Beispielprojekt WPF-Anwendung

Dieses Projekt soll die Grundsätze von WPF Anwendungen zeigen und wie man das MVVM Model in einer Anwendung benutzt

Inhaltsverzeichnis / list of contents

Erste Überlegungen

Zunächst muss man festlegen was das Programm denn alles können muss:

- Aufträge erstellen und anzeigen
- Aufträge durchführen

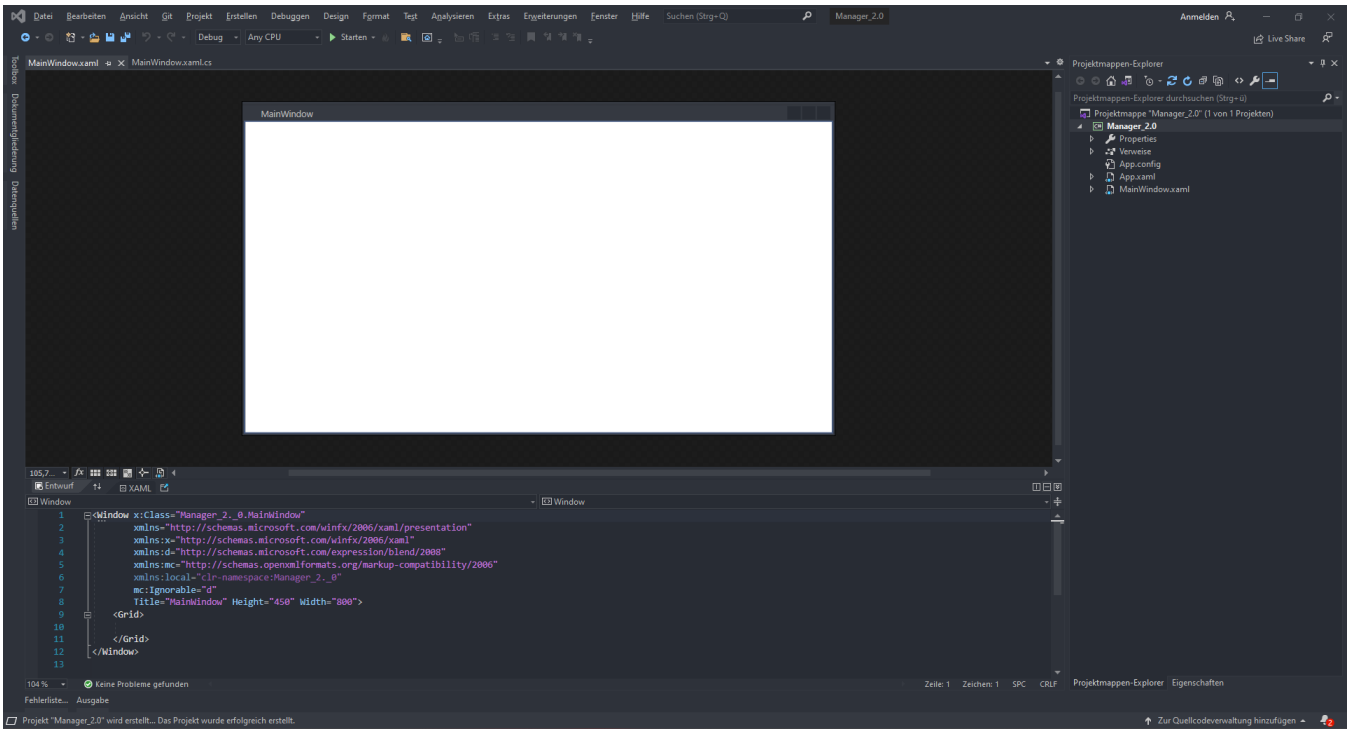
Der Anfang

Nach den ersten Überlegungen kann man schon loslegen

Zunächst erstellt man ein neues Projekt in Visual Studio dazu muss man nur auf **Datei Neu Projekt WPF App**

Bitte wähle einen geeigneten Projektnamen für die Anwendung!

