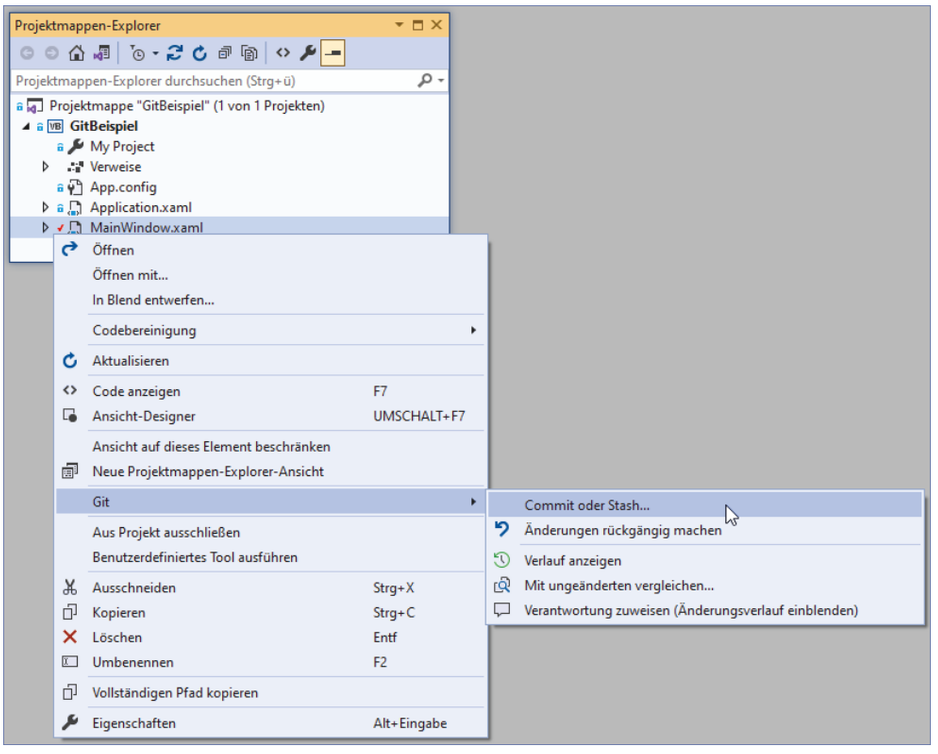


Bild 8: Element einchecken

Was ist passiert?

Durch das Einchecken oder Commit einer oder

mehrerer Dateien mit Änderungen sorgen Sie für die folgenden Änderungen:

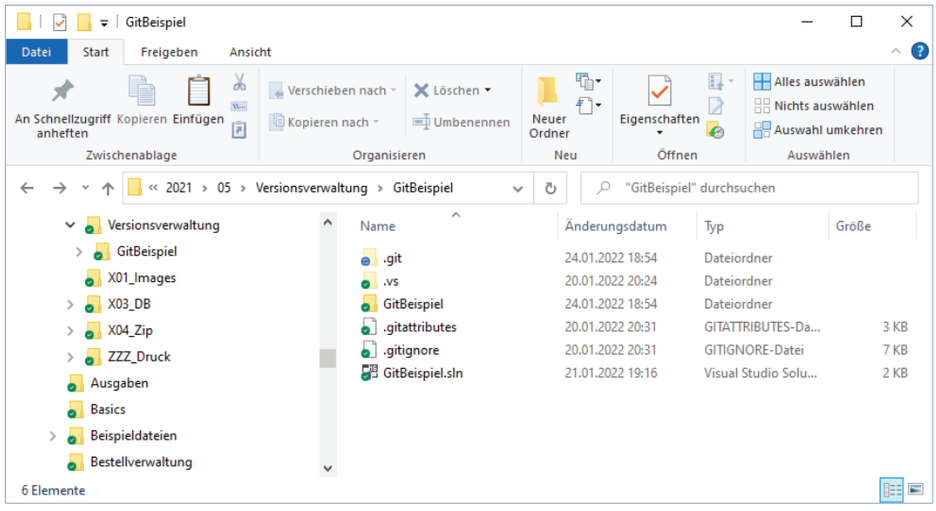


Bild 9: Elemente des lokalen Repositorys

- Die Datei wieder als **Eingecheckt** markiert.
- Im Repository sind nun zwei Versionsstände dieser Datei hinterlegt.
- Es liegen einige neue Elemente im Dateisystem vor (siehe Bild 9).

Die Datei **.gitignore** enthält beispielsweise die Namen aller Dateien, die von der Quellcodeverwaltung ignoriert werden sollen.

Hierzu gehören beispielsweise die beim Debuggen erstellten Dateien oder auch benutzerspezifische Files.

Und Sie können nun verschiedene Aktionen anstoßen:

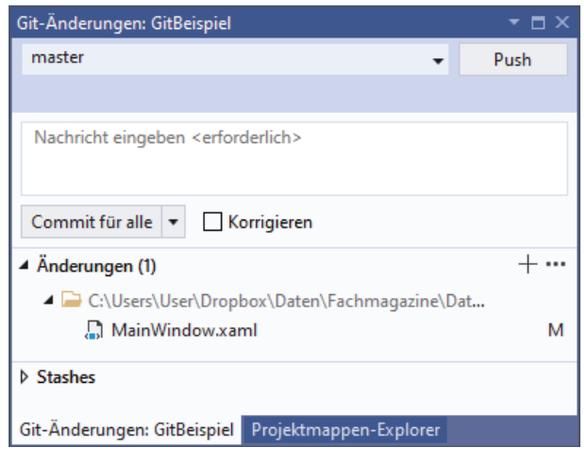


Bild 10: Einstellungen für das Commit

- Anzeigen des Verlaufs
- Anzeigen der Unterschiede zwischen den verschiedenen Versionen
- Wiederherstellen älterer Versionen

Verlauf ansehen

Der Verlauf listet alle Versionen einer Datei auf.

Um diese anzuzeigen, klicken Sie mit der rechten Maustaste auf die Datei im Projektmappen-Explorer, die Sie untersuchen wollen, und wählen dort

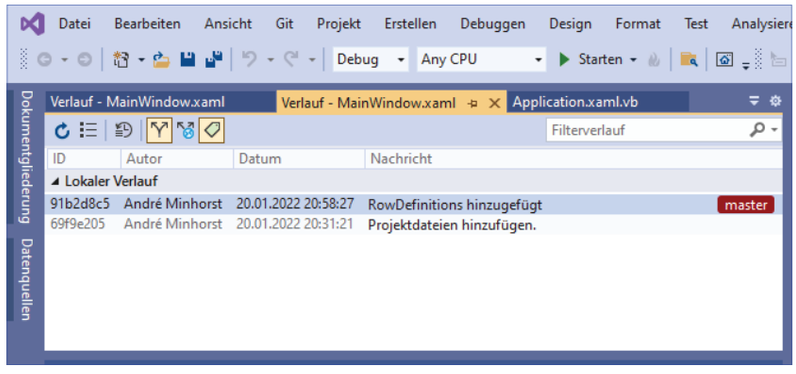


Bild 11: Anzeige des bisherigen Verlaufs

den Eintrag **Git|Verlauf anzeigen** aus. Der nun erscheinende Bereich **Verlauf** zeigt die bisherigen Versionen an (siehe Bild 11).

Die einfachste Methode, hiermit zu arbeiten, ist ein Doppelklick auf einen der Einträge. Dies öffnet ein Codefenster mit dem Code der jeweiligen Version. Diese Versionen können, auch wenn es sich um die aktuellste Version handelt, nicht bearbeitet werden. Viel interessanter ist es natürlich, die Unterschiede zwischen zwei Versionen aufzuzeigen.

Mit einem Rechtsklick auf die aktuelle Version erhalten Sie einige Optionen, unter anderem die Möglichkeit, diese Version mit der vorherigen Version zu vergleichen (siehe Bild 12).

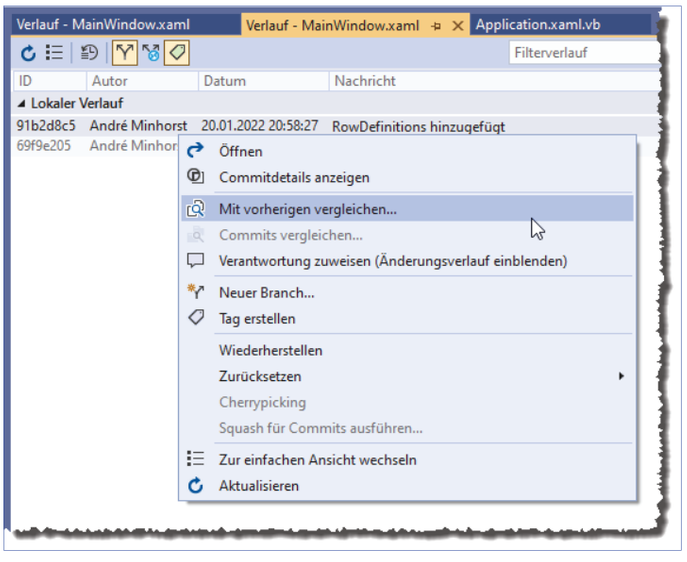


Bild 12: Vergleich mit der vorherigen Version

Unterschiede anzeigen

Die Unterschiede erscheinen dann in einem weiteren Bereich namens **Unterschied** (siehe Bild 13). Die Unterschiede zwischen den verschiedenen Versionen sind entsprechend markiert.

