WPF

Data Binding

Data Binding geschieht über den DataContext.

SomeWindow.xaml.cs

```
public partial class SomeWindow : Window
{
    public SomeWindow()
    {
        InitializeComponent();
        var viewModel = new MyViewModel(new MyService());
        this.DataContext = viewModel;
        //Gets a value that indicates whether this element has been loaded for
presentation.
        this.Loaded += async (sender, routedEventArgs) => await
viewModel.InitializeDataAsync();
    }
}
```

MyViewModel.cs

```
public class MyViewModel
{
    private IMyService myService;

    public MyViewModel(IMyService myService)
    {
        this.myService = myService ?? throw new
ArgumentException(nameof(myService));
    }

    public ObservableCollection<Data> Data {get; private set;}

    public async Task InitializeDataAsync(){
        foreach(var item in await myService.GetAllDataAsync())
        {
            this.Data.Add(item);
        }
    }
}
```

Das kann dann wie folgt benutzt werden

```
<ListBox ItemsSource="{Binding Data}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal" Margin="0,6">
                <TextBlock Text="{Binding Property1}"/>
                <TextBlock Text="{Binding Property2}"></TextBlock>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Es gibt OneWay, TwoWay und OneWayToSource Binding.

OneWay binding causes changes to the source property to automatically update the target property, but changes to the target property are not propagated back to the source property. This type of binding is appropriate if the control being bound is implicitly read-only. For instance, you may bind to a source such as a stock ticker, or perhaps your target property has no control interface provided for making changes, such as a data-bound background color of a table. If there's no need to monitor the changes of the target property, using the OneWay binding mode avoids the overhead of the TwoWay binding mode

TwoWay binding causes changes to either the source property or the target property to automatically update the other. This type of binding is appropriate for editable forms or other fully interactive UI scenarios. Most properties default to OneWay binding, but some dependency properties (typically properties of user-editable controls such as the TextBox.Text and CheckBox.IsChecked default to TwoWay binding.

OneWayToSource is the reverse of OneWay binding; it updates the source property when the target property changes. One example scenario is if you only need to reevaluate the source value from the UI.

Beispiel: Wir haben ein Textfeld, wo man was eingeben kann

MyViewModel.cs

```
public event PropertyChangedEventHandler PropertyChanged;
private string currentInput;
public string CurrentInput {
    get => currentInput;
    set
    {
        currentInput = value;
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(nameof(ProperyChanged)));
    }
}
```

Binding.UpdateSourceTrigger gibt an, durch was das Update getriggert wird.

```
<TextBox Grid.Column="0" FontSize="14" Text="{Binding CurrentInput, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}"/>
```

Für mehr Informationen zu DataBinding:

https://docs.microsoft.com/en-us/dotnet/desktop/wpf/data/?view=netdesktop-6.0
https://docs.microsoft.com/en-us/dotnet/api/system.windows.frameworkelement.isloaded?view=windowsdesktop-6.0

Commands

Commands haben mehrere Zwecke. Der erste ist, dass mit ihnen die das Objekt, das das Command auslöst und die Logik, die das Command ausführt, getrennt sind. So können mehrere Quellen dieselbe Logik auslösen. Zudem können mit Commands angegeben werden, ob eine bestimmte Aktion verfügbar ist oder nicht. (z.B.Ctr+X macht nur Sinn, wenn etwas ausgewählt ist)

Um ein Command selbst zu schreiben, muss das Interface ICommand implementiert werden. ICommand hat zwei Methoden und ein Event:

`CanExecute(Object)`: Defines the method that determines whether the command can execute in its current state.

`Execute(Object)`: Defines the method to be called when the command is invoked.

`CanExecuteChanged`: Occurs when changes occur that affect whether or not the command should execute.

Um das Beispiel von oben zu erweitern, soll es jetzt einen Button geben, den man drücken kann, um den eingegebenen Text an das Backend zu schicken. MyCommand.cs