Nach der Erstellung Sieht man zunächst das hier



Erstellung der ganzen Ordner

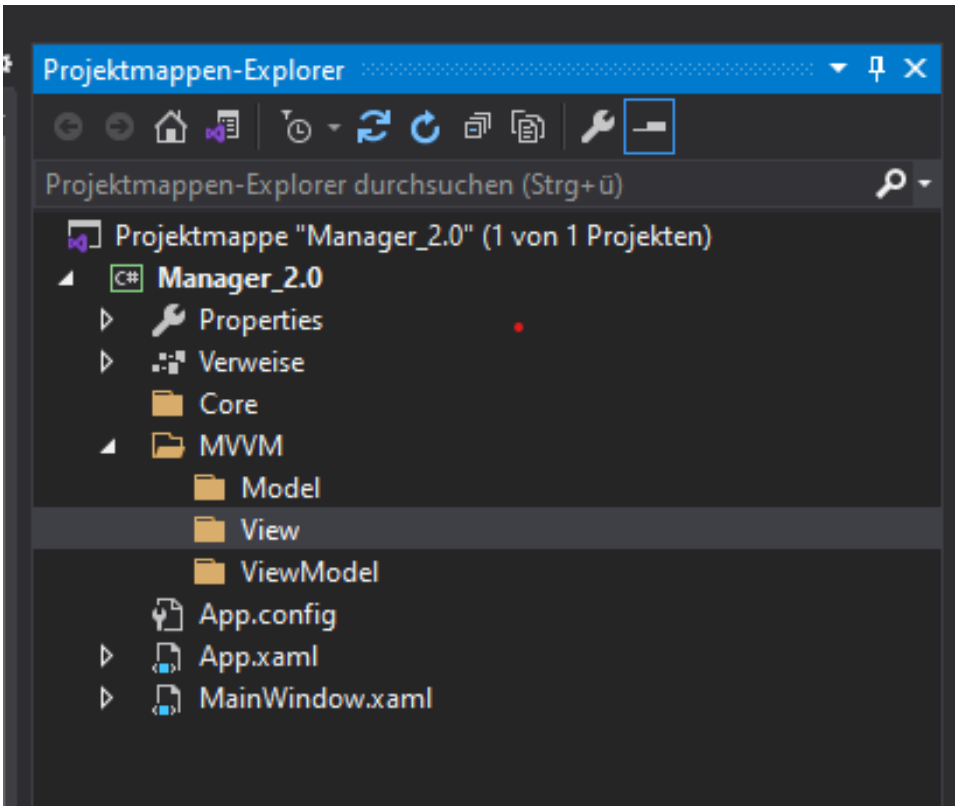
Da dies eine Anwendung die später verschiedene Fenster haben soll und wir nicht immer ein neues Fenster für jede Ansicht öffnen wollen benutzen wir das **MVVM Model**

MVVM bedeutet **M**odel-**V**iew-**V**iew**M**odel eine kurze Erklärung gibt es hier: MVVM Erklärung

Ebenfalls brauchen wir einen Core Ordner der **Commands** und die **INotifyPropertyChanged** Logik enthält (später dazu mehr)

Ordner fügt man durch Rechtsklick auf den Projektnamen hinzufügen hinzu

Am ende sollte es so aussehen



Erstellung der Klassen

Core Ordner

Wie vorher Gesagt Braucht man im Core Ordner eine Klasse Namens **RelayCommand** und eine Klasse namens **ObservableObject**

RelayCommand

```
class RelayCommand : ICommand
    {

        private Action<object> _execute;

        private Predicate<object> _canEcecute;

        //Konstruktor für einen Command
        public RelayCommand(Action<object> execute, Predicate<object> canEcecute = null)
        {
            _execute = execute;
            _canEcecute = canEcecute;
        }


        public event EventHandler CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }

        //Kann ein Command durchgeführt werden
        public bool CanExecute(object parameter)
        {
            return _canEcecute == null || _canEcecute(parameter);
        }

        //Was bei einem Command passieren soll
        public void Execute(object parameter)
        {
            _execute(parameter);
        }
    }
```

ObservableObject

```
class ObservableObject : INotifyPropertyChanged
    {
                //Das Event wird immer dann ausgelöst wenn sich ein Wert ändert
        public event PropertyChangedEventHandler PropertyChanged;

        protected void OnPropertyChanged([CallerMemberName] string name = null)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
        }
    }
```

Model Ordner

Im Model Ordner sind die ganzen Daten für die Anwendung und normalerweise werden hier Datenbanken abgerufen aber das ist Momentan zu fortgeschritten

In diesem Fall Erstellen wir eine **Paletten** Klasse und eine **PalettenManager** Klasse da dies die Daten für die Anwendung sind

Die **Palette** soll folgende Eigenschaften enthalten: Palettennummer, Hallennummer, Zielhalle und Palettengröße (string).

Der **PalettenManager** soll eine ObservableCollection von Paletten erstellen und auch durch eine Methode diese zurückgeben ebenfalls sollen Methoden Paletten zu dieser Collection hinzufügen oder entfernen.

