WPF-Elemente

Inhalt

[Progressbar 2](#_Toc535851644)

[XAML-Code 3](#_Toc535851645)

[ProgressBarVM 3](#_Toc535851646)

[Radiobuttons & Checkbox 5](#_Toc535851647)

[XAML-Code 5](#_Toc535851648)

[RadioButtonCheckBoxVM 7](#_Toc535851649)

[Converter 7](#_Toc535851650)

[TextBox, TextBlock, Label 8](#_Toc535851651)

[XAML-Code 9](#_Toc535851652)

[TextBoxTextBlockLabelVM 10](#_Toc535851653)

[Simple Slider Binding 12](#_Toc535851654)

[Slider Binding mit ViewModel 12](#_Toc535851655)

[SliderVM.cs 12](#_Toc535851656)

[RelayCommand.cs 13](#_Toc535851657)

[ColourSlider.xaml 13](#_Toc535851658)

[ColourSlider.xaml.cs 14](#_Toc535851659)

[ListBox 15](#_Toc535851660)

[Skill.cs 15](#_Toc535851661)

[MainWindow.xaml.cs 15](#_Toc535851662)

[MainWindow.xaml 16](#_Toc535851663)

[RelayCommand.cs 17](#_Toc535851664)

[SkillVM.cs 17](#_Toc535851665)

[DataGrid 20](#_Toc535851666)

[User.cs 20](#_Toc535851667)

[BaseRepo.cs 20](#_Toc535851668)

[UserRepo.cs 21](#_Toc535851669)

[ContextClass.cs 22](#_Toc535851670)

[CustomInitializer.cs 22](#_Toc535851671)

[AViewModel.cs 23](#_Toc535851672)

[RelayCommand.cs 23](#_Toc535851673)

[UserVM.cs 24](#_Toc535851674)

[UsersVM.cs 25](#_Toc535851675)

[MainWindow.xaml 27](#_Toc535851676)

[UsersView.xaml 27](#_Toc535851677)

[UserView.xaml 28](#_Toc535851678)

[CreateUserView.xaml 28](#_Toc535851679)

[CreateUserView.xaml.cs 29](#_Toc535851680)

[UpdateUserView.xaml 29](#_Toc535851681)

[UpdateUserView.xaml.cs 30](#_Toc535851682)

# Progressbar

Das Progressbar-Element hat grundsätzlich drei Properties, welche für die Darstellung der Werte besonders wichtig sind.

* Minimum: Der kleinstmögliche Wert im Progressbar-Element.
* Maximum: Der größtmögliche Wert im Progressbar-Element.
* Value: Der aktuelle Wert des Progressbar-Elements.

## XAML-Code

<Window x:Class="ProgressBarExample.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:ProgressBarVMSpace"

mc:Ignorable="d"

Title="MainWindow" Height="132.036" Width="532.335">

<Window.DataContext>

<local:ProgressBarVM></local:ProgressBarVM>

</Window.DataContext>

<Grid>

<!-- Displays CurrentValue from ProgressBarVM. The maximum and minimum are also defined in ProgressBarVM-->

<ProgressBar HorizontalAlignment="Left" Height="32" Margin="10,10,0,0" VerticalAlignment="Top" Width="497" Minimum="{Binding MinValue}" Maximum="{Binding MaxValue}" Value="{Binding CurrentValue}"/>

<!-- Changes current value and text changes if the CurrentValue changes-->

<TextBox HorizontalAlignment="Left" Height="23" Margin="387,68,0,0" TextWrapping="Wrap" Text="{Binding CurrentValue, Mode=TwoWay}" VerticalAlignment="Top" Width="120" x:Name="txt\_value"/>

<!-- Automatically increase CurrentValue every second -->

<Button Content="Start" HorizontalAlignment="Left" Margin="10,68,0,0" VerticalAlignment="Top" Width="75" Height="23" Command="{Binding IncreaseToMax}"/>

<!-- Stops the timer that increases CurrentValue-->

<Button Content="Stop" HorizontalAlignment="Left" Margin="90,68,0,0" VerticalAlignment="Top" Width="75" Height="23" Command="{Binding StopIncreaseToMax}"/>

</Grid>

</Window>

## ProgressBarVM

public class ProgressBarVM:AViewModel

{

private System.Timers.Timer aTimer;

private int currentValue;

private int minValue;

private int maxValue;

public ProgressBarVM()

{

minValue = 0;

maxValue = 100;

currentValue = 99;

}

//Current value of the progressbar

public int CurrentValue

{

get

{

return currentValue;

}

set

{

currentValue = value;

CallPropertyChanged("CurrentValue");

}

}

//Minimum of the progressbar

public int MinValue

{

get

{

return minValue;

}

}

//Maximumn of the progressbar

public int MaxValue

{

get

{

return maxValue;

}

}

//Command that sets the timer

public RelayCommand IncreaseToMax

{

get

{

return new RelayCommand(

o =>

{

aTimer = new System.Timers.Timer(1000);

aTimer.Elapsed += Increase;

aTimer.AutoReset = true;

aTimer.Enabled = true;

},

o => true);

}

}

//Command that stops and disposes the timer

public RelayCommand StopIncreaseToMax

{

get

{

return new RelayCommand(

o =>

{

aTimer.Stop();

aTimer.Dispose();

},

o => true);

}

}

//Method that is called every second by the timer

private void Increase(Object source, ElapsedEventArgs e)

{

if (CurrentValue < MaxValue)

{

CurrentValue=CurrentValue+1;

}

else

{

CurrentValue = MinValue;

}

}

}

Sobald der Command IncreaseToMax aufgerufen wird, wird CurrentValue so lange inkrementiert, bis das Maximum erreicht wird. Sobald CurrentValue das Maximum erreicht, wird wieder vom Minimum losgezählt.

# Radiobuttons & Checkbox

Normalerweise werden Radiobuttons und Ceckboxes für die Darstellung von bools verwendet. Sobald diese Elemente andere Werte abdecken sollen, werden Converter benötigt.

## XAML-Code

<Window x:Class="RadioButtonCheckBox\_Example.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:RadioButtonCheckBox\_Example"

mc:Ignorable="d"

Title="MainWindow" Height="277.695" Width="220.06">

<Window.DataContext>

<local:RadioButtonCheckBoxVM></local:RadioButtonCheckBoxVM>

</Window.DataContext>

<Grid>

<!-- Includes coverters that are being used-->

<Grid.Resources>

<!--This is a custom converter which converts enums to bools and bools to enums-->

<local:EnumBooleanConverter x:Key="EnumBooleanConverter" />

<!--This converter is already implemented and provided-->

<BooleanToVisibilityConverter x:Key="b2v"/>

</Grid.Resources>

<!--The header of the groupbox is the chosen enum value. The property Visibility uses a bool in the RadioButtonCheckBoxVM and the provided BooleanToVisibilityConverter to determine if the groubox is visible or not-->

<GroupBox Header="{Binding Path=Selection}" HorizontalAlignment="Left" Height="153" Margin="10,10,0,0" VerticalAlignment="Top" Width="175" Visibility="{Binding Path=Visible, Converter={StaticResource b2v}}">

<StackPanel>

<!--The IsChecked-Properties of the radiobuttons are bound to Selection from RadioButtonCheckBoxVM. They use the custom EnumBooleanConverter and ConverterParameter (in this example the enum values are being used; can also be string or something else) to determine if the rradiobuttons needs to be checked-->

<RadioButton IsChecked="{Binding Path=Selection, Converter={StaticResource EnumBooleanConverter}, ConverterParameter={x:Static local:ESelection.FIRST\_SELECTION}}" x:Name="rdb\_FirstSelection" Content="FirstSelection" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top"/>

<RadioButton IsChecked="{Binding Path=Selection, Converter={StaticResource EnumBooleanConverter}, ConverterParameter={x:Static local:ESelection.SECOND\_SELECTION}}" x:Name="rdb\_SecondSelection" Content="SecondSelection" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top"/>

<RadioButton IsChecked="{Binding Path=Selection, Converter={StaticResource EnumBooleanConverter}, ConverterParameter={x:Static local:ESelection.THIRD\_SELECTION}}" x:Name="rdb\_ThirdSelection" Content="ThirdSelection" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top"/>

<RadioButton IsChecked="{Binding Path=Selection, Converter={StaticResource EnumBooleanConverter}, ConverterParameter={x:Static local:ESelection.FOURTH\_SELECTION}}" x:Name="rdb\_FourthSelection" Content="FourthSelection" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top"/>

<RadioButton IsChecked="{Binding Path=Selection, Converter={StaticResource EnumBooleanConverter}, ConverterParameter={x:Static local:ESelection.FIFTH\_SELECTION}}" x:Name="rdb\_FifthSelection" Content="FifthSelection" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top"/>

</StackPanel>

</GroupBox>

<!--Command that opens a messagebox with the selected enum value-->

<Button Content="Show Selection" HorizontalAlignment="Left" Margin="10,168,0,0" VerticalAlignment="Top" Width="175" Command="{Binding ShowSelection}"/>

<!—Changes Visible from RadioButtonCheckBoxVM-->

<CheckBox Content="Visible Radiobuttons" HorizontalAlignment="Left" Margin="10,205,0,0" VerticalAlignment="Top" Width="175" IsChecked="{Binding Path=Visible}"/>

</Grid>

</Window>

## RadioButtonCheckBoxVM

public class RadioButtonCheckBoxVM : AViewModel

{

private ESelection selection;

private bool visible;

//Currently selected enum value

public ESelection Selection{

get

{

return selection;

}

set

{

selection = value;

CallPropertyChanged("Selection");

}

}

//Bool which determines if groupbox is visible or not

public bool Visible

{

get

{

return visible;

}

set

{

visible = value;

CallPropertyChanged("Visible");

}

}

public RadioButtonCheckBoxVM()

{

selection = ESelection.SECOND\_SELECTION;

visible = true;

}

//Shows the currently selected enum value

public RelayCommand ShowSelection

{

get

{

return new RelayCommand(

o =>

{

MessageBox.Show(selection.ToString());

},

o => true);

}

}

}

## Converter

public class EnumBooleanConverter : IValueConverter

{

/\*

\*value -> Selection from RadioButtonCheckBoxVM

\*parameter -> ConverterParameter (in the radiobuttons)

\*targetType -> type of the target property (not important for this example)

\*culture -> important for culture dependent conversions (not important for this example)

\*/

public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)

{

//checks if Selection from RadioButtonCheckBoxVM has the same value as the ConverterParameter. Returns true or false

return ((Enum)value).HasFlag((Enum)parameter);

}

/\*

\*value -> IsChecked from radiobutton (true/false)

\*parameter -> ConverterParameter (in the radiobuttons)

\*targetType -> type of the target property (not important for this example)

\*culture -> important for culture dependent conversions (not important for this example)

\*/

public object ConvertBack(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)

{

//If the radiobutton is checked, it returns the ConverterParameter

return value.Equals(true) ? parameter : Binding.DoNothing;

}

}

Jeder Converter erbt von IValueConverter. Es müssen die Methoden Convert und ConvertBack implementiert werden.

* Convert: Da die IsChecked-Property auf Selection von RadioButtonCheckBoxVM gebunden ist, muss die Convert-Methode einen bool-Wert zurückgeben. Dieser bestimmt, ob der RadioButton in der Oberfläche selektiert ist. Ein Enum wird zu einem Bool-Wert „umgewandelt“.
* ConvertBack: Bei einer Änderung des ausgewählten RadioButtons, muss Selection von RadioButtonCheckBoxVM aktualisiert werden. Hierfür muss ein Wert des Enums ESelection zurückgegeben werden. Ein Bool-Wert wird zu einem Enum „umgewandelt“.

# TextBox, TextBlock, Label

Das TextBox-Element wird für die Eingabe von Daten verwendet. Für das Darstellen dieser Werte können zwei ähnliche Elemente, welche sich jedoch unterscheiden, in Frage kommen.

* Label: Mithilfe von AccessKeys kann der Zugriff auf TextBox-Elemente erleichtert werden. Kann sowohl Text als auch andere Elemente (z.B.:Image) enthalten. Hat keine Text-Property; stattdessen wird Content verwendet.
* TextBlock: Nur für die Darstellung von Text gedacht. Hat Text-Property, welche nur strings akzeptiert (deswegen können nur strings gerendert werden).

## XAML-Code

<Window x:Class="TextBoxTextBlockLabel\_Example.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:TextBoxTextBlockLabel\_Example"

mc:Ignorable="d"

Title="MainWindow" Height="459.007" Width="525">

<Window.DataContext>

<local:TextBoxTextBlockLabelVM></local:TextBoxTextBlockLabelVM>

</Window.DataContext>

<Grid>

<!--Label bound to Heading from TextBoxTextBlockLabelVM-->

<Label Content="{Binding Heading}" HorizontalAlignment="Left" Margin="277,10,0,0" VerticalAlignment="Top" Width="230"/>

<!--TextBlock bound to Content form TextBoxTextBlockLabelVM-->

<TextBlock HorizontalAlignment="Left" Margin="277,41,0,0" TextWrapping="Wrap" Text="{Binding Content}" VerticalAlignment="Top" Height="156" Width="230"/>

<!--TextBox bound to Heading from TextBoxTextBlockLabelVM.-->

<TextBox x:Name="txt\_Heading" HorizontalAlignment="Left" Height="23" Margin="10,40,0,0" TextWrapping="Wrap" Text="{Binding Heading, Mode=TwoWay}" VerticalAlignment="Top" Width="216"/>

<!--Label that helps to access txt\_Heading easily (alt+H)-->

<Label HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top" Width="216" Target="{Binding ElementName=txt\_Heading}">

<StackPanel Orientation="Horizontal">

<Image Source="http://cdn1.iconfinder.com/data/icons/fatcow/16/bullet\_green.png" Height="10" Width="10"/>

<AccessText Text="\_Heading" />

</StackPanel>

</Label>

<!--Label that helps to access txt\_Content easily (alt+C)-->

<Label Content="\_Content" HorizontalAlignment="Left" Margin="10,68,0,0" VerticalAlignment="Top" Width="216" Target="{Binding ElementName=txt\_Content}"/>

<TextBox HorizontalAlignment="Left" Height="178" Margin="10,99,0,0" TextWrapping="Wrap" Text="{Binding Content, Mode=TwoWay}" VerticalAlignment="Top" Width="216" x:Name="txt\_Content"/>

<!--Command that clears Heading from TextBoxTextBlockLabelVM-->

<Button Content="Clear heading" HorizontalAlignment="Left" Margin="10,282,0,0" VerticalAlignment="Top" Width="216" Command="{Binding ClearHeading}"/>

<!--Command that clears Content from TextBoxTextBlockLabelVM-->

<Button Content="Clear content" HorizontalAlignment="Left" Margin="10,307,0,0" VerticalAlignment="Top" Width="216" Command="{Binding ClearContent}"/>

<!--Command that clears Content and Heading from TextBoxTextBlockLabelVM-->

<Button Content="Clear everything" HorizontalAlignment="Left" Margin="10,332,0,0" VerticalAlignment="Top" Width="216" Command="{Binding ClearBoth}"/>

<Rectangle Fill="#FFF4F4F5" HorizontalAlignment="Left" Height="1" Margin="279,31,0,0" Stroke="Black" VerticalAlignment="Top" Width="221" RenderTransformOrigin="0.5,0.5">

<Rectangle.RenderTransform>

<TransformGroup>

<ScaleTransform ScaleY="-1"/>

<SkewTransform/>

<RotateTransform/>

<TranslateTransform/>

</TransformGroup>

</Rectangle.RenderTransform>

</Rectangle>

</Grid>

</Window>

Um Labels mit TextBox-Elementen zu verbinden, muss die Target-Property verwendet werden. Da die Content-Property ein Underscore vor dem eigentlichen Text des Labels enthält, kann mittels „alt+<Anfangsbuchstabe des Texts>“ die zugehörige TextBox aufgerufen werden.

Falls ein Label weitere Elemente wie Bilder enthält, muss der „Suchtext“ für die gewünschte TextBox durch den AccessText-Tag angegeben werden.

## TextBoxTextBlockLabelVM

public class TextBoxTextBlockLabelVM:AViewModel

{

private string heading;

private string content;

//Content displayed in textblock

public string Content

{

get

{

return content;

}

set

{

content = value;

CallPropertyChanged("Content");

}

}

//Heading displayed in label

public string Heading

{

get

{

return heading;

}

set

{

heading = value;

CallPropertyChanged("Heading");

}

}

public TextBoxTextBlockLabelVM()

{

heading = "Beispiel";

content = "Das ist ein Beispiel!Das ist ein Beispiel!Das ist ein Beispiel!Das ist ein Beispiel!Das ist ein Beispiel!Das ist ein Beispiel!Das ist ein Beispiel!";

}

//Clears Heading

public RelayCommand ClearHeading

{

get

{

return new RelayCommand(

o =>

{

Heading = "";

},

o => true);

}

}

//Clears Content

public RelayCommand ClearContent

{

get

{

return new RelayCommand(

o =>

{

Content = "";

},

o => true);

}

}

//Clears Content and Heading

public RelayCommand ClearBoth

{

get

{

return new RelayCommand(

o =>

{

Heading = "";

Content = "";

},

o => true);

}

}

}

# Simple Slider Binding

Dieses Beispiel zeigt einfaches im XAML Slider-Binding, es ist kein C# Code dahinter. Man bindet hier in der TextBox direkt auf den Element-Namen des Sliders, ohne ViewModel-Klasse

<DockPanel VerticalAlignment="Center" Margin="10">

<TextBox Text="{Binding ElementName=slider, Path=Value, UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right" TextAlignment="Right" Width="40" />

<Slider Maximum="255" TickPlacement="BottomRight" TickFrequency="5" IsSnapToTickEnabled="True" Name="slider" />

</DockPanel>

# Slider Binding mit ViewModel

Dieses Beispiel zeigt einen **ColourPicker** mit einem Slidern, ViewModel, DataBinding und ICommand umgesetzt.

## **SliderVM.cs**

In der **SliderVM**-Klasse wird das **INotifyPropertyChanged**-Interface implementiert und jedes Mal wenn der rote, grüne oder blaue Wert sich ändert wird das **OnPropertyChanged**-Interface aufgerufen.

Es wird ein **Command** implementiert der bei Aufruf einen zufälligen Wert für die 3 Values setzt.

class SliderVM:INotifyPropertyChanged

{

private int redValue;

public int RedValue

{

get { return redValue; }

set

{

redValue = value;

OnPropertyChanged();

}

}

private int greenValue;

public int GreenValue

{

get { return greenValue; }

set

{

greenValue = value;

OnPropertyChanged();

}

}

private int blueValue;

public int BlueValue

{

get { return blueValue; }

set

{

blueValue = value;

OnPropertyChanged();

}

}

public RelayCommand RandomColourCommand

{

get

{

return new RelayCommand(

o =>

{

Random rnd = new Random();

RedValue = rnd.Next(1, 255);

GreenValue = rnd.Next(1, 255);

BlueValue = rnd.Next(1, 255);

},

o => true

);

}

}

public event PropertyChangedEventHandler PropertyChanged;

protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)

{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

}

}

## **RelayCommand.cs**

Implementiert wird eine einfache Variante des **RelayCommands** (ohne **CommandManager**).

public class RelayCommand : ICommand

{

private Func<object, bool> canExecute;

private Action<object> execute;

public event EventHandler CanExecuteChanged;

public RelayCommand(Action<object> execute,

Func<object, bool> canExecute)

{

this.canExecute = canExecute;

this.execute = execute;

}

public bool CanExecute(object parameter)

=> canExecute(parameter);

public void Execute(object parameter)

=> execute(parameter);

}

## **ColourSlider.xaml**

Auf den **RandomColour**-Button wird das **RandomColourCommand** gebunden. Danach folgen 3 Slider und zugehörige **Textboxen** die auf den Wert des Sliders gebunden werden. Das Binding der Slider selbst wird auf das jeweilige Property der ViewModel-Klasse gelegt.

<Window.DataContext>

<local:SliderVM></local:SliderVM>

</Window.DataContext>

<StackPanel Margin="10" VerticalAlignment="Center">

<Button x:Name="button" Content="Random Colour" Margin="60,0,60,20" Command="{Binding RandomColourCommand}" />

<DockPanel VerticalAlignment="Center" Margin="10">

<Label DockPanel.Dock="Left" FontWeight="Bold" Content="R:"/>

<TextBox Text="{Binding RedValue, UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right" TextAlignment="Right" Width="40" />

<Slider Maximum="255" TickPlacement="BottomRight" TickFrequency="5" Value="{Binding RedValue,Mode=TwoWay}" IsSnapToTickEnabled="True" x:Name="slider\_red" ValueChanged="ColorSlider\_ValueChanged"/>

</DockPanel>

<DockPanel VerticalAlignment="Center" Margin="10">

<Label DockPanel.Dock="Left" FontWeight="Bold" Content="G:"/>

<TextBox Text="{Binding GreenValue, UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right" TextAlignment="Right" Width="40" />

<Slider Maximum="255" TickPlacement="BottomRight" TickFrequency="5" Value="{Binding GreenValue,Mode=TwoWay}" IsSnapToTickEnabled="True" x:Name="slider\_green" ValueChanged="ColorSlider\_ValueChanged" />

</DockPanel>

<DockPanel VerticalAlignment="Center" Margin="10">

<Label DockPanel.Dock="Left" FontWeight="Bold" Content="B:"/>

<TextBox Text="{Binding BlueValue, UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right" TextAlignment="Right" Width="40" />

<Slider Maximum="255" TickPlacement="BottomRight" TickFrequency="5" Value="{Binding BlueValue,Mode=TwoWay}" IsSnapToTickEnabled="True" x:Name="slider\_blue" ValueChanged="ColorSlider\_ValueChanged" />

</DockPanel>

</StackPanel>

## **ColourSlider.xaml.cs**

Hier wird nur in der **ValueChanged**-Methode eine neue Hintergrundfarbe für die Werte der Slider gesetzt.

public partial class ColourSlider : Window

{

public ColourSlider()

{

InitializeComponent();

ValueChanged();

}

private void ColorSlider\_ValueChanged(object sender, RoutedPropertyChangedEventArgs<double> e)

{

ValueChanged();

}

private void ValueChanged()

{

Color color = Color.FromRgb((byte)slider\_red.Value, (byte)slider\_green.Value, (byte)slider\_blue.Value);

this.Background = new SolidColorBrush(color);

}

}

# ListBox

Die **ListBox**-Klasse ist ein WPF-Steuerelement, welches aus einer Collection von **ListBoxItems** besteht. Im folgendem Code-Beispiel wird gezeigt wie man eine ListBox mit einer **ViewModel**-Klasse binden kann.

## **Skill.cs**

Die Skill Klasse wird in diesem Beispiel für die Datensätze der **ListBox** verwendet. Das **INotifyPropertyChanged**-Interface wird implementiert, um auch das Updaten dieser Datensätze zu ermöglichen.

public class Skill:INotifyPropertyChanged

{

private string name;

public string Name

{

get => name;

set { name = value; OnPropertyChanged(); }

}

private int percent;

public int Percent

{

get => percent;

set { percent = value; OnPropertyChanged(); }

}

public event PropertyChangedEventHandler PropertyChanged;

[NotifyPropertyChangedInvocator]

protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)

{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

}

}

## **MainWindow.xaml.cs**

Hier wird nur eine Liste von Datensätzen erstellt und dem im XAML gesetzten **DataContext** hinzugefügt. Normalerweise würde man Datensätze direkt im ViewModel von einem Repository abrufen, in diesem Beispiel wurde allerdings auf eine das EntityFramework verzichtet.

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

//Initialize some Data

List<Skill> skills = new List<Skill>()

{

new Skill() {Name = "VueJS", Percent = 80},

new Skill() {Name = "ReactJS", Percent = 50},

new Skill() {Name = "Django", Percent = 20},

new Skill() {Name = "Rails", Percent = 60},

new Skill() {Name = "Angular", Percent = 70},

new Skill() {Name = "Aurelia", Percent = 30},

new Skill() {Name = "Backbone", Percent = 10},

new Skill() {Name = "Ember", Percent = 0},

};

var skillVm = DataContext as SkillVM;

skillVm.AddSkills(skills);

}

}

## **MainWindow.xaml**

Im XAML wird der **DataContext** gesetzt. Man binded außerdem die ObservableCollection der ViewModel-Klasse als **ItemsSource** der ListBox und als **SelectedItem** das **SelectedSkill**-Poperty der ViewModel-Klasse.

Es besteht die Möglichkeit ein eigenes **ItemTemplate** in der ListBox zu erstellen um nicht nur Text in einer ListBox anzeigen zu lassen. In diesem Beispiel wird zusätzlich zu einem **TextBlock** noch eine **ProgressBar** in einem **ListBoxItem** angezeigt. Diese Elemente werden zu den Properties der Skill-Objekte der **ObservableCollection** gebunden.

<Window.DataContext>

<local:SkillVM/>

</Window.DataContext>

<Grid>

<ListBox Name="lb\_skills" HorizontalContentAlignment="Stretch" Margin="0,0,264.4,-0.2" ItemsSource="{Binding Skills}" SelectedItem="{Binding SelectedSkill}">

<ListBox.ItemTemplate>

<DataTemplate>

<Grid Margin="0,2">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="100" />

</Grid.ColumnDefinitions>

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

<ProgressBar Grid.Column="1" Minimum="0" Maximum="100" Value="{Binding Percent}" />

</Grid>

</DataTemplate>

</ListBox.ItemTemplate>

</ListBox>

<Label Content="Skill" HorizontalAlignment="Left" Margin="259,10,0,0" VerticalAlignment="Top"/>

<TextBox HorizontalAlignment="Left" Height="23" Margin="289,13,0,0" TextWrapping="Wrap" Text="{Binding Skill.Name,UpdateSourceTrigger=PropertyChanged}" VerticalAlignment="Top" Width="120"/>

<Button Content="Add" HorizontalAlignment="Left" Margin="414,13,0,0" VerticalAlignment="Top" Width="75" Command="{Binding AddSkillCommand}"/>

<Rectangle Fill="#FFF4F4F5" HorizontalAlignment="Left" Height="3" Margin="254,52,0,0" Stroke="Black" VerticalAlignment="Top" Width="264"/>

<TextBlock HorizontalAlignment="Left" Margin="360,60,0,0" TextWrapping="Wrap" Text="{Binding SelectedSkill.Name}" VerticalAlignment="Top"/>

<Button Content="Add Percent" HorizontalAlignment="Left" Margin="268,92,0,0" VerticalAlignment="Top" Width="105" Command="{Binding AddPercentageCommand}"/>

<Button Content="Remove Percent" HorizontalAlignment="Left" Margin="401,92,0,0" VerticalAlignment="Top" Width="107" Command="{Binding RemovePercentageCommand}" />

</Grid>

## **RelayCommand.cs**

Das **RelayCommand** implementiert das **ICommand**-Interface. Es gibt zwei Konstruktoren, einen ohne dem **CanExecute**-Predicate und einmal mit.   
Wichtig bei dieser Implementierung des RelayCommands ist das **CanExecuteChanged**-Event, um auf Änderungen im CanExecute-Predicate während der Laufzeit zu reagieren.

public class RelayCommand : ICommand

{

private readonly Action<object> \_execute;

private readonly Predicate<object> \_canExecute;

public RelayCommand(Action<object> execute) : this(execute, null) { }

public RelayCommand(Action<object> execute, Predicate<object> canExecute)

{

\_execute = execute ?? throw new ArgumentNullException(nameof(execute));

\_canExecute = canExecute;

}

public bool CanExecute(object parameter)

{

return \_canExecute?.Invoke(parameter) ?? true;

}

public event EventHandler CanExecuteChanged

{

add => CommandManager.RequerySuggested += value;

remove => CommandManager.RequerySuggested -= value;

}

public void Execute(object parameter)

{

\_execute(parameter);

}

}

## **SkillVM.cs**

**SkillVM** hat eine ObservableCollection von Skills, die in der ListBox angezeigt werden. Es gibt 2 konkrete Skill-Objekte: **Skill**, um einen neuen Listeneintrag anzulegen und **SelectedSkill**, um einen bestehenden Skill aus der Liste auszuwählen und zu bearbeiten. Das **INotifyPropertyChanged** wird implementiert.

Die **AddSkills**-Methode ist nur dazu da, am Anfang Daten einzufügen.

Das Command **AddSKillCommand** fügt einen Skill hinzu, es kann nur ausgeführt werden, wenn der Name des Skills nicht null oder ““ ist.

Die Commands **AddPercentageCommand** und **RemovePercentage** Command fügen oder entfernen jeweils 10 Prozent und werden nur ausgeführt wenn ein Skill ausgewählt ist und die Prozent nicht über 100 oder unter 0 gesetzt werden.

class SkillVM:INotifyPropertyChanged

{

private Skill skill;

public Skill Skill

{

get => skill;

set

{

skill = value;

OnPropertyChanged();

}

}

private Skill selectedSkill=null;

public Skill SelectedSkill

{

get => selectedSkill;

set

{

selectedSkill = value;

OnPropertyChanged();

}

}

public ObservableCollection<Skill> Skills { get; set; }=new ObservableCollection<Skill>();

public SkillVM()

{

Skill=new Skill();

}

public SkillVM(List<Skill> skills)

{

Skills=new ObservableCollection<Skill>(skills);

Skill=new Skill();

}

public void AddSkills(List<Skill>skills)

{

foreach (var item in skills)

{

Skills.Add(item);

}

}

public ICommand AddSkillCommand

{

get { return new RelayCommand(o=>AddSkill(),o=>!string.IsNullOrEmpty(skill.Name));}

}

private void AddSkill()

{

Skills.Add(new Skill(){Name = skill.Name,Percent = 0});

Skill=new Skill();

}

public ICommand AddPercentageCommand

{

get { return new RelayCommand(o=>AddPercentage(),o=>SelectedSkill!=null&&SelectedSkill.Percent<91);}

}

private void AddPercentage()

{

SelectedSkill.Percent += 10;

}

public ICommand RemovePercentageCommand

{

get { return new RelayCommand(o => RemovePercentage(), o => SelectedSkill != null && SelectedSkill.Percent >9); }

}

private void RemovePercentage()

{

SelectedSkill.Percent -= 10;

}

public event PropertyChangedEventHandler PropertyChanged;

[NotifyPropertyChangedInvocator]

protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)

{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

}

}

# DataGrid

In diesem Beispiel wird gezeigt wie man ein DataGrid mit Werten aus einer Datenbank füllen kann, und diese Werte Löschen, Bearbeiten und Hinzufügen. EntityFramework 6.2 wird verwendet.

## **User.cs**

Die Klasse User ist die Klasse die wir in unserem DataGrid anzeigen lassen wollen.   
Wichtig: BirthDate mit dem TypeName „datetime2“, da das defaultmäßige „datetime“ erst beim Jahr 1753 beginnt und „datetime2“ bei 0001. Im Normalfall stellt das kein Problem dar, allerdings beginnt der WPF eigene DatePicker, der in diesem Beispiel benutzt wird, auch beim Jahr 0001 und es kann zu Parsing-Errors kommen.

[Table("users")]

public partial class User

{

[Key]

public int UserId { get; set; }

[Required]

public string FirstName { get; set; }

[Required]

public string LastName { get; set; }

[Required]

[Column("e-mail-adress")]

public string Email { get; set; }

[Required]

public string UserName { get; set; }

[Required]

[Column(TypeName = "datetime2")]

public DateTime BirthDate { get; set; }

}

public partial class User

{

[NotMapped]

public string FullName

{

get { return FirstName + " " + LastName; }

}

}

## **BaseRepo.cs**

Die folgende BaseRepo wird für dieses Beispiel verwendet.

public abstract class BaseRepo<T> : IDisposable where T : class, new()

{

public ContextClass Context { get; } = ContextClass.GetInstance();

protected DbSet<T> Table;

public T GetOne(int? id) => Table.Find(id);

public List<T> GetAll() => Table.ToList();

public int Add(T entity)

{

Table.Add(entity);

return SaveChanges();

}

public int AddRange(IEnumerable<T> entities)

{

Table.AddRange(entities);

return SaveChanges();

}

public int AddRange(IList<T> entities)

{

Table.AddRange(entities);

return SaveChanges();

}

public int Save(T entity)

{

Context.Entry(entity).State = System.Data.Entity.EntityState.Modified;

return SaveChanges();

}

public int Delete(T entity)

{

Table.Remove(entity);

return SaveChanges();

}

public List<T> ExecuteQuery(string sql) => Table.SqlQuery(sql).ToList();

public List<T> ExecuteQuery(string sql, object[] sqlParametersObjects)

{

return Table.SqlQuery(sql, sqlParametersObjects).ToList();

}

internal int SaveChanges()

{

return Context.SaveChanges();

}

bool \_disposed = false;

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)

{

if (!this.\_disposed)

{

if (disposing)

{

ContextClass.GetInstance().Dispose();

}

}

this.\_disposed = true;

}

}

## **UserRepo.cs**

Die UserRepo erbt von BaseRepo und wird als Singleton implementiert um unnötige Kosten zu sparen, die ein häufiges Instanzieren mit sich bringen würde.

public class UserRepo:BaseRepo<User>

{

private static UserRepo repo;

public static UserRepo GetInstance()

{

return repo ?? (repo=new UserRepo());

}

public UserRepo()

{

Table = Context.Users;

}

}

## **ContextClass.cs**

Die Context-Klasse der Anwendung erbt von DbContext und wird auch als Singleton implementiert. Im Konstruktor wird ein eigener Intializer festgelegt.

public class ContextClass:DbContext

{

public DbSet<User> Users { get; set; }

private static ContextClass context;

public static ContextClass GetInstance()

{

return context ?? (context = new ContextClass());

}

public ContextClass() : base("Datagrid\_Example")

{

Database.SetInitializer(new CustomInitializer());

Database.Initialize(true);

}

}

## **CustomInitializer.cs**

Der Initializer erbt von DropCreateDatabaseIfModelChanges, um nicht bei jedem Starten eine neue Datenbank zu erstellen. In der Seed Methode werden einige Werte in die Datenbank eingefügt.

class CustomInitializer: DropCreateDatabaseIfModelChanges<ContextClass>

{

protected override void Seed(ContextClass context)

{

List<User> users=new List<User>()

{

new User(){BirthDate = DateTime.Today,Email = "b.adams@gmail.com",FirstName = "Baker",LastName = "Adams",UserName = "badams"},

new User(){BirthDate = DateTime.Today,Email = "c.davis@gmail.com",FirstName = "Clark",LastName = "Davis",UserName = "cavis"},

new User(){BirthDate = DateTime.Today,Email = "e.frank@gmail.com",FirstName = "Evans",LastName = "Frank",UserName = "frevans"},

new User(){BirthDate = DateTime.Today,Email = "g.hills@gmail.com",FirstName = "Ghosh",LastName = "Hills",UserName = "gills"},

new User(){BirthDate = DateTime.Today,Email = "i.jones@gmail.com",FirstName = "Irwin",LastName = "Jones",UserName = "jirwin"},

new User(){BirthDate = DateTime.Today,Email = "k.lopez@gmail.com",FirstName = "Klein",LastName = "Lopez",UserName = "klopez"},

new User(){BirthDate = DateTime.Today,Email = "m.nalty@gmail.com",FirstName = "Mason",LastName = "Nalty",UserName = "malty"},

new User(){BirthDate = DateTime.Today,Email = "o.patel@gmail.com",FirstName = "Ochoa",LastName = "Patel",UserName = "opatel"},

new User(){BirthDate = DateTime.Today,Email = "q.reily@gmail.com",FirstName = "Quinn",LastName = "Reily",UserName = "queilly"},

new User(){BirthDate = DateTime.Today,Email = "s.trott@gmail.com",FirstName = "Smith",LastName = "Trott",UserName = "schrott"},

new User(){BirthDate = DateTime.Today,Email = "u.valdo@gmail.com",FirstName = "Usman",LastName = "Valdo",UserName = "uvaldo"},

};

UserRepo.GetInstance().AddRange(users);

}

}

## **AViewModel.cs**

In der abstrakten Klasse AViewModel wird das INotifyPropertyChanged-Interface implementiert.

public abstract class AViewModel:INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged;

[NotifyPropertyChangedInvocator]

protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)

{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

}

}

## **RelayCommand.cs**

Das **RelayCommand** implementiert das **ICommand**-Interface. Es gibt zwei Konstruktoren, einen ohne dem **CanExecute**-Predicate und einmal mit.   
Wichtig bei dieser Implementierung des RelayCommands ist das **CanExecuteChanged**-Event, um auf Änderungen im CanExecute-Predicate während der Laufzeit zu reagieren.

public class RelayCommand : ICommand

{

private readonly Action<object> \_execute;

private readonly Predicate<object> \_canExecute;

public RelayCommand(Action<object> execute) : this(execute, null) { }

public RelayCommand(Action<object> execute, Predicate<object> canExecute)

{

\_execute = execute ?? throw new ArgumentNullException(nameof(execute));

\_canExecute = canExecute;

}

public bool CanExecute(object parameter)

{

return \_canExecute?.Invoke(parameter) ?? true;

}

public event EventHandler CanExecuteChanged

{

add => CommandManager.RequerySuggested += value;

remove => CommandManager.RequerySuggested -= value;

}

public void Execute(object parameter)

{

\_execute(parameter);

}

}

## **UserVM.cs**

Das ViewModel für einen einzelnen User enthält Properties für alle Werte des Users die in der View ausgegeben werden.

public class UserVM:AViewModel

{

public User User { get; }

public UserVM(User user)

{

this.User = user;

}

public string FirstName

{

get => User.FirstName;

set

{

User.FirstName = value;

OnPropertyChanged();

}

}

public string LastName

{

get => User.LastName;

set

{

User.LastName = value;

OnPropertyChanged();

}

}

public string Email

{

get => User.Email;

set

{

User.Email = value;

OnPropertyChanged();

}

}

public string UserName

{

get => User.UserName;

set

{

User.UserName = value;

OnPropertyChanged();

}

}

public DateTime BirthDate

{

get => User.BirthDate;

set

{

User.BirthDate = value;

OnPropertyChanged();

}

}

public string FullName => User.FullName;

}

## **UsersVM.cs**

UsersVM hat eine ein Objekt der UserVM Klasse, um auf die ausgewählte Zeile des DataGrids zugreifen zu können. Außerdem eine ObservableCollection von UserVMs.

Es werden ICommands erstellt um User löschen, hinzufügen und bearbeiten zu können, und Commands um die verschiedenen Views aufzurufen. Wichtig hierbei ist, das den Views die aufgerufen werden die UsersVM Klasse als DataContext mitgegeben wird.

public class UsersVM : AViewModel

{

private UserVM \_user;

public UserVM User

{

get => \_user;

set

{

if(value==\_user)return;

\_user = value;

OnPropertyChanged();

}

}

private readonly UserRepo \_repo = UserRepo.GetInstance();

public ObservableCollection<UserVM> Users { get; set; } = new ObservableCollection<UserVM>();

public UsersVM()

{

\_repo.GetAll().ForEach(u => Users.Add(new UserVM(u)));

User = null;

}

public ICommand RemoveUserCommand

{

get { return new RelayCommand(o => DeleteUser(), o => User!=null); }

}

private void DeleteUser()

{

\_repo.Delete(User.User);

Users.Remove(User);

}

public ICommand OpenCreateUserViewCommand

{

get { return new RelayCommand(o => ShowCreateUserView()); }

}

private void ShowCreateUserView()

{

User = new UserVM(new User());

var createUserView = new CreateUserView {DataContext = this};

createUserView.ShowDialog();

User = null;

}

public ICommand CreateUserCommand

{

get { return new RelayCommand(o => CreateUser()); }

}

private void CreateUser()

{

var u = new User

{

FirstName = User.FirstName,

LastName = User.LastName,

UserName = User.UserName,

Email = User.Email,

BirthDate = User.BirthDate

};

\_repo.Add(u);

Users.Add(new UserVM(\_repo.GetOne(u.UserId)));

User = null;

}

public ICommand OpenUpdateUserViewCommand

{

get { return new RelayCommand(o => ShowUpdateUserView(), o => User!=null); }

}

private void ShowUpdateUserView()

{

var updateUserView = new UpdateUserView() {DataContext = this};

updateUserView.ShowDialog();

}

public ICommand UpdateUserCommand

{

get { return new RelayCommand(o => UpdateUser()); }

}

private void UpdateUser()

{

var u = User.User;

\_repo.Save(u);

User = null;

}

public ICommand OpenUserViewCommand

{

get { return new RelayCommand(o => ShowUserView(), o => User != null); }

}

private void ShowUserView()

{

var userView = new UserView() { DataContext = this };

userView.ShowDialog();

}

## **MainWindow.xaml**

Im MainWindow wird das UserControl UsersView augegeben.

<Window x:Class="Datagrid\_Databinding\_Example.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:Datagrid\_Databinding\_Example"

xmlns:view="clr-namespace:Datagrid\_Databinding\_Example.View"

mc:Ignorable="d"

Title="MainWindow" Height="410" Width="525">

<Grid>

<view:UsersView VerticalAlignment="Center"/>

</Grid>

</Window>

## **UsersView.xaml**

Als erstes muss der DataContext gesetzt werden. Danach wird für die ItemsSource des DataGrid die ObservableCollection von UserVMs festgelegt und als SelectedItem User. Wichtig ist, dass das Attribut AutoGeneratedColumns auf false gesetzt wird, um zu verhindern das Spalten angelegt werden die nicht explizit definiert wurden. Im nächsten Schritt werden mit dem XAML-Element DataGridTextColumn die Spalten explizit festgelegt und auf das jeweilige Property der UserVM-Klasse gebinded.

Die Buttons werden mit den Commands festgelegt die die Views öffnen, bzw. Datensätze entfernen.

<UserControl x:Class="Datagrid\_Databinding\_Example.View.UsersView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:local="clr-namespace:Datagrid\_Databinding\_Example.View"

xmlns:viewmodel="clr-namespace:Datagrid\_Databinding\_Example.ViewModel"

mc:Ignorable="d"

d:DesignHeight="350" d:DesignWidth="520">

<UserControl.DataContext>

<viewmodel:UsersVM/>

</UserControl.DataContext>

<Grid>

<DataGrid x:Name="dgr\_users" HorizontalAlignment="Center"

Height="263" Margin="10,10,0,0" VerticalAlignment="Top" Width="500"

ItemsSource="{Binding Users}" AutoGenerateColumns="False" SelectedItem="{Binding User,Mode=TwoWay}"

SelectionMode="Extended" SelectionUnit="FullRow">

<DataGrid.Columns>

<DataGridTextColumn Header="FirstName" Binding="{Binding FirstName}"/>

<DataGridTextColumn Header="LastName" Binding="{Binding LastName}" />

<DataGridTextColumn Header="UserName" Binding="{Binding UserName}" />

<DataGridTextColumn Header="E-Mail" Binding="{Binding Email}" />

<DataGridTextColumn Header="Birthday" Binding="{Binding BirthDate}" />

</DataGrid.Columns>

</DataGrid>

<Button Name="btn\_delete" Content="Delete" Command="{Binding RemoveUserCommand}" HorizontalAlignment="Left" Margin="15,278,0,-18" VerticalAlignment="Top" Width="104" Height="40"/>

<Button x:Name="btn\_create" Content="Create" Command="{Binding OpenCreateUserViewCommand}" HorizontalAlignment="Left" Margin="145,278,0,0" VerticalAlignment="Top" Width="104" Height="40"/>

<Button x:Name="btn\_update" Content="Update" Command="{Binding OpenUpdateUserViewCommand}" HorizontalAlignment="Left" Margin="270,278,0,0" VerticalAlignment="Top" Width="104" Height="40"/>

<Button x:Name="btn\_detail" Content="Detail" Command="{Binding OpenUserViewCommand}" HorizontalAlignment="Left" Margin="395,278,0,0" VerticalAlignment="Top" Width="104" Height="40"/>

</Grid>

</UserControl>

## **UserView.xaml**

Die UserView dient ist nur dazu da, die Werte eines einzelnen Users anzuzeigen.

<Grid>

<Label Content="{Binding User.FullName}" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top" Width="272"/>

<Rectangle Fill="#FFF4F4F5" HorizontalAlignment="Left" Height="1" Margin="10,36,0,0" Stroke="Black" VerticalAlignment="Top" Width="272"/>

<Label Content="{Binding User.UserName}" HorizontalAlignment="Left" Margin="84,42,0,0" VerticalAlignment="Top" Width="198"/>

<Label Content="{Binding User.Email}" HorizontalAlignment="Left" Margin="73,73,0,0" VerticalAlignment="Top" Width="209"/>

<Label Content="UserName:" HorizontalAlignment="Left" Margin="10,42,0,0" VerticalAlignment="Top"/>

<Label Content="E-Mail:" HorizontalAlignment="Left" Margin="10,73,0,0" VerticalAlignment="Top"/>

<Label Content="BirthDate:" HorizontalAlignment="Left" Margin="10,104,0,0" VerticalAlignment="Top"/>

<Label Content="{Binding User.BirthDate}" HorizontalAlignment="Left" Margin="73,104,0,0" VerticalAlignment="Top" Width="209"/>

</Grid>

## **CreateUserView.xaml**

In dieser View werden neue User erstellt. Die Textboxen werden auf die Properties des UserVM-Objektes gebunden.

<Grid>

<TextBox x:Name="txt\_firstName" HorizontalAlignment="Left" Height="23" Margin="10,10,0,0" TextWrapping="Wrap" Text="{Binding User.FirstName, Mode=TwoWay}" VerticalAlignment="Top" Width="120"/>

<TextBox x:Name="txt\_lastName" HorizontalAlignment="Left" Height="23" Margin="10,38,0,0" TextWrapping="Wrap" Text="{Binding User.LastName, Mode=TwoWay}" VerticalAlignment="Top" Width="120"/>

<TextBox x:Name="txt\_userName" HorizontalAlignment="Left" Height="23" Margin="10,66,0,0" TextWrapping="Wrap" Text="{Binding User.UserName, Mode=TwoWay}" VerticalAlignment="Top" Width="120"/>

<TextBox x:Name="txt\_email" HorizontalAlignment="Left" Height="23" Margin="10,94,0,0" TextWrapping="Wrap" Text="{Binding User.Email, Mode=TwoWay}" VerticalAlignment="Top" Width="120"/>

<DatePicker Name="dp\_birthdate" HorizontalAlignment="Left" DisplayDateStart="01.01.1980" Margin="10,122,0,0" VerticalAlignment="Top" Width="120" SelectedDate="{Binding User.BirthDate}" Loaded="dp\_birthdate\_Loaded"/>

<Button Content="Create" HorizontalAlignment="Left" Margin="10,142,0,0" VerticalAlignment="Top" Width="120" Command="{Binding CreateUserCommand}" Click="Button\_Click"/>

<Label Content="FirstName" HorizontalAlignment="Left" Margin="135,7,0,0" VerticalAlignment="Top"/>

<Label Content="LastName" HorizontalAlignment="Left" Margin="135,35,0,0" VerticalAlignment="Top"/>

<Label Content="UserName" HorizontalAlignment="Left" Margin="135,61,0,0" VerticalAlignment="Top"/>

<Label Content="Email" HorizontalAlignment="Left" Margin="138,87,0,0" VerticalAlignment="Top"/>

</Grid>

## **CreateUserView.xaml.cs**

Hier wird nur definiert das das Fenster nach dem Anlegen eines Users geschlossen wird und das der DatePicker als Start-Datum das heutige hat und nicht im Jahr 0001 anfängt.

public partial class CreateUserView : Window

{

public CreateUserView()

{

InitializeComponent();

}

private void Button\_Click(object sender, RoutedEventArgs e)

{

this.Close();

}

private void dp\_birthdate\_Loaded(object sender, RoutedEventArgs e)

{

dp\_birthdate = sender as DatePicker;

dp\_birthdate.DisplayDate=DateTime.Now;

dp\_birthdate.SelectedDate=DateTime.Now;

}

}

## **UpdateUserView.xaml**

In dieser View werden User editiert. Die Textboxen werden auf die Properties des UserVM-Objektes gebunden.

<Grid>

<TextBox x:Name="txt\_firstName" HorizontalAlignment="Left" Height="23" Margin="10,10,0,0" TextWrapping="Wrap" Text="{Binding User.FirstName, Mode=TwoWay}" VerticalAlignment="Top" Width="120"/>

<TextBox x:Name="txt\_lastName" HorizontalAlignment="Left" Height="23" Margin="10,38,0,0" TextWrapping="Wrap" Text="{Binding User.LastName, Mode=TwoWay}" VerticalAlignment="Top" Width="120"/>

<TextBox x:Name="txt\_userName" HorizontalAlignment="Left" Height="23" Margin="10,66,0,0" TextWrapping="Wrap" Text="{Binding User.UserName, Mode=TwoWay}" VerticalAlignment="Top" Width="120"/>

<TextBox x:Name="txt\_email" HorizontalAlignment="Left" Height="23" Margin="10,94,0,0" TextWrapping="Wrap" Text="{Binding User.Email, Mode=TwoWay}" VerticalAlignment="Top" Width="120"/>

<DatePicker Name="dp\_birthdate" HorizontalAlignment="Left" DisplayDateStart="01.01.1980" Margin="10,122,0,0" VerticalAlignment="Top" Width="120" SelectedDate="{Binding User.BirthDate}"/>

<Button Content="Update" HorizontalAlignment="Left" Margin="10,142,0,0" VerticalAlignment="Top" Width="120" Command="{Binding UpdateUserCommand}" Click="Button\_Click"/>

<Label Content="FirstName" HorizontalAlignment="Left" Margin="135,7,0,0" VerticalAlignment="Top"/>

<Label Content="LastName" HorizontalAlignment="Left" Margin="135,35,0,0" VerticalAlignment="Top"/>

<Label Content="UserName" HorizontalAlignment="Left" Margin="135,61,0,0" VerticalAlignment="Top"/>

<Label Content="Email" HorizontalAlignment="Left" Margin="138,87,0,0" VerticalAlignment="Top"/>

</Grid>

## **UpdateUserView.xaml.cs**

Hier wird festgelegt das nach dem bearbeiten des Users das Fenster wieder geschlossen wird.

public partial class UpdateUserView : Window

{

public UpdateUserView()

{

InitializeComponent();

}

private void Button\_Click(object sender, RoutedEventArgs e)

{

this.Close();

}

}