TUTORIALREIHE WPF LERNEN

WPF verständlich und anhand von Praxisbeispielen erklärt

Sascha-Heinz Patschka

Inhaltsverzeichnis (inkrementell)

4.0

Eine WPF Telefonbuch App

Jetzt kommen wir zum praktischen Teil. Wir erstellen nun eine kleine App unter Verwendung des bisher gelerntem. Wir werden noch nicht mit MVVM oder ähnlichem Pattern in Berührung kommen, das ist aber gewollt so. Es ist besser wenn wir uns noch nicht mit komplizierten Pattern Beschäftigen und erstmal lernen wie wir nun z.b. von Binding profitieren können.

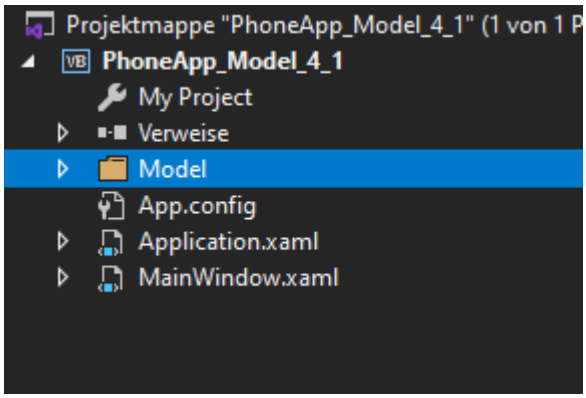
Wir haben ja gelernt dass wir wunderbar auf Klassen Binden können und hier ein wirklich Leistungsstarkes System an die Hand bekommen um schnell und unkompliziert Daten mit Controls zu verbinden. Und genau das versuchen wir nun in der Praxis.

Es wird eine Telefonbuch App.
Da ich ungerne Beispiele mache welche nicht wirklich funktionsfähig sind soll die App - auch wenn sie nur diesem Tutorial dienen soll - Daten speichern können. Also soll auch die persistierung funktionieren, damit wir hier flexibel sind werde ich Klassen haben welche schlicht Serialisiert werden. Wir könnten in diese „Modelklassen“ natürlich theoretisch INotifyPropertyChanged implementieren um diese direkt als eine Art „ViewModel“ zu missbrauchen. Ich bin aber kein Freund davon. Warum werden wir im Laufe dieses Kapitel noch näher erfahren. Nun gut, fangen wir mit dem Model eines Telefonbuchs an.

4.1

Das Model erstellen / was soll die App können

Wir erstellen eine neue WPF Solution in welcher wir erstmal einen Order „Model“ anlegen. In diesem landen unsere Modelklassen.



Wir legen eine neue Klasse „Contact“ an und fügen folgende typische Properties ein.

1

```
Namespace Model

    Public Class Contact

        Public Property FirstName As String
        Public Property LastName As String
        Public Property ImagePath As String
        Public Property Birthday As Date?
        Public Property Note As String

    End Class
End Namespace
```

Ich denke diese erklären sich von selbst und sind typisch für einen Kontakt eines Telefonbuchs.

Nun hätten wir gerne noch eine Eigenschaft um die Telefonnummern oder Mailadressen zu hinterlegen. Das können ja mehrere je Kontakt sein. Also ein Kontakt kann Beispielsweise zwei Mailadressen und eine Telefonnummer haben. Aber vielleicht auch eine Website und einen SocialMedia Account wie Facebook oder Instagram.
Wir benötigen hierfür eine Auflistung. Also legen wir uns eine Eigenschaft vom Typ List(Of T) an. Dafür benötigen wir aber erstmal eine neue Klasse. Nennen wir diese ContactData.

```
Namespace Model
    Public Class ContactData

        Public Property Data As String
        Public Property DataType As EnumContactDataType

    End Class

    Public Enum EnumContactDataType
        Phonenumber = 0
        Mail = 1
        SocialMedia = 2
        Fax = 3
        Website = 4
    End Enum

End Namespace
```

Diese Klasse hat zwei Eigenschaften. Zum einen die Daten vom Typ String. Denn es sollen sowohl Telefonnummern aber auch z.b. Mailadressen gespeichert werden können. Da ist String der einzige Datentyp der hierfür geeignet ist. Um aber später unterschieden zu können um welche Art von Daten es sich hierbei handelt habe ich einen Enumerator erstellt und eine Eigenschaft welche den Wert dieses Enumerators hält. So können wir später die Daten richtig zuordnen.

Nun können wir diese als Auflistung in unsere Contact Klasse einbinden.

2

```
Namespace Model

    Public Class Contact

        Public Property FirstName As String
        Public Property LastName As String
        …
        Public Property ContactData As List(Of ContactData)

    End Class
End Namespace
```

Tja, ich würde sagen das reicht uns fürs erste erstmal als Model für unser Telefonbuch. Ich weiß das Kapitel war etwas kurz, aber keine Sorge, beim nächsten geht's ran an den Speck.

Hier geht's zum Video: https://youtu.be/2eWaAs1w9ok

Hier zum Thread im VB-Paradise-Forum: https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

4.2

Die erste View erstellen

In diesem Kaptitel werden wir unsere erste View mit einer Klasse verbinden. Wie im letzten Kaptiel bereits erwähnt werde ich hier nicht direkt mit den Modelklassen arbeiten, ihr werdet im Verlauf der nächsten Kapitel – aber auch in diesem – sehen warum dies von Vorteil sein kann.

Als legen wir uns einen Order für unsere „ViewModel"-Klassen an. Ich nenne diese ViewModel-Klassen auch wenn es hier nicht um MVVM geht. Das eine hat mit dem anderen ja nicht viel zu tun. Jede Klasse welche mit einem View verbunden wird kann als ViewModel-Klasse bezeichnet werden.

Als erste Klasse lege ich eine Basisklasse an damit wir nicht für jede einzelne Klasse INotifyPropertyChanged implementieren müssen sondern eine Basisklasse haben von welcher wir immer wieder erben können. Das spart uns viel Tipparbeit. Die ViewModel-Klasse packe ich alle in einen seperaten Namespace „ViewModel".

3

```
Namespace ViewModel
    Public MustInherit Class ViewModelBase
        Implements INotifyPropertyChanged

        Protected Overridable Sub RaisePropertyChanged(<CallerMemberName> Optional
prop As String = "")
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(prop))
        End Sub

        Public Event PropertyChanged As PropertyChangedEventHandler Implements
INotifyPropertyChanged.PropertyChanged
    End Class
End Namespace
```

Nun können wir bereits beginnen die erste ViewModel-Klasse anzulegen. Die ContactViewModel-Klasse. Diese Klasse hat die Aufgabe die Eigenschaften der Model-Klasse „Contact“ für die View aufzubereiten. Wie schon erwähnt machen es viele so das sie INotifyPropertyChanged in der Modelklasse implementieren um sich so diese Zwischenklasse zu „sparen“. Das ist natürlich ein Weg den man gehen kann um dem *dont repeat yourself* Grundsatz zu folgen, ich habe aber eben die Erfahrung gemacht das sich diese zusätzlicher Arbeit durchaus auszahlt und sogar eher komplexität herausnimmt – auch wenn dies auf den ersten Blick nicht so aussieht - speziell auf Hinblick auf späteres MVVM, aber das ist ein späteres Thema, ihr werdet auch in diesem Kapitel bereits sehen das dies von Vorteil sein kann.

Erstellen wir erstmal eine Klasse, ich erkläre hierzu gleich ein paar Dinge:

```
Namespace ViewModel
    Public Class ContactViewModel
        Inherits ViewModelBase

        Private ReadOnly _modelContact As Model.Contact

        Public Sub New()
            _modelContact = New Model.Contact
            ContactData = New ObservableCollection(Of ContactDataViewModel)
        End Sub

        Friend Sub New(model As Model.Contact)
            _modelContact = model
            ContactData = New ObservableCollection(Of ContactDataViewModel)
            model.ContactData.ForEach(Sub(x) ContactData.Add(New ContactDataViewModel(x)))
        End Sub

        Public Property FullName() As String
            Get
                Return $"{_modelContact.FirstName} {_modelContact.LastName}"
            End Get
            Set(ByVal value As String)
                Dim fullname = value.Split(" ")
                _modelContact.FirstName = fullname.First
                If fullname.Length > 1 Then _modelContact.LastName = fullname.Last
                RaisePropertyChanged()
            End Set
        End Property

        Public Property ImagePath As String
            Get
                Return _modelContact.ImagePath
            End Get
            Set(value As String)
                _modelContact.ImagePath = value
                RaisePropertyChanged()
            End Set
        End Property
        Public Property Birthday As Date?
            Get
                Return _modelContact.Birthday
            End Get
            Set(value As Date?)
                _modelContact.Birthday = value
                RaisePropertyChanged()
            End Set
        End Property
        Public Property Note As String
            Get
                Return _modelContact.Note
            End Get
            Set(value As String)
                _modelContact.Note = value
                RaisePropertyChanged()
            End Set
        End Property

        Private _contactData As ObservableCollection(Of ContactDataViewModel)
        Public Property ContactData() As ObservableCollection(Of ContactDataViewModel)
            Get
                Return _contactData
            End Get
            Set(ByVal value As ObservableCollection(Of ContactDataViewModel))
                _contactData = value
                RaisePropertyChanged()
            End Set
        End Property

    End Class
End Namespace
```

OK, das sieht erstmal verwirrend aus. Gebe ich zu.
Fangen wir oben an. Wir haben eine Private Schreibgeschützte Variable vom Typ Model.Contact.
Diese Variable dient dazu unser „Modelobjekt“ zu halten. Wie an den beiden Konstuktoren zu sehen, wird entweder eine neue Instanz eines Contact erstellt falls beim Instaziieren unseres ViewModels keine Instanz hereingereicht wird, oder es wird die Instanz an die Variable übergeben. Wir haben somit immer ein Model-Object in dieser Variable.

In späterer folge haben wir dann eine Eigenschaft FullName. Hier ist schon mal ein Fall für den eine ViewModel Klasse eben praktisch ist. In unserem Model ist definiert das wir FirstName und LastName getrennt speichern werden. Wie Ihr es vom Handy evtl. kennt kann man einen neuen Kontakt aber meißt vereinfacht eingeben indem man einfach den Vor und den Zunamen in einer Zeile eingibt. Tja, unser Telefonbuch soll dies auch können, gespeichert soll aber trotzdem getrennt werden. Also wenn der User „Sascha Patschka“ in das Feld eingibt soll unsere App hier „Sascha“ als Vornamen und „Patschka“ als Nachnamen speichern. Das ist bereits der Vorteil der Trennung zwischen einem *ViewModel* und einem *Model*. Wir können frei bestimmen und sind flexibel wie die View aussehen soll bzw. wie Daten sowohl angezeigt als auch eingegeben werden können.

Im Getter der Eigenschaft setzen wir die Model-Eigenschaften FirstName und LastName zusammen und geben diese mit einem Leehrzeichen getrennt aus.
Im Setter lesen wir den String aus und versuchen diesen zu Splitten und anschliessend getrennt an das Modelobjekt zu übergeben . Die Art wie dies geschied ist im Moment nur sehr rudimentär implementiert und hat noch so einige Fehler, reicht uns aber fürs erste um einfach hier nur eine Idee zu bekommen was wir hier vorhaben.

Nun sehen wir uns die nächsten Eigenschaften an. Diese reichen im Grunde einfach nur durch.

```
Public Property ImagePath As String
     Get
         Return _modelContact.ImagePath
     End Get
     Set(value As String)
        _modelContact.ImagePath = value
        RaisePropertyChanged()
    End Set
End Property
```

Im Getter wir der ImagePath des ModelObjekts einfach durchgereicht und im Setter wird ImagePath des Modelobjekts gesetzt. Man kann dies als Wrapper bezeichnen.

Im Grunde geht es einfach darum eine Eigenschaft zu haben welche nach dem setzen des Wertes das Event PropertyChanged wirft und das macht unsere Eigenschaft im Setter mit der Methode RaisePropertyChanged().
Kommen wir zur letzten Eigenschaft „ContactData“ vom Typ ObservableCollection(Of ContactDataViewModel).
Diese in Verbindung mit der dritten Zeile des Konstruktors ist etwas speziell und kann vielleicht verwirrend sein. Aber keine Sorge, wir gehen das langsam durch.

Erstmal benötigen wir den Typ ContactDataViewModel.

```
Namespace ViewModel
    Public Class ContactDataViewModel
        Inherits ViewModelBase

        Private ReadOnly _modelContactData As ContactData

        Public Sub New()
            _modelContactData = New ContactData()
        End Sub

        Friend Sub New(model As Model.ContactData)
            _modelContactData = model
        End Sub

        Public Property Data As String
            Get
                Return _modelContactData.Data
            End Get
            Set(value As String)
                _modelContactData.Data = value
                RaisePropertyChanged()
            End Set
        End Property
        Public Property DataType As EnumContactDataType
            Get
                Return _modelContactData.DataType
            End Get
            Set(value As EnumContactDataType)
                _modelContactData.DataType = value
                RaisePropertyChanged()
            End Set
        End Property

    End Class
End Namespace
```

Diese Klasse sollte soweit verständlich sein, auch diese dient als „Wrapper“ für ein Model-Objekt.

Zurück zur ContactViewModel-Klasse…
Wir bekommen ja unser Model-Objekt in die private Variable. Um nun die Auflistung an ContactData des Model-Objekts in die Eigenschaft ContactData des ViewModels zu bekommen können wir diesmal nicht einfach „durchreichen“ da die Typen sich unterschieden. Wir müssen

die Typen also konvertieren. Das machen wir am einfachsten indem wir in einer Schleife die Auflistung durchgehen.

Da wir in der Klasse ContactDataViewModel einen Konstruktor haben welchem wir ein Model-Objekt übergeben können ist dies auch einfach. Wir erstellen schlicht für jedes Objekt in der Auflistung des Model-Objekts eine neue Instanz von ContactDataViewModel und verwenden den Konstruktor mit der Überladung welcher wir ein Modelobjekt mitgeben können was in der dritten Zeile des Konstruktors von ContactViewModel passiert.

Schon haben wir dafür gesorgt dass die Auflistung mit Daten gefüllt ist.

Nun muss nur noch schnell ein View her um das mal zu testen.
Wir öffnen die CodeBehind des MainWindow und erstellen erstmal eine Eigenschaft welche eine Hand voll Testdaten halten soll. Eine ObservableCollection(Of ContactViewModel)

```
Class MainWindow

    Private _allContacts As ObservableCollection(Of ContactViewModel)
    Public Property AllContacts() As ObservableCollection(Of ContactViewModel)
        Get
            Return _allContacts
        End Get
        Set(ByVal value As ObservableCollection(Of ContactViewModel))
            _allContacts = value
        End Set
    End Property

End Class
```

OK, aber wir müssen die WPF für Binding wieder Benachrichtigen. Die Basisklasse für unsere ViewModels können wir bei einem Window aber nicht erben, wir müssen hier also INotifyPropertyChanged implementieren.

```
Class MainWindow
    Implements INotifyPropertyChanged

    Private _allContacts As ObservableCollection(Of ContactViewModel)
    Public Property AllContacts() As ObservableCollection(Of ContactViewModel)
        Get
            Return _allContacts
        End Get
        Set(ByVal value As ObservableCollection(Of ContactViewModel))
            _allContacts = value
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(NameOf(AllContacts)))
        End Set
    End Property


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

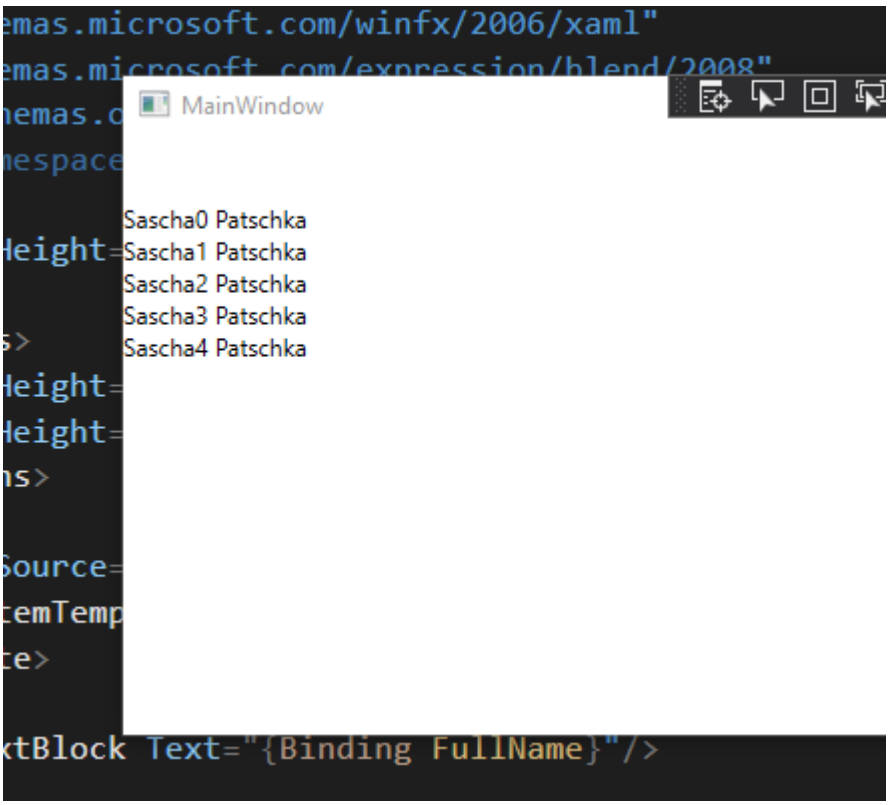
Nun Abonnieren wir das Loaded Event des Window um zum einen ein paar Testdaten zu erstellen und zum anderen die View an die CodeBehind zu Binden.

```
Class MainWindow
    Implements INotifyPropertyChanged

    Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles Me.Loaded
        AllContacts = New ObservableCollection(Of ContactViewModel)

        For i As Integer = 0 To 4
            AllContacts.Add(New ContactViewModel(New Model.Contact() With {.FirstName = "Sascha" & i,
.LastName = "Patschka",
                                                 .Birthday = New Date(1983, 9, 12), .Note = "Eine
Notiz", .ContactData = New List(Of Model.ContactData) From {
                                                   New Model.ContactData() With {.Data = "+43 664
1234567", .DataType = Model.EnumContactDataType.Phonenumber}})))
        Next

        Me.DataContext = Me
    End Sub


    Private _allContacts As ObservableCollection(Of ContactViewModel)
    Public Property AllContacts() As ObservableCollection(Of ContactViewModel)
        Get
            Return _allContacts
        End Get
        Set(ByVal value As ObservableCollection(Of ContactViewModel))
            _allContacts = value
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(NameOf(AllContacts)))
        End Set
    End Property


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

Wir füllen also AllContacts mit Testdaten in einer Schleife. Nun können wir versuchen diese Testdaten mal in einem View anzuzeigen.

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:_4_2_PhoneBook"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="47*"/>
            <RowDefinition Height="372*"/>
        </Grid.RowDefinitions>

        <ItemsControl ItemsSource="{Binding AllContacts}" Grid.Row="1">
            <ItemsControl.ItemTemplate>
                <DataTemplate>

                        <TextBlock Text="{Binding FullName}"/>

                </DataTemplate>
            </ItemsControl.ItemTemplate>
        </ItemsControl>


    </Grid>
</Window>
```

Na das sieht doch gut aus. Nicht von der Optik her, sondern das wir korrekt gebunden haben und unser Code bis hierher erstmal funktioniert.



Hier geht's zum Video: https://youtu.be/t-0bwri7wVk

Hier zum Thread im VB-Paradise-Forum: https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/