Dies könntest du noch selbst herausfinden wie die Klassen aussehen sollen

Palette

```
class Palette
    {

        public Palette(int palettennummer, int hallennummer, int zielhalle, string palettengröße)
        {
            Palettennummer = palettennummer;
            Hallennummer = hallennummer;
            Zielhalle = zielhalle;
            PalettenGröße = palettengröße;
        }

        public string PalettenGröße { get; set; }
        public int Palettennummer { get; set; }
        public int Hallennummer { get; set; }
        public int Zielhalle { get; set; }
    }
```

PalettenManager

```
class PalettenManager
    {
        public static ObservableCollection<Palette> PalettenListe = new ObservableCollection<Palette>();

        public static ObservableCollection<Palette> GetPalettes()
        {
            return PalettenListe;
        }

        public static void removePalette(Palette palette)
        {
            PalettenListe.Remove(palette);
        }

        public static void addPalette(Palette palette)
        {
            PalettenListe.Add(palette);
        }
    }
```

So Nun haben wir bereits die Grund Logik erstellt. Jetzt geht es darum wir wir die Daten anzeigen wollen

Man kann gefühlt Tage damit verbringen das aussehen von jedem Element der Anwendung mithilfe von Styles zu verändern aber ich beschränke mich jetzt auf das Grundlegendste

Die Views erstellen

Die verschiedenen Ansichten der Anwendung sind sogenannte **Benutzersteuerelemente** die kann man auch mit Rechtsklick auf View Hinzufügen Benutzersteuerelement

Wir brauchen drei Views:

- CreatePalett (Hier erstellt man einen Auftrag/Palette)
- Liefern (Soll den Auftrag durchführen)
- PalettenListe (Soll die Paletten anzeigen)

Die ViewModels erstellen

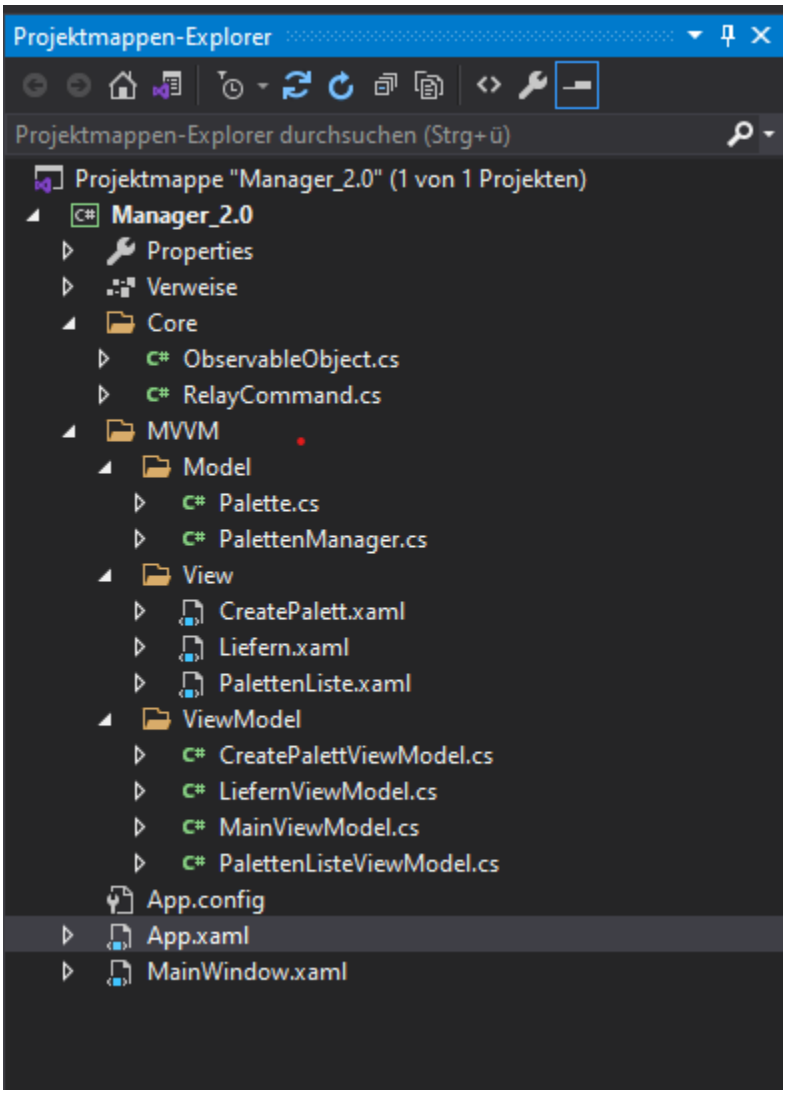
Für jede View brauchen wir auch ein ViewModel das ist nur eine Normale Klasse