Mit anderer Version als der vorherigen vergleichen

Sie werden schnell mehr als zwei Versionen einer Datei commited haben, sodass Sie sich vermutlich fragen, wie Sie beispielsweise die erste mit der ersten Version vergleich können, wenn das Kontextmenü nur den Eintrag **Mit vorherigen vergleichen...** anbietet. Auch das ist schnell gelöst: Dazu markieren Sie einfach

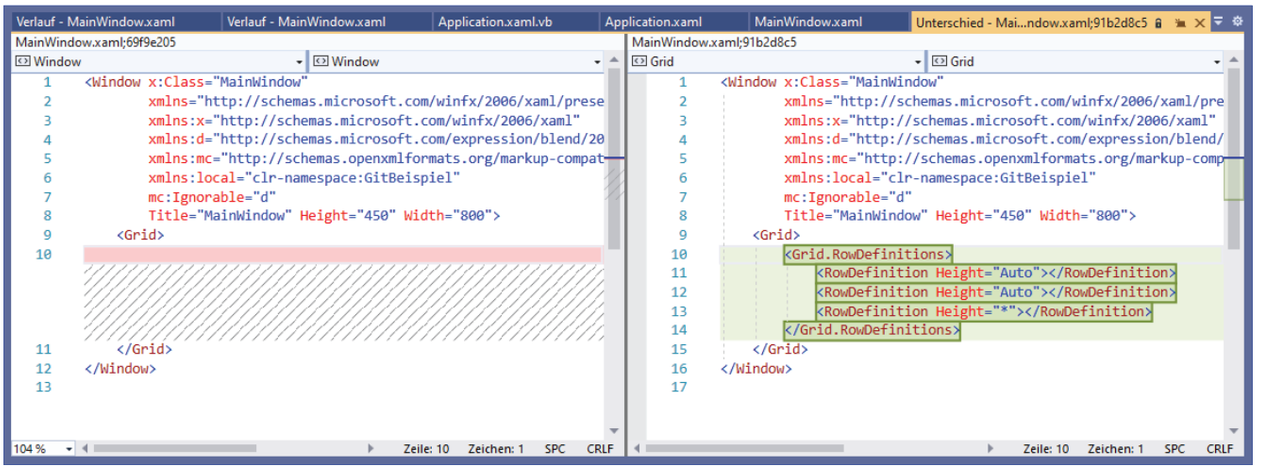


Bild 13: Anzeige der Unterschiede zwischen zwei Versionen

DATENBANK ENTWICKLER

Beispieldaten generieren mit Bogus

Wenn Sie eine Anwendung entwickeln, können Sie die meisten Funktionen nur testen, wenn die zugrunde liegende Datenbank bereits Beispieldaten enthält. Ausnahmen sind beispielsweise Formulare zum Anlegen neuer Datensätze – hier legen Sie die Beispieldatenbank selbst an. In allen anderen Fällen kann es nicht schaden, ein paar Beispieldatensätze in den jeweiligen Tabellen bereitzustellen. Oft reicht es aus, das Anlegen einiger Elemente wie Kunden oder Produkte hart im Code zu verdrahten. Wenn Sie allerdings flexibel schnell für verschiedene Datenmodelle und Anwendungen Beispieldaten benötigen, reduzieren Sie den Aufwand zum Zusammenstellen der Beispieldaten zwar auf das einmalige Schreiben des Codes. Es macht aber auch keinen Spaß, sich dann Informationen wie Teststraße, Beispielfirma und Co. aus den Fingern zu saugen. Um dies zu automatisieren und gegebenenfalls auch größere Mengen an Beispieldaten zu generieren, gibt es spezielle Tools. Eines davon heißt Bogus – und diese stellen wir in diesem Artikel im Detail vor.

Für Menschen wie mich, die immer wieder frische Daten für die Beispiele für die Artikel und Bücher benötigen, ist ein Tool zum automatischen Generieren von Beispielcode natürlich Gold wert. Aber auch Sie können sich vermutlich eine Menge Arbeit sparen, wenn Sie die Tabellen Ihrer Anwendung vor dem Testen schnell mit passenden Beispieldaten füllen können.

Wie für die meisten Anwendungsfälle, die Sie mit Visual Studio abarbeiten, finden sich auch hier passende Erweiterungen im NuGet-Paket-Manager. Das Paket, das wir in diesem Artikel vorstellen wollen, heißt Bogus und bietet bereits in der kostenlosen Variante sehr viele verschiedene mögliche Generatoren für Daten unterschiedlichster Art an.

Um zuvor einmal die grundlegende Idee zu skizzieren: Wir gehen davon aus, dass Sie in der zu programmierenden Anwendung beispielsweise eine Tabelle namens **Kunden** verwenden, dessen Einträge im Entity Data Model der Anwendung jeweils in einem Element namens **Kunde** gespeichert werden. Das Tool, in diesem Fall Bogus, soll uns eine Möglichkeit bieten, die Eigenschaften neu erstellter Elemente des Typs **Kunde** mit Werten zu füllen.

Wenn Kunde also beispielsweise Eigenschaften wie **Anrede**, **Vorname**, **Nachname**, **Strasse**, **PLZ** und **Ort** enthält, dann müsste das Tool Funktionen bereitstellen, mit denen Werte für genau diese Felder erzeugt werden können und die nach dem Zufallsprinzip ausreichend neue Werte liefern.

Noch spannender wird es an der Stelle, wo wir verknüpfte Tabellen haben – zum Beispiel **Kunden** und **Bestellungen**. Im einem Entity Data Model haben wir dann zwei Klassen namens **Kunde** und **Bestellung**, deren Eigenschaften wir nacheinander füllen müssen. Allerdings müssen wir zuerst die Kunden erstellen und wenn wir in einer Bestellung den Kunden referenzieren, der diese Bestellung aufgegeben hat, müssen wir auf einen der zuvor erstellten Kunden verweisen. Auch diese Möglichkeit bietet das hier verwendete Tool **Bogus**.

Beispielanwendung

Um die Funktionen von Bogus schnell und komfortabel testen zu können, nutzen wir **LINQPad5**. Hier können wir Code schnell eingeben und ausführen, ohne immer direkt ein Projekt erstellen zu müssen. Wenn Sie **LINQPad5** geöffnet haben, sind zwei Schritte nötig: der Wechsel zu einem passenden Query-Typ, hier **VB Program**, und das Hinzufügen von Bogus zur Query.

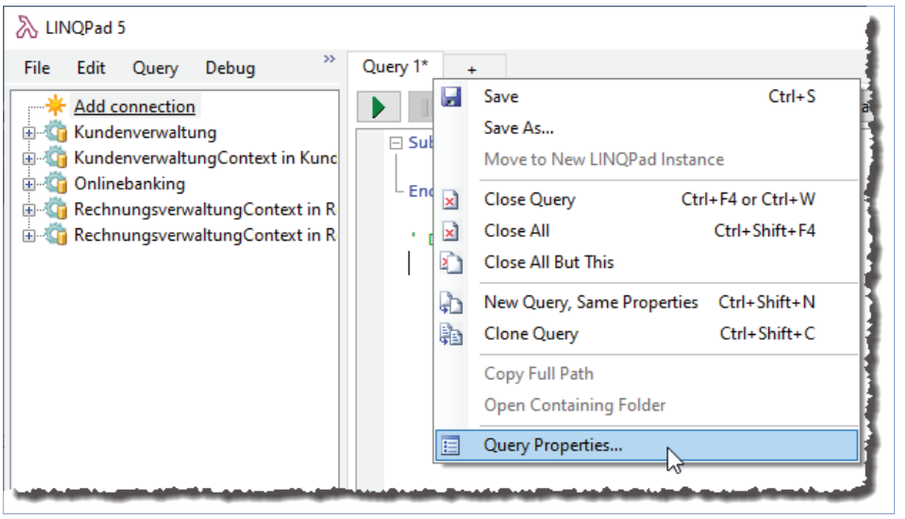


Bild 1: Aufrufen der Query-Eigenschaften

Bogus zur Query hinzufügen

Um das Bogus-Paket zur Query hinzuzufügen, klicken Sie mit der rechten Maustaste auf den Registerreiter für die Query. Dort wählen Sie den Eintrag **Query Properties...** aus (siehe Bild 1).

Dies öffnet den Dialog **Query Properties**. Hier klicken Sie unten auf die Schaltfläche **Add NuGet...**, was den Dialog **LINQPad NuGet Manager** öffnet. Hier geben Sie im mittleren Bereich den Suchbegriff **Bogus** ein und klicken dann für den obersten Eintrag auf die Schaltfläche **Add to Query** (siehe Bild 2). Schließen Sie den Dialog, finden Sie den Eintrag **Bogus** nun im Dialog **Query Properties** im Bereich **Additional Reference** vor.

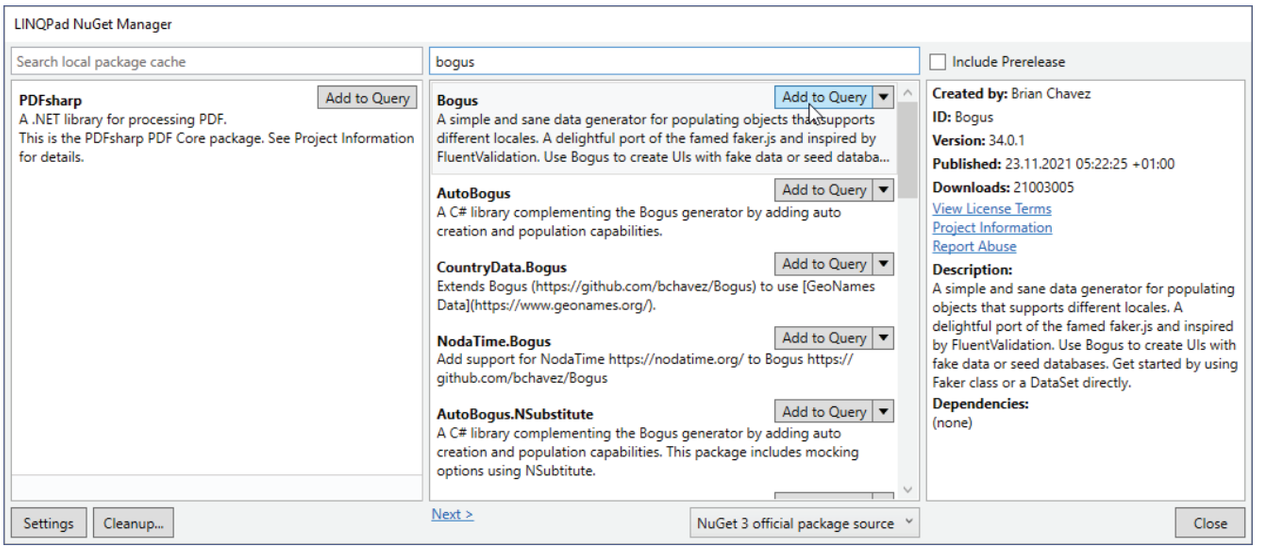


Bild 2: Hinzufügen des Pakets **Bogus**

Beispielkunde erstellen

Der Rahmen für die ersten Experimente mit Bogus ist eine einfache Kundenklasse. Diese definieren Sie wie folgt unterhalb der in der LINQPad-Query vorgegebenen Methode **Main**:

```
Public Class Kunde
    Property Vorname As String
    Property Nachname As String
    Property Strasse As String
    Property PLZ As String
    Property Ort As String
    Property Land As String
    Property EMail As String
    Property Telefon As String
End Class
```

In der Methode **Main** werden wir gleich die Beispielmethoden aufrufen. Die erste heißt **KundeAnlegen**:

```
Sub Main
    KundeAnlegen
End Sub
```

Bogus nutzen

Die einfachste Möglichkeit, Bogus zu verwenden, ist das Initialisieren einer Objektvariablen auf Basis des Objekts **Bogus.Faker** und das anschließende Erzeugen der verschiedenen Zufallsdaten.

Das folgende Beispiel erledigt das und gibt so beispielsweise ein zufällig ermitteltes Land aus:

```
Public Sub AndereBeispiele
    Dim objFaker As New Bogus.Faker()
    Debug.Print (objFaker.Address.Country)
End Sub
```

Auf Basis dieses Beispiels könnten Sie im Grunde schon die Eigenschaften von neu erzeugten Objekten füllen. Es gibt jedoch noch praktischere Methoden. Mit der soeben vorgestellten Methode können Sie sich allerdings schon einmal durch die einzelnen Klassen und deren Eigenschaften arbeiten und sich die Möglichkeiten anschauen. Weiter unten finden Sie eine Beschreibung der einzelnen Klassen.

Objekte mit Bogus füllen

Nun schauen wir uns in der Methode **KundeAnlegen** am Beispiel eines neuen Kunden an, wie Sie Bogus nutzen können. Dabei erstellen wir zuerst ein neues **Kunde**-Objekt und ein neues Objekt des Typs **Bogus.Faker** für unsere Klasse **Kunde**:

```
Public Sub KundeAnlegen
    Dim Beispielkunde As New Kunde
    Dim objFaker As New Bogus.Faker(Of Kunde)
```

Das **Faker**-Objekt ist unser eigentlicher Generator. Für diesen legen wir über die **RuleFor**-Eigenschaft fest, welche Eigenschaft des Zielobjekts mit welcher Eigenschaft des Fakers gefüllt wird. Die **RuleFor**-Definitionen legen wir einfach in einer langen Zeile jeweils durch einen Punkt getrennt hintereinander an – der Übersichtlichkeit halber haben wir hier Zeilenumbrüche eingefügt. Die erste Regel besagt beispielsweise, dass wir die Eigenschaft **Vorname** mit der Eigenschaft **Name.FirstName** des Fakers füllen wollen, die weiteren Regeln legen die Quelle für die übrigen Eigenschaften fest:

```
    objFaker.RuleFor(Function(k) k.Vorname, Function(f) f.Name.FirstName) _
        .RuleFor(Function(k) k.Nachname, Function(f) f.Name.LastName) _
        .RuleFor(Function(k) k.Strasse, Function(f) f.Address.StreetAddress) _
        .RuleFor(Function(k) k.PLZ, Function(f) f.Address.ZipCode) _
        .RuleFor(Function(k) k.Ort, Function(f) f.Address.City) _
        .RuleFor(Function(k) k.Land, Function(f) f.Address.Country) _
        .RuleFor(Function(k) k.EMail, Function(f) f.Internet.Email) _
        .RuleFor(Function(k) k.Telefon, Function(f) f.Phone.PhoneNumber)
```

Dabei nutzen wir nicht nur die **Name**-Klasse (für **FirstName** und **LastName**), sondern auch die **Address**-Klasse (für alle Adress-relevanten Daten) sowie die **Internet**-Klasse (für die E-Mail-Adresse aus der Eigenschaft **Email**) und die **Phone**-Klasse (für die Telefonnummer aus **PhoneNumber**). Danach folgt das eigentliche Erstellen. Hier rufen wir die **Generate**-Methode des **Faker**-Objekts auf und schreiben das Ergebnis vom Typ **Kunde** in die Variable **Beispielkunde**:

```
    Beispielkunde = objFaker.Generate
```

Die Werte der Eigenschaften des neuen **Kunde**-Objekts schreiben wir danach wie folgt in den Debug-Bereich:

```
    Debug.Print("Vorname: " & Beispielkunde.Vorname)
    Debug.Print("Nachname: " & Beispielkunde.Nachname)
    Debug.Print("Strasse: " & Beispielkunde.Strasse)
    Debug.Print("PLZ: " & Beispielkunde.PLZ)
    Debug.Print("Ort: " & Beispielkunde.Ort)
    Debug.Print("Land: " & Beispielkunde.Land)
    Debug.Print("E-Mail: " & Beispielkunde.EMail)
    Debug.Print("Telefon: " & Beispielkunde.Telefon
End Sub
```

Das Ergebnis nach dem ersten Aufruf sieht beispielsweise wie folgt aus:

```
Vorname: Annamae
Nachname: Kling
Strasse: 02613 Schmidt Light
PLZ: 51900
Ort: Rachelleshire
Land: Ecuador
E-Mail: Barton_Casper58@yahoo.com
Telefon: (985) 647-9446
```

Einfachere Schreibweise

Wenn Sie die Schreibweise oben mit den verschachtelten **Function**-Anweisungen zu kompliziert finden, können Sie auch eine einfachere Schreibweise nutzen. Dazu verwenden wir eine einzige **Function**-Anweisung, die **f** und **k** gleichzeitig als Parameter nutzt. Dadurch können Sie innerhalb der **Function** nun einfache Zuweisungen nutzen wie beispielsweise **k.Vorname = f.Name.FirstName**. Wichtig ist, dass Sie die Parameter in der Function in umgekehrter Reihenfolge wie bei den anschließenden Zuweisungen angeben. Die folgende Version zum Anlegen eines Kunden funktioniert genauso wie die vorherige, lässt sich aber viel leichter lesen und auch bearbeiten:

```
Public Sub KundeAnlegenKompakt
    Dim Beispielkunde As New Kunde
    Dim objFaker As New Bogus.Faker(Of Kunde)
    objFaker.Rules(Function(f, k)
        k.Vorname = f.Name.FirstName
        k.Nachname = f.Name.LastName
        k.Strasse = f.Address.StreetAddress
        k.PLZ = f.Address.ZipCode
        k.Ort = f.Address.City
        k.Land = f.Address.Country
        k.EMail = f.Internet.Email
        k.Telefon = f.Phone.PhoneNumber
    End Function)
    Beispielkunde = objFaker.Generate
    ...
End Sub
```

Mehrere Objekte gleichzeitig erstellen

Mit der oben verwendeten **Generate**-Methode haben wir zunächst nur ein einziges **Kunde**-Objekt erstellt. Wenn Sie mehrere **Kunde**-Objekte gleichzeitig erstellen wollen, können Sie dies innerhalb einer Schleife mit der gewünschten Anzahl an Durchläufen erledigen. Es geht jedoch auch noch eleganter. Dazu legen wir in einer weiteren Beispielmethode eine **List**-Variable namens **Kunden** an, die wir für die Aufnahme von **Kunde**-Elemente auslegen. Dann legen wir die üblichen Regeln für das Anlegen der **Kunde**-Elemente fest. Anschließend rufen wir wieder die **Generate**-Methode des **Faker**-Objekts auf, diesmal jedoch übergeben wir als Parameter die Anzahl

Beispieldaten für ein EDM generieren

Im Artikel »Beispieldaten generieren mit Bogus« haben wir gezeigt, wie Sie grundsätzlich Beispieldaten mit der Erweiterung Bogus erzeugen. Dort haben wir allerdings noch offen gelassen, wie Sie solche Daten erzeugen, die in verknüpften Tabellen gespeichert werden sollen – also beispielsweise in zwei Tabellen namens »Kunden« und »Bestellungen«, wobei die Tabelle »Bestellungen« über ein Fremdschlüsselfeld namens »KundeID« mit der Tabelle »Kunden« verknüpft ist. Wie das gelingt, und welche Techniken noch interessant sind für das Schreiben von Beispieldaten über ein Entity Data Model direkt in die zugrunde liegenden Tabellen, beschreiben wir im vorliegenden Artikel.

Beispielprojekt erstellen

Wir erstellen ein einfaches Beispielprojekt des Typs **WPF-App** (für **Visual Basic**) namens **BeispieldatenGenerieren**. Diesem fügen wir ein Entity Data Model hinzu, das wir **BeispieldatenContext** nennen (Menüeintrag **Projekt|Neues Element hinzufügen...**) und das den Typ **Leeres Code First-Modell** erhalten soll. Für dieses Projekt legen wir als Erstes eine Klasse namens **Kunde** an, die wie folgt aussieht:

```
Partial Public Class Kunde
    Public Overridable Property ID As Int32
    Public Overridable Property Vorname As String
    Public Overridable Property Nachname As String
    Public Overridable Property Firma As String
    Public Overridable Property Strasse As String
    Public Overridable Property PLZ As String
    Public Overridable Property Ort As String
    Public Overridable Property Land As String
    Public Overridable Property EMail As String
    Public Overridable Property Bestellungen As ICollection(Of Bestellung)
End Class
```

Außerdem fügen wir eine Klasse namens **Bestellung** hinzu:

```
Partial Public Class Bestellung
    Public Overridable Property ID As Int32
    Public Overridable Property Bestelldatum As DateTime
    Public Overridable Property KundeID As Int32
    Public Overridable Property Kunde As Kunde
    Public Overridable Property Bestellpositionen As ICollection(Of Bestellposition)
End Class
```

Damit wir auch eine m:n-Beziehung abbilden können, legen wir noch eine Klasse **Produkt** und eine Klasse **Bestellposition** an:

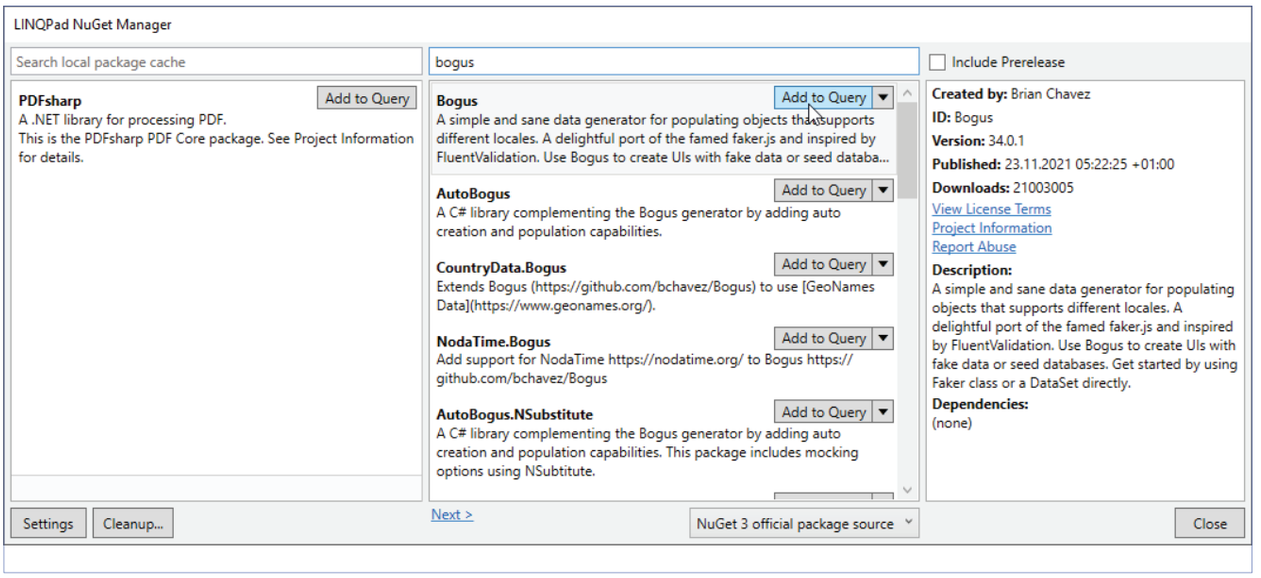


Bild 1: Hinzufügen des Pakets **Bogus**

```
Partial Public Class Bestellposition
    Public Overridable Property ID As Int32
    Public Overridable Property ProduktID As Int32
    Public Overridable Property BestellungID As Int32
    Public Overridable Property Produkt As Produkt
    Public Overridable Property Bestellung As Bestellung
    Public Overridable Property Einzelpreis As Decimal
    Public Overridable Property Menge As Int32
End Class

Partial Public Class Produkt
    Public Overridable Property ID As Int32
    Public Overridable Property Produktname As String
    Public Overridable Property Einzelpreis As Decimal
    Public Overridable Property Bestellpositionen As ICollection(Of Bestellposition)
End Class
```

Der Klasse **BeispieldatenContext.vb** fügen wir eine **DbSet**-Definition für die vorgestellten Klassen hinzu. Außerdem legen wir in der Konstruktor-Methode **New** fest, dass die noch zu definierende Klasse **BeispieldatenInitializer** zum Initialisieren der Datenbank verwendet werden soll:

```
Public Class BeispieldatenContext
    Inherits DbContext
    Public Sub New()
```

```
        MyBase.New("name=BeispieldatenContext")
        Database.SetInitializer(New BeispieldatenInitializer())
    End Sub
    Public Overridable Property Kunden() As DbSet(Of Kunde)
    Public Overridable Property Bestellungen() As DbSet(Of Bestellung)
    Public Overridable Property Produkte() As DbSet(Of Produkt)
    Public Overridable Property Bestellpositionen() As DbSet(Of Bestellposition)
End Class
```

Schließlich legen wir hier noch fest, dass die Elemente der Klassen **Kunde**, **Bestellung**, **Bestellposition** und **Produkt** in den Tabellen **Kunden**, **Bestellungen**, **Bestellpositionen** und **Produkte** landen sollen:

```
Protected Overrides Sub OnModelCreating(modelBuilder As DbModelBuilder)
    MyBase.OnModelCreating(modelBuilder)
    modelBuilder.Entity(Of Kunde)().ToTable("Kunden")
    modelBuilder.Entity(Of Bestellung)().ToTable("Bestellungen")
    modelBuilder.Entity(Of Bestellposition)().ToTable("Bestellpositionen")
    modelBuilder.Entity(Of Produkt)().ToTable("Produkte")
End Sub
```

Bogus zum Projekt hinzufügen

Danach fügen wir das Bogus-Paket zum Projekt hinzu. Dazu wählen Sie den Menüeintrag **Projekt|NuGet-Pakete verwalten** aus. Dies öffnet ein neues Fenster, indem Sie zum Bereich **Durchsuchen** wechseln. Hier geben Sie im Suchfeld den Text **Bogus** ein und finden gleich das passende Paket von **Brian Chavez** vor.

Markieren Sie dieses und starten die Installation mit einem Klick auf die Schaltfläche **Installieren** (siehe Bild 1).

Kundentabelle mit Bogus füllen

Danach starten wir direkt mit der Anwendung. Wir wollen erst einmal nur Daten in der Tabelle **Kunden** anlegen. Dazu fügen wir der Datenbank-Initialisierer-Klasse **Beispieldateninitialisierer** wie folgt eine **Seed**-Methode hinzu:

```
Public Class BeispieldatenInitializer
    Inherits DropCreateDatabaseAlways(Of BeispieldatenContext)

    Protected Overrides Sub Seed(context As BeispieldatenContext)
        Dim Kunden As List(Of Kunde) = New List(Of Kunde)
        Dim objFaker As Bogus.Faker(Of Kunde) = New Bogus.Faker(Of Kunde)("de")
        objFaker.Rules(Function(f, k) As Object
                           k.Vorname = f.Name.FirstName
                           k.Nachname = f.Name.LastName
```

```
                        k.Firma = f.Company.CompanyName
                        k.Strasse = f.Address.StreetAddress
                        k.PLZ = f.Address.ZipCode
                        k.Ort = f.Address.City
                        k.Land = "Deutschland"
                        k.EMail = f.Internet.Email
                    End Function)
        Kunden = objFaker.Generate(1000)
        context.Kunden.AddRange(Kunden)
        MyBase.Seed(context)
    End Sub
End Class
```

Die **Seed**-Methode deklariert und initialisiert ein **List**-Objekt namens **Kunden** für Elemente des Typs **Kunde**. Dann erstellt sie ein sogenanntes **Faker**-Objekt namens **objFaker**, mit dem wir Elemente des Typs **Kunde** füllen wollen. Damit diese später beispielsweise deutsche Adressdaten verwendet und keine amerikanischen, hängen wir noch den Parameter **("de")** an.

Danach legen wir mit der Methode **Rules** die Regeln fest, nach denen neue Elemente mit Daten gefüllt werden sollen. Als Parameter für die **Rules**-Methode geben wir einen **Function**-Lambda-Ausdruck an. Dieser enthält die Zuweisungen der verschiedenen Klassen und Funktionen von Bogus (**f**) zu den Eigenschaften des zu erzeugenden Elements (**k**).

Die Eigenschaft **Vorname** füllen wir beispielsweise mit der Funktion **FirstName** der **Name**-Klasse. Wir möchten nur Kunden mit deutscher Adresse hinzufügen, also stellen wir für das Feld **Land** fix den Wert **Deutschland** ein. Nachdem wir die Zuweisung für alle Eigenschaften erstellt haben, rufen wir die **Generate**-Methode des Objekts aus **objFaker** auf und übergeben dieser als Parameter die Anzahl der zu erstellenden Elemente und weisen diese dem **List**-Objekt **Kunden** zu.

Diese Liste schreiben wir schließlich mit der **AddRange**-Methode in das **DbSet**-Element **Kunden**. Damit die Daten noch in die zugrunde liegende Tabelle der Datenbank geschrieben werden, rufen wir schließlich noch die **Seed**-Methode für den Datenbankkontext aus **context** auf.

Daten beim Start der Anwendung erstellen

Damit diese Daten beim Start der Anwendung geschrieben werden, benötigen wir einen Zugriff auf die Daten der Datenbank. Da wir eine WPF-Anwendung als Beispiel verwenden, die beim Start das Fenster **MainWindow.xaml** anzeigt, bietet es sich an, die notwendigen Anweisungen in die Konstruktor-Methode der zugrunde liegenden Klasse **MainWindow.xaml.vb** zu hinterlegen.

Diese sieht anschließend wie folgt aus und soll einfach die Kunden der Datenbank in die Liste **AlleKunden** einlesen:

```
Class MainWindow
    Public Sub New()
        Dim AlleKunden As List(Of Kunde)
        Dim dbContext As BeispieldatenContext
        dbContext = New BeispieldatenContext
        AlleKunden = New List(Of Kunde) _
            (dbContext.Kunden)
    End Sub
End Class
```

Starten Sie die Anwendung nun, stellt diese beim Zugriff auf den Datenbankkontext fest, dass die Datenbank noch nicht existiert und legt diese nach den Vorgaben in der Klasse **BeispieldatenInitializer** an.

Wenn Sie nun mit dem Menübefehl **Ansicht|SQL Server-Objekt-Explorer** den Bereich **SQL Server-Objekt-Explorer** anzeigen und dort zu der Datenbank navigieren, die in der Datei **App.config** im Bereich **connectionString** definiert ist, finden Sie hier bereits die frisch erstellten Tabellen vor (siehe Bild 2).

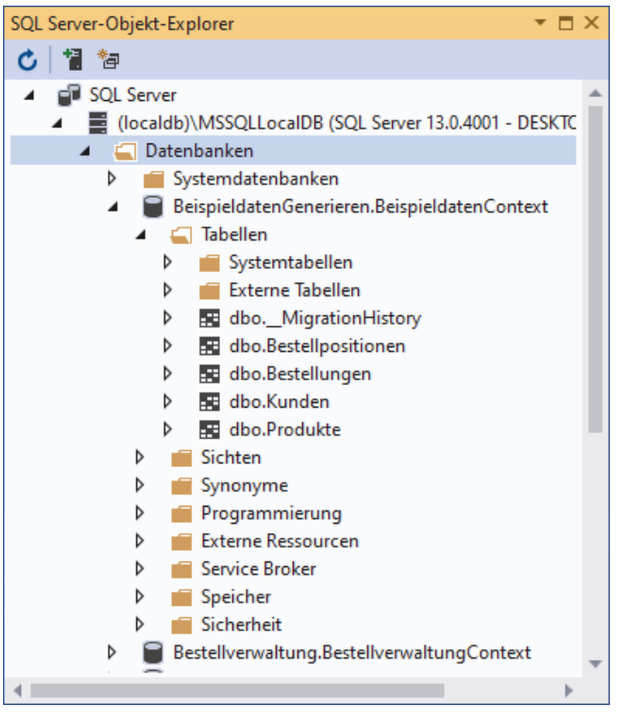
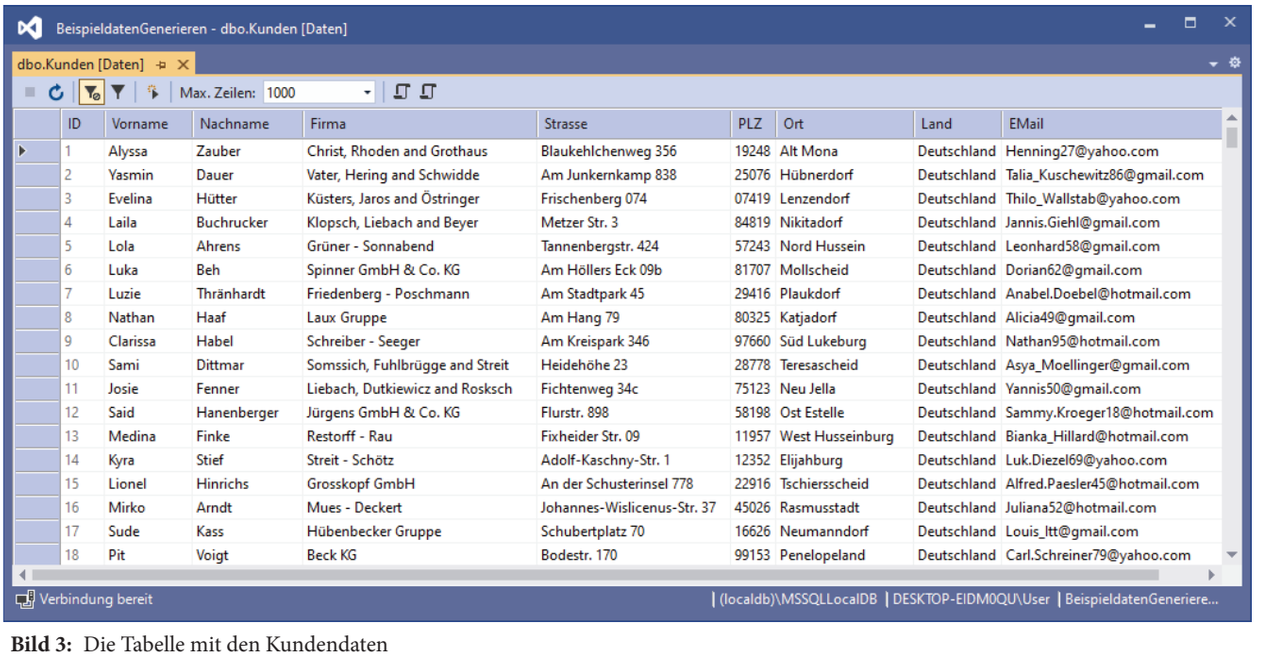


Bild 2: Die neu erstellte Datenbank samt Tabellen im SQL Server-Objekt-Explorer



ID	Vorname	Nachname	Firma	Strasse	PLZ	Ort	Land	EMail
1	Alyssa	Zauber	Christ, Rhoden and Grothaus	Blaukehlchenweg 356	19248	Alt Mona	Deutschland	Henning27@yahoo.com
2	Yasmin	Dauer	Vater, Hering and Schwidde	Am Junkernkamp 838	25076	Hübnerdorf	Deutschland	Talia_Kuschewitz86@gmail.com
3	Evelina	Hütter	Küsters, Jaros and Östringer	Frischenberg 074	07419	Lenzendorf	Deutschland	Thilo_Wallstab@yahoo.com
4	Laila	Buchrucker	Klopsch, Liebach and Beyer	Metzer Str. 3	84819	Nikitadorf	Deutschland	Jannis.Giehl@gmail.com
5	Lola	Ahrens	Grüner - Sonnabend	Tannenbergstr. 424	57243	Nord Hussein	Deutschland	Leonhard58@gmail.com
6	Luka	Beh	Spinner GmbH & Co. KG	Am Höllers Eck 09b	81707	Mollscheid	Deutschland	Dorian62@gmail.com
7	Luzie	Thränhardt	Friedenberg - Poschmann	Am Stadtpark 45	29416	Plaukdorf	Deutschland	Anabel.Doebel@hotmail.com
8	Nathan	Haaf	Laux Gruppe	Am Hang 79	80325	Katjadorf	Deutschland	Alicia49@gmail.com
9	Clarissa	Habel	Schreiber - Seeger	Am Kreispark 346	97660	Süd Lukeburg	Deutschland	Nathan95@hotmail.com
10	Sami	Dittmar	Somssich, Fuhlbrügge and Streit	Heidehöhe 23	28778	Teresascheid	Deutschland	Asya_Moellinger@gmail.com
11	Josie	Fenner	Liebach, Dutkiewicz and Rosksch	Fichtenweg 34c	75123	Neu Jella	Deutschland	Yannis50@gmail.com
12	Said	Hanenberger	Jürgens GmbH & Co. KG	Flurstr. 898	58198	Ost Estelle	Deutschland	Sammy.Kroeger18@hotmail.com
13	Medina	Finke	Restorff - Rau	Fixheider Str. 09	11957	West Husseinburg	Deutschland	Bianka_Hillard@hotmail.com
14	Kyra	Stief	Streit - Schötz	Adolf-Kaschny-Str. 1	12352	Elijahburg	Deutschland	Luk.Diezel69@yahoo.com
15	Lionel	Hinrichs	Grosskopf GmbH	An der Schusterinsel 778	22916	Tschiersscheid	Deutschland	Alfred.Paesler45@hotmail.com
16	Mirko	Arndt	Mues - Deckert	Johannes-Wislicenus-Str. 37	45026	Rasmusstadt	Deutschland	Juliana52@hotmail.com
17	Sude	Kass	Hübenbecker Gruppe	Schubertplatz 70	16626	Neumanndorf	Deutschland	Louis_Itt@gmail.com
18	Pit	Voigt	Beck KG	Bodestr. 170	99153	Penelopeland	Deutschland	Carl.Schreiner79@yahoo.com

Bild 3: Die Tabelle mit den Kundendaten

Seminarverwaltung I: Entity Data Model

Es wird Zeit, die gelernten Techniken mal wieder an einer praktischen Lösung auszuprobieren. In diesem Fall soll es eine Anwendung werden, die ich selbst einsetzen will, da ich seit ein paar Wochen auch Webinare zu verschiedenen Themen anbiete. Eine der Herausforderungen besteht darin, die Kunden, die über einen Onlineshop bestellt haben, in die Anwendung einzulesen und diese den entsprechenden Seminaren beziehungsweise Webinaren zuzuordnen. Zu gegebener Zeit sollen die Teilnehmer eine Mail mit dem Link zur Teams-Sitzung erhalten und anschließend noch einen Link mit dem Download der Aufzeichnung des Seminars. Schließlich sollen auch noch Zertifikate über die Teilnahme erstellt und versendet werden. Wie dies alles gelingt, zeigt der vorliegende Artikel.

Projekt anlegen

Die Gestaltung einer Anwendung zum Verwalten von Daten startet mit dem Anlegen eines Projekts. Wir verwenden ein Projekt des Typs **WPF-App (.NET Framework)** mit der Sprache Visual Basic und auf Basis von XAML für die Gestaltung der Benutzeroberfläche.

Entity Data Model hinzufügen

Wir wollen die Daten in einer SQL Server-Datenbank speichern und über ein Entity Data Model auf diese Daten zugreifen. Dazu fügen wir dem Projekt ein Entity Data Model hinzu. Mit **Strg + Umschalt + A** öffnen wir den Dialog **Neues Element hinzufügen** und wählen dort den Eintrag **ADO.NET Entity Data Model** aus. Nach der Angabe des Namens **SeminarverwaltungContext** klicken wir auf die Schaltfläche **Hinzufügen** (siehe Bild 1).

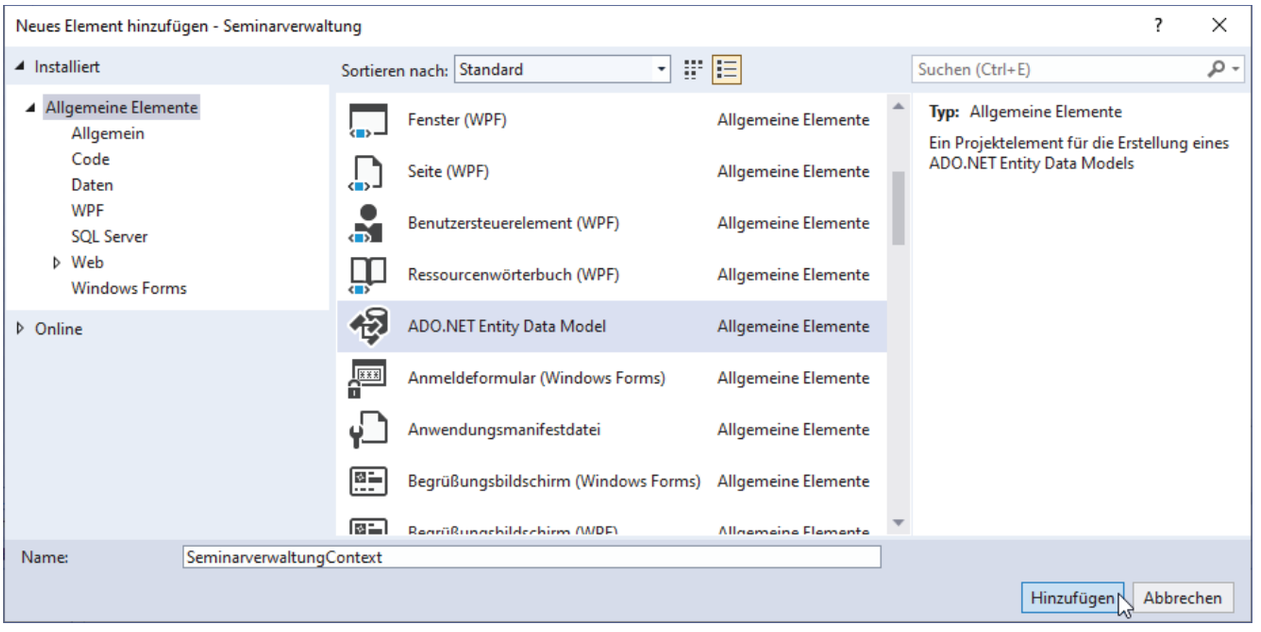


Bild 1: Hinzufügen eines Entity Data Models

Danach wählen Sie im Dialog **Assistent für Entity Data Model** den Eintrag **Leeres Code First-Modell** aus (siehe Bild 2).

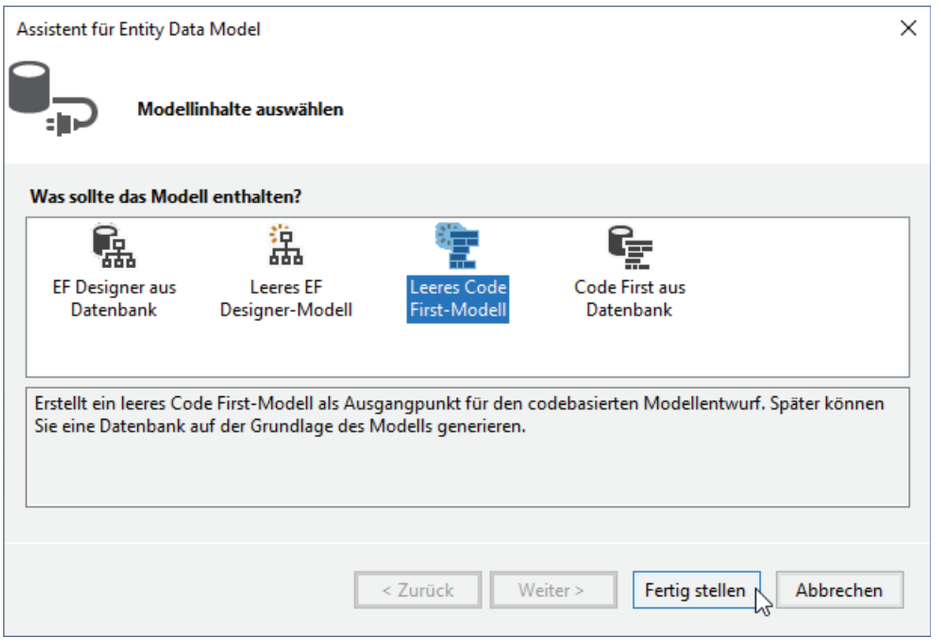


Bild 2: Auswahl des Model-Typs

Nach wenigen Sekunden erscheint die Klasse **Seminarverwaltung-Context**, in der wir für jede zu erstellende Entität beziehungsweise Tabelle einen Eintrag hinzufügen. Hier finden wir auch eine Vorlage für eine Entitätsklasse.

Entitäten hinzufügen

Damit können wir beginnen, die gewünschten Entitäten hinzuzufügen. Dazu überlegen wir uns zunächst, welche Entitäten wir benötigen. Wir benötigen auf jeden Fall eine Entität für die Kunden und eine für die Seminare. Jedem Kunden soll jedes Seminar zugeordnet werden können und umgekehrt. Dafür soll später in der Datenbank eine Verknüpfungstabelle angelegt werden. Damit dies geschieht, benötigen wir keine eigene Entität für die Verknüpfungstabelle – die Beziehungstabelle erstellen wir dann über ein spezielles Mapping.

Die Kunden haben eine Anrede, die wir in einer eigenen Tabelle speichern und dann über ein Fremdschlüsselfeld zuweisen wollen, also legen wir auch eine entsprechende Entität für die Anreden an.

Außerdem benötigen wir zumindest eine Tabelle zur Verwaltung der Seminare. An dieser Stelle kann man sich gleich überlegen, ob es nur eine Tabelle zum Speichern der Seminare gibt. Dann muss man Seminare mit Themen, die nochmal wiederholt werden sollen, immer wieder komplett neu anlegen. Man könnte auch eine Tabelle mit Seminaren erstellen, welche die Basisinformationen eines Seminars enthält wie den Titel, die Inhaltsangabe, den Preis et cetera – beispielsweise namens **Seminarthemen**.

Und dann fügt man eine neue Tabelle namens **Seminare** hinzu, welche die Seminartermine enthält und das Datum mit dem Seminar verknüpft. Oder man macht es gleich wie beim Beispiel der Bestellpositionen in einer Bestellverwaltung, wo man die Details des Artikels wie Einzelpreis oder Steuersatz aus der Artikeltabelle übernimmt, damit diese auch nach Änderung von Preis oder Steuersatz in der Artikeltabelle in der Tabelle für Bestellpositionen enthalten bleiben. In diesem Fall würden wir dann die Felder **Titel**, **Inhalt** und **Preis** von der Tabelle **Seminarthemen** in die Tabelle **Seminare** übernehmen.

Schließlich benötigen wir noch die Verknüpfungstabelle zwischen den Tabellen **Seminare** und **Kunden**, die wir, wie oben beschrieben, allein über Mapping mit der Fluent API definieren. Diese Tabelle soll **SeminareKunden** heißen.

Die Entität Kunde

Für die Entität **Kunde** legen wir die üblichen Eigenschaften wie Vorname, Nachname, Adresse et cetera fest. Wichtig ist hier vor allem die E-Mail-Adresse. Schließlich definieren wir hier eine **ICollection** namens **Seminare** für die Elemente des Typs **Seminar**. Diese wird im Konstruktor der Entitätsklasse erstellt:

```
Public Class Kunde
    Public Overridable Property ID As Int32
    Public Overridable Property Vorname As String
    Public Overridable Property Nachname As String
    Public Overridable Property Firma As String
    Public Overridable Property Strasse As String
    Public Overridable Property PLZ As String
    Public Overridable Property Ort As String
    Public Overridable Property Land As String
    Public Overridable Property EMail As String
    Public Overridable Property UstIDNr As String
    Public Overridable Property Seminare As ICollection(Of Seminar)
End Class
```

Die Entität Seminar

Darauf baut die Klasse für die Entität **Seminar** auf. Sie enthält einen Verweis auf das Seminarthema, auf dessen Basis es erstellt wurde. Außerdem enthält es drei Eigenschaften zum Aufnehmen von **Titel**, **Inhalt** und **Preis** des Seminars, die beim Anlegen eines Objekt auf Basis der Klasse **Seminar** aus dem jeweiligen **Seminarthema**-Objekt gefüllt werden. Dazu gibt es noch das Feld **Seminartermin** und eine Auflistung namens **Kunden**, welche die Kunden enthalten soll, die dieses Seminar gebucht haben. Diese wird in der Konstruktormethode erstellt:

```
Partial Public Class Seminar
    Public Overridable Property ID As Int32
    Public Overridable Property Titel As String
    Public Overridable Property Inhalt As String
    Public Overridable Property Preis As Decimal
    Public Overridable Property Seminartermin As Date
    Public Overridable Property Kunden As ICollection(Of Kunde)
End Class
```

Die dbContext-Klasse SeminarverwaltungContext

Die beim Hinzufügen des Entity Data Models automatisch erstellte Klasse **SeminarverwaltungContext** passen wir wie folgt an.

Seminarverwaltung II: Ribbon und Frame

Um eine Anwendung wie eine Seminarverwaltung ergonomisch steuern zu können, benötigen wir die Möglichkeit, alle wichtigen Elemente der Benutzeroberfläche und Funktionen schnell aufzurufen. Dazu verwenden wir ein Ribbon, dass alle benötigten Steuerelemente oben im Anwendungsfenster anbietet. Dieser Artikel zeigt, wie Sie das Ribbon für diese Anwendung definieren und wie Sie die einzelnen Funktionen vom Ribbon aus aufrufen.

Um ein Ribbon zum Hauptfenster der Anwendung hinzuzufügen, benötigen wir zuerst das **Ribbon**-Element. Dieses steht allerdings nicht standardmäßig zur Verfügung, sodass wir noch einen Verweis zum Projekt hinzufügen. Dazu betätigen wir als Erstes den Menüeintrag **Projekt|Verweis hinzufügen...**, was den Dialog **Verweis-Manager** öffnet. Hier suchen wir mit dem Suchfeld oben rechts nach dem Eintrag **Ribbon**, was direkt den Verweis **System.Windows.Controls.Ribbon** einblendet. Diesen markieren wir und schließen den Dialog mit einem Klick auf die Schaltfläche **OK** (siehe Bild 1).

Ribbon anlegen

Danach finden wir im XAML-Code für das Hauptfenster per IntelliSense schnell das **Ribbon**-Element, mit dem wir die Definition beginnen. Der Code für das Ribbon und das Frame-Element, in dem wir die einzelnen Seiten anzeigen wollen, sieht wie folgt aus. Als Erstes definieren wir ein paar **Grid**-Zeilen, um das Ribbon und das **Frame**-Element in jeweils einer Zeile unterzubringen:

```
<Window x:Class="MainWindow" ...
        Title="MainWindow" Height="450" Width="800" WindowStartupLocation="CenterScreen">
    <Grid>
        <Grid.ColumnDefinitions>
```

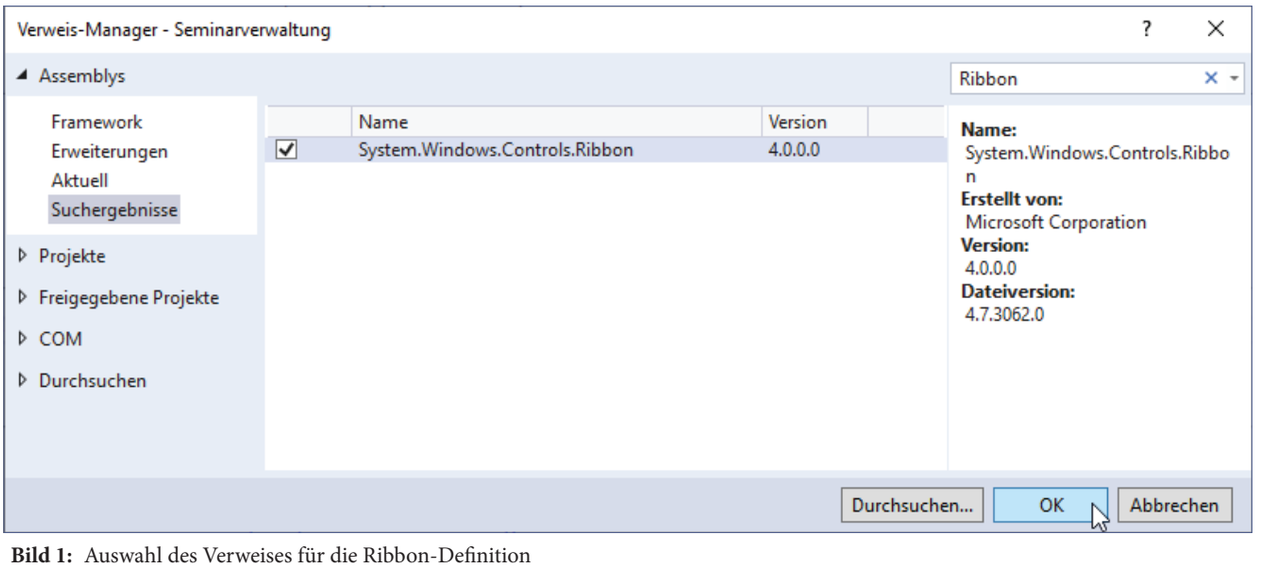


Bild 1: Auswahl des Verweises für die Ribbon-Definition

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

Dann folgt die Definition des **Ribbon**-Elements, dem wir ein **RibbonTab**- und einige **RibbonGroup**-Elemente hinzufügen. Die **RibbonGroup**-Elemente enthalten ein oder mehrere **RibbonButton**-Elemente. Für diese haben wir jeweils die Attribute **x:Name**, **Label**, **LargeImageSource** und **Click** definiert:

```
    <Ribbon Grid.Row="0">
        <RibbonTab Header="Seminarverwaltung">
            <RibbonGroup Header="Kunden">
                <RibbonButton x:Name="btnKundenuebersicht" Label="Übersicht"
                              LargeImageSource="images/users_crowd.png"
                              Click="btnKundenuebersicht_Click"></RibbonButton>
                <RibbonButton x:Name="btnKundeAnlegen" Label="Anlegen"
                              LargeImageSource="images/user_add.png"
                              Click="btnKundeAnlegen_Click"></RibbonButton>
            </RibbonGroup>
            <RibbonGroup Header="Seminare">
                <RibbonButton x:Name="btnSeminaruebersicht" Label="Übersicht"
                              LargeImageSource="images/lecture.png"
                              Click="btnSeminaruebersicht_Click"></RibbonButton>
                <RibbonButton x:Name="btnSeminarAnlegen" Label="Anlegen"
                              LargeImageSource="images/book_plus.png"
                              Click="btnSeminarAnlegen_Click"></RibbonButton>
            </RibbonGroup>
            <RibbonGroup Header="Referenten">
                <RibbonButton x:Name="btnReferentenuebersicht" Label="Übersicht"
                              LargeImageSource="images/teacher_blackboard.png"
                              Click="btnReferentenuebersicht_Click"></RibbonButton>
                <RibbonButton x:Name="btnReferentAnlegen" Label="Anlegen"
                              LargeImageSource="images/teacher_blackboard_plus.png"
                              Click="btnReferentAnlegen_Click"></RibbonButton>
            </RibbonGroup>
            <RibbonGroup Header="Bestellungen">
                <RibbonButton x:Name="btnImportieren" Label="Importieren"
                              LargeImageSource="images/shopping_basket_full.png"
                              Click="btnImportieren_Click"></RibbonButton>
```

Seminarverwaltung III: Daten vom Shopsystem

Unsere Seminarverwaltung lebt unter anderem davon, dass Bestellungen über ein Shopsystem eingehen. Hier nutzen wir elopage, einen Anbieter, mit dem man verschiedene digitale Dienstleistungen online verkaufen kann. elopage übernimmt dabei einige Aufgaben wie etwa die Bereitstellung von Produkt- und Bestellseiten. Nachdem eine Bestellung erfolgt ist, landen die Daten des Kunden inklusive der Daten zum bestellten Artikel in der Datenbank von elopage. Die Bestelldaten können wir per CSV-Datei exportieren und in unsere Seminarverwaltung einlesen. Dazu verwenden wir das NuGet-Paket CsvHelper, das wir in zwei weiteren Artikeln bereits vorgestellt haben. Der vorliegende Artikel zeigt, wie wir die beschriebenen Techniken für den Import in einem konkreten Anwendungsfall nutzen können.

Schritt für Schritt

Wir wollen den Import der CSV-Datei in die einzelnen Tabellen der Anwendung in mehreren Schritten vollziehen:

- Als Erstes wollen wir die Daten aus der CSV-Datei in einer einzigen Klasse verfügbar machen, damit wir komfortabler auf die enthaltenen Daten zugreifen können. Dazu erstellen wir eine Klasse, die für jede Spalte der CSV-Datei ein Feld enthält. Das wird hier und da etwas aufwendiger, wie wir gleich sehen werden.

- Die einzelnen Objekte mit den Daten aus jeweils einer Zeile fügen wir zu einer Auflistung hinzu.

- Danach durchlaufen wir die Auflistung mit allen Datensätzen und lesen daraus die Informationen aus, die wir den einzelnen Entitäten unseres Entity Data Models zuweisen wollen. Diese legen wir dann an und speichern die enthaltenen Daten in der zugrunde liegenden Datenbank.

Herausfordernde CSV-Datei

Die von elopage exportierte Datei liefert uns einige Herausforderungen. Sie sieht wie in Bild 1 aus.

Die Herausforderung ist, dass wir die Felder möglichst so wie in der Kopfzeile dargestellt in Objekten einer neuen Klasse namens **ImportBestellung** abbilden wollen – um möglichst ein 1:1-Mapping zu realisieren:

```
TOKEN;PRODUKT;ERSTELLT;ZAHLUNGSSTATUS;METHODE;PLAN;TESTZEITRAUM;ANZAHL ZAHLUNGEN;BEZAHLT;PRODUCT ID;FUER
PS;FAELLIGER BETRAG;WAEHRUNG;UNTERNEHMEN;PLAN;ZAHLUNGSPLAN;ZAHLUNGSPLAN ID;GUTSCHEINCODE;CAMPAIGN-ID;MIT
TESTZEITRAUM;FIXER FAELLIGKEITSTAG;PRODUKTNAME;EVENT;ORT;EVENT-DATUM;TICKETANZAHL;TICKET CODE;VORNA-
ME;NACHNAME;E-MAIL;TELEFON;LAND;STADT;STRASSE;HAUSNUMMER;PLZ;UNTERNEHMEN;UST-IDNR.;EMPFAENGER NAME;EMP-
FAENGER E-MAIL-ADRESSE;EMPFAENGER TELEFON;EMPFAENGER LAND;EMPFAENGER STADT;EMPFAENGER STRASSE;EMPFAENGER
HAUSNUMMER;EMPFAENGER PLZ;EMPFAENGER FIRMA;NUTZER;PUBLISHER ID;AELTESTE BEZAHLTE RATE;FAELLIGKEITSDA-
TUM;BESTELL-ID;"";"";"";"";"";"";"";"";"";"";"";"";"";"";"";"";""
```

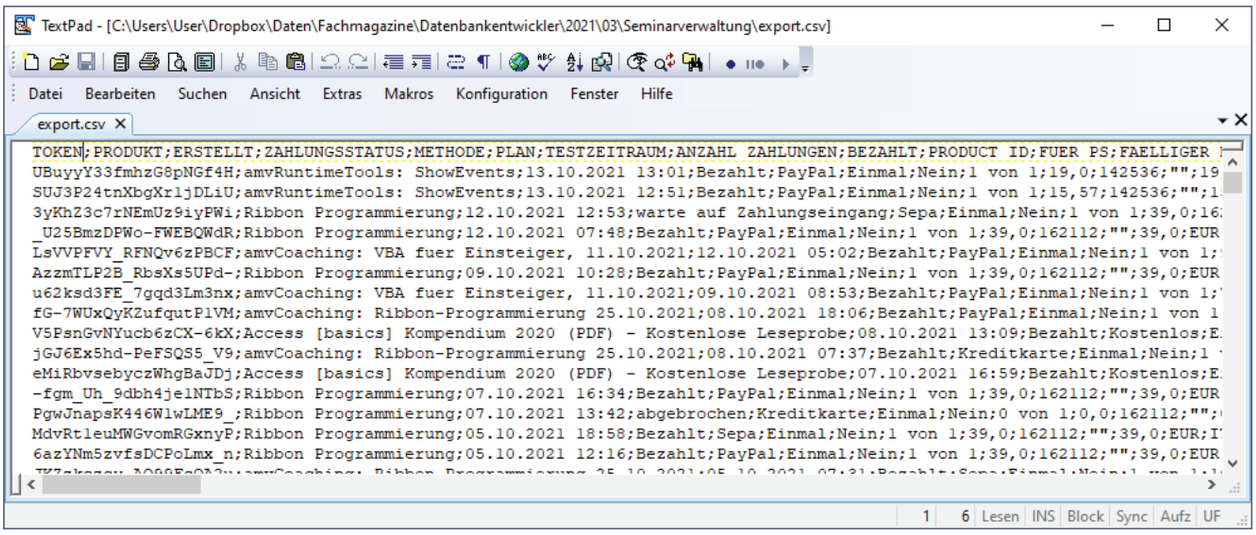


Bild 1: Der CSV-Export des Shopsystems

Sie sehen hier aber direkt einige Elemente mit Namen, die so nicht als Eigenschaften einer .NET-Klasse verwendet werden können. Einige der Spaltenüberschriften enthalten Leerzeichen (zum Beispiel **PRODUCT ID** oder **FAELLIGER BETRAG**), andere entsprechen reservierten Begriffen (**EVENT**), wieder andere kommen doppelt vor (**UNTERNEHMEN**). Zum Glück bietet das **CsvHelper**-Paket Möglichkeiten, solche Felder über entsprechende Data Annotations zuzuordnen.

Wir haben im Folgenden die Klasse **ImportBestellung** ausschnittsweise abgebildet und vor allem die Spezialfälle aufgeführt. Die Klasse verwendet einige zusätzliche Namespaces:

```
Imports CsvHelper
Imports CsvHelper.Configuration
Imports CsvHelper.Configuration.Attributes
```

Die Klasse selbst sieht wie folgt aus:

```
Public Class ImportBestellung
    Public Property TOKEN As String
    ...
    <Name("PLAN")>
    <NameIndex(0)>
    Public Property PLAN1 As String 'Kommt zwei Mal vor
    Public Property TESTZEITRAUM As String
    <Name("ANZAHL ZAHLUNGEN")>
    Public Property ANZAHLZAHLUNGEN As String 'Enthält Leerzeichen
    <Name("PRODUCT ID")>
```

```
    Public Property PRODUCT_ID As String 'Enthält Leerzeichen
    <Name("FUER PS")>
    ...
    <Name("UNTERNEHMEN")>
    <NameIndex(0)>
    Public Property UNTERNEHMEN As String 'Kommt zwei Mal vor.
    <Name("PLAN")>
    <NameIndex(1)>
    Public Property PLAN2 As String
    Public Property ZAHLUNGSPLAN As String
    <Name("EVENT")>
    Public Property Event1 As String 'Event ist ein reserviertes Schlüsselwort
    <Name("EVENT-DATUM")>
    Public Property Event_DATUM As String 'Enthält Minuszeichen
    <Name("UNTERNEHMEN")>
    <NameIndex(1)>
    Public Property UNTERNEHMEN_1 As String 'Zweites Vorkommen
    <Name("UST-IDNR.")>
    ...
End Class
```

Die vollständige Klasse finden Sie im Beispielprojekt zu diesem Artikel.

Importieren der CSV-Datei

Das Importieren der CSV-Datei wollen wir möglichst komfortabel gestalten, deshalb speichern wir das zuletzt verwendete Verzeichnis in einer Option der Anwendung und rufen dieses beim nächsten Anzeigen des Dateiauswahl-Dialogs wieder ab.

Für das Importieren haben wir dem Ribbon eine Schaltfläche namens **btnImportieren** hinzugefügt. Diese löst die folgende Ereignismethode aus:

```
Private Sub btnImportieren_Click(sender As Object, e As RoutedEventArgs)
    Dim strDateiname As String
    Dim objImportBestellungen As List(Of ImportBestellung)
    strDateiname = DateinameErmitteln()
    objImportBestellungen = CSVImport(strDateiname)
    CSVImportVerarbeiten(objImportBestellungen)
End Sub
```

Hier rufen wir zuerst die Funktion **DateinameErmitteln** auf. Diese liest mit der Funktion **VerzeichnisLesen** das zuletzt verwendete Verzeichnis in die Variable **strImportverzeichnis** ein. Ist diese leer, verwendet die Methode

das Verzeichnis der **.exe**-Datei der Anwendung als Startverzeichnis für die Eigenschaft **InitialDirectory**. Anderenfalls kommt das Verzeichnis aus **strImportverzeichnis** zum Einsatz:

```
Private Function DateinameErmitteln() As String
    Dim strImportpfad As String
    Dim strImportverzeichnis As String
    Dim objOpenFileDialog As New OpenFileDialog
    strImportverzeichnis = VerzeichnisLesen()
    If strImportverzeichnis.Length = 0 Then
        objOpenFileDialog.InitialDirectory = AppDomain.CurrentDomain.BaseDirectory
    Else
        objOpenFileDialog.InitialDirectory = strImportverzeichnis
    End If
```

Dann stellt die Methode **.csv** als Dateiendung für den **Datei öffnen**-Dialog ein. Die Methode **ShowDialog** zeigt den Dialog an. Wird dieser unter Angabe eines Dateinamens geschlossen, weist die Methode diesen der Variablen **strImportpfad** zu. Schließlich sorgt die Methode **VerzeichnisAktualisieren** noch dafür, dass das gewählte Verzeichnis in die Optionen der Anwendung geschrieben wird:

```
    objOpenFileDialog.DefaultExt = ".csv"
    If (objOpenFileDialog.ShowDialog = True) Then
        strImportpfad = objOpenFileDialog.FileName
        VerzeichnisAktualisieren(strImportpfad)
    End If
    Return strImportpfad
End Function
```

Speichern des Importverzeichnisses

Damit das Importverzeichnis gespeichert wird und nicht immer wieder neu ausgewählt werden muss, legen wir in den Einstellungen der Anwendung einen neuen Eintrag namens **Importverzeichnis** an. Dieses erhält den Typ **String** und den Bereich **Benutzer** (siehe Bild 2).

Die Methode **VerzeichnisAktualisieren** erwartet den Pfad mit dem zu speichernden Verzeichnis als Parameter. Sie ermittelt zunächst mit der Methode **GetDirectoryName** der Klasse **Path** das Verzeichnis aus der Pfadangabe. Dann stellt sie die Eigenschaft **Importverzeichnis** der Anwendungseinstellungen auf den Wert aus **strImportverzeichnis** ein:

```
Private Sub VerzeichnisAktualisieren(strImportpfad As String)
    Dim objSettings As New MySettings
    Dim strImportverzeichnis As String
    strImportverzeichnis = Path.GetDirectoryName(strImportpfad)
```