```
public class MyCommand : ICommand
{
    private readonly Func<object, Task> executeAsync;
    private readonly Predicate<object> canExecute;

    public event EventHandler CanExecuteChanged
    {
        //CommandManager: Provides command related utility methods that register
CommandBinding and InputBinding objects for class owners and commands, add and
remove command event handlers, and provides services for querying the status of a
command.
        add {CommandManager.RequerySuggested += value;}
        remove {CommandManager.RequerySuggested -= value}
    }

    public MyCommand(Func<object, Task> executeAsync, Predicate<object> canExecute
= null)
    {
        this.executeAsync = executeAsync ?? throw new
ArgumentNullException(nameof(executeAsync));
        this.canExecute = canExecute;
    }

    public bool CanExecute(object parameter)
```

```
    {
        return this.canExecute == null || this.canExecute(parameter);
    }

    public async Task ExecuteAsync(object parameter)
    {
        await this.executeAsync(parameter);
    }

    async void ICommand.Execute(object parameter)
    {
        await this.ExecuteAsync(parameter);
    }
}
```

MyViewModel.cs

```
public ICommand DoSomethingImportantCommand {get; private set;}

public event PropertyChangedEventHandler PropertyChanged;
private string currentInput;
public string CurrentInput {
    get => currentInput;
    set
    {
        currentInput = value;
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(nameof(PropertyChanged)));
    }
}

public MyViewModel(IMyService myService)
    {
        Data = new ObservableCollection<Data>();
        this.myService = myService ?? throw new
ArgumentException(nameof(myService));
        this.DoSomethingImportantCommand = new MyCommand(this.SendStringAsync, _
=> this.Data != null);
    }

private async Task SendStringAsync(object arg)
{
    //not pretty, element is created in service and gets inserted into
ObservableList
    this.Data.Add(await myService.SendStringAsync(this.CurrentInput));
}
```

```
<TextBox Grid.Column="0" FontSize="14" Text="{Binding CurrentInput, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}" Margin="10,250,663,144"/>
<Button Content="Send" Width="80" Margin="10,316,710,80" Command="{Binding
DoSomethingImportantCommand}"/>
```

Für mehr Informationen:

https://docs.microsoft.com/en-us/dotnet/desktop/wpf/advanced/commanding-overview?view=netframeworkdesktop-4.8

https://docs.microsoft.com/en-us/dotnet/api/system.windows.input.icommand?view=net-6.0

Trigger

Mit Trigger können Properties verändert werden, wenn eine bestimmte Condition erfüllt ist. Mit Triggern kann man Dinge direkt im Xaml-File tun, die sonst nur im Code behind möglich sind. Es gibt Property Triggers, Event Triggers und Data Triggers, in der Übung haben wir aber nur den Data Trigger benutzt.

DataTriggers (`<DataTrigger>`) werden verwendet, wenn ein Trigger ausgelöst werden soll, weil ein Property einen bestimmten Wert hat.

```
<TextBlock Margin="371,276,67,40">
    <TextBlock.Style>
        <Style TargetType="TextBlock">
            <Setter Property="Text" Value=""/>
            <Style.Triggers>
                <DataTrigger Binding="{Binding CurrentInput}" Value="42">
                    <Setter Property="Text" Value="The answer for everything">
</Setter>
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </TextBlock.Style>
</TextBlock>
```

Für mehr Informationen:

https://docs.microsoft.com/en-us/dotnet/api/system.windows.datatrigger?view=windowsdesktop-6.0

Nützliche Informationen

Falls beim KT gefragt ist, wie beim Chat aus der Übung links eine Liste zu machen und rechts das ausgewählte Element anzuzeigen, so braucht man ein Property im ViewModel.

```
public event PropertyChangedEventHandler PropertyChanged;
private Data selectedItem;
        public Data SelectedItem
        {
```

```
            get => selectedItem;
            set
            {
                selectedItem = value;
                PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(nameof(PropertyChanged)));
            }
        }
```

Im Xaml nimmt man eine ListBox (siehe oben) und setzt SelectedItem.

```
<ListBox ItemsSource="{Binding Data}" SelectedItem="{Binding SelectedItem}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal" Margin="0,6">
                <TextBlock Text="{Binding Property1}"/>
                <TextBlock Text="{Binding Property2}"></TextBlock>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Jetzt wird hier immer das ausgewählte Element angezeigt.

```
<TextBlock Text="{Binding SelectedItem.Property1}" Margin="666,344,38,0"
Height="80" VerticalAlignment="Top"/>
```