Wir erstellen ViewModels da jede View eine andere Logik bzw. Daten benötigt und man somit mehr Übersicht erlangt

Dennoch brauchen wir auch für das MainWindow ein ViewModel weil von dort aus das Hauptfenster ihre Datenbekommt

- CreatePalettViewModel
- LiefernViewModel
- PalettenListeViewModel
- MainViewModel

nach erfolgreichen Erstellen Jeder Datei müsste das Projekt so aussehen:



App.xaml

Nun müssen wir aber noch Veränderungen in der App.xaml Datei vornehmen, damit wir das DataBinding überhaupt anwenden können für die verschiedenen View

App.xaml

```
//Nahc namespace: Muss dein Projektname stehen bei mir (Manager_2._0)
// dieses Zwei Zeilen Müssen noch vor StartupURI eingefügt werden

xmlns:viewModel="clr-namespace:Manager_2._0.MVVM.ViewModel"
xmlns:view="clr-namespace:Manager_2._0.MVVM.View"

 <Application.Resources>
        <DataTemplate DataType="{x:Type viewModel:CreatePalettViewModel}">
            <view:CreatePalett/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type viewModel:LiefernViewModel}">
            <view:Liefern/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type viewModel:PalettenListeViewModel}">
            <view:PalettenListe/>
        </DataTemplate>
    </Application.Resources>
```

Das Design

Hauptfenster

Das Design ist ganz deine Sache. Ich persönlich fand eine **grau/schwarze** Anwendung am besten.

Wir müssen aber zunächst ein paar Änderungen am Fenster vornehmen:

Änderungen am Fenster

```
                Height="550" Width="910"
        WindowStyle="None"
        Background="Transparent"
        AllowsTransparency="True"
        WindowStartupLocation="CenterScreen"
```

Hautpfenster Design

```
<Border Background="#2e3136" CornerRadius="20">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="200"/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="50"/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Border Grid.RowSpan="2" Grid.Column="1" Background="#495057" CornerRadius="0,20,20,0"/>
            <TextBlock Text="ForkLiftManager"
                       HorizontalAlignment="Center"
                       VerticalAlignment="Center"
                       Foreground="White"
                       Margin="10,0,0,0"
                       FontWeight="Bold"
                       FontSize="20"/>
            <StackPanel Grid.Row="1">

                <RadioButton Height="50"
                             Content="Auftrag aufgeben"
                             IsChecked="True"/>
                <RadioButton Height="50"
                             Content="Paletten"/>
                <RadioButton Height="50"
                             Content="Lieferung"/>
            </StackPanel>
                        <ContentControl Margin="10"  Grid.Column="1" Grid.Row="1" />
        </Grid>
    </Border>
```

Das Fenster sollte nun schon ganz anders ausschauen aber du wirst bemerken, dass die **Radiobuttons** nicht besonders schön ausschauen um das zu beheben muss man sogenannte Styles in die App.xaml Datei einfügen dadurch wird der Style automatisch für jeden Radiobutton geändert

man muss einfach nach dem **letzen** DataTemplate folgendes einfügen

Radiobutton Style

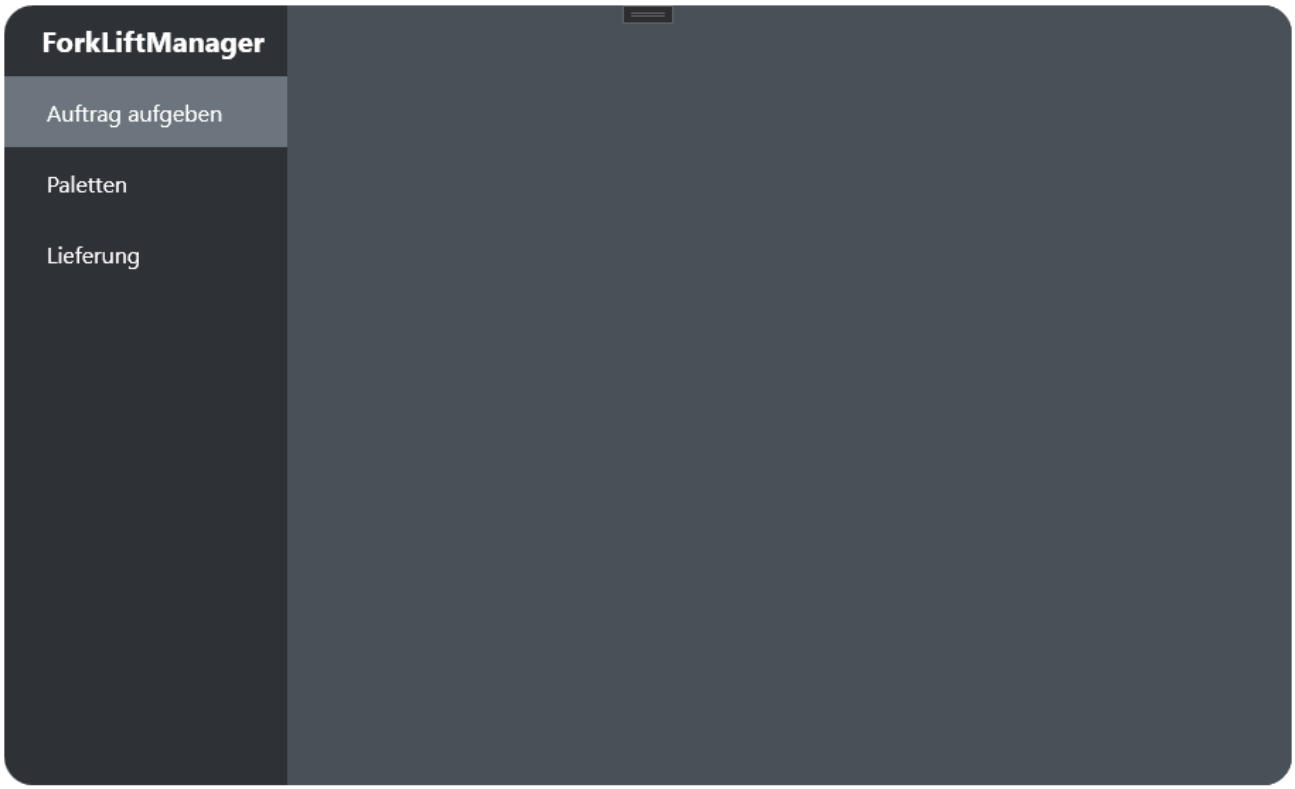
```
<Style BasedOn="{StaticResource {x:Type ToggleButton}}" TargetType="{x:Type RadioButton}">
            <Style.Setters>
                <Setter Property="Template">
                    <Setter.Value>
                        <ControlTemplate TargetType="RadioButton">
                            <Grid VerticalAlignment="Stretch"
                              HorizontalAlignment="Stretch"
                              Background="{TemplateBinding Background}">

                                <TextBlock VerticalAlignment="Center"
                                       Margin="30,0,0,0"
                                       FontSize="16"
                                       Text="{TemplateBinding Property=Content}"/>
                            </Grid>
                        </ControlTemplate>
                    </Setter.Value>
                </Setter>
                <Setter Property="Background" Value="Transparent"/>
                <Setter Property="BorderThickness" Value="0"/>
                <Setter Property="Foreground" Value="White"/>
            </Style.Setters>

            <Style.Triggers>
                <Trigger Property="IsChecked" Value="True">
                    <Setter Property="Background" Value="#6c757d"/>
                </Trigger>
            </Style.Triggers>

        </Style>
```

Nun müsste das Fenster so ausschauen:



Wenn du nun auf die Buttons klickst siehst du, dass sie ihre Farbe ändern.

Du kannst die Farben vom Fenster durch die Background Eigenschaft ändern und die Farbe vom Button im Style beim Trigger "isChecked".

Palette hinzufügen

Im CreatePalett Fenster brauchen wir folgende Elemente

- 4 Textboxen für die 4 Eigenschaften
- 4 Überschriften über den Textboxen
- Einen Button

Für die Anordnung kann man ein **Stackpanel** benutzen mit der **Orientation ="Vertical".** Die **Überschrift** kann man mit einem Label machen wobei man die Foreground farbe noch ändern sollte.

Bei der Textbox muss man nur die Width und Height Eigenschaft ändern genau so wie beim Button.

Wenn du willst kannst du auch ein Grid erstellen mit zwei Spalten in den jeweils zwei Eigenschaften kommen.

Ich habe es so gestaltet:

```
        <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <StackPanel Orientation="Vertical">
            <Label Content="Palettennummer" Foreground="White" HorizontalAlignment="Center"/>
            <TextBox Background="White" Width="180" Height="30"  />
            <Label Content="Hallennummer" Foreground="White" HorizontalAlignment="Center"/>
            <TextBox Background="White" Width="180" Height="30"  />
        </StackPanel>
        <StackPanel Grid.Column="1">
            <Label Content="Zielhalle" Foreground="White" HorizontalAlignment="Center"/>
            <TextBox Background="White" Width="180" Height="30"  />
            <Label Content="Palettengröße" Foreground="White" HorizontalAlignment="Center"/>
            <TextBox Background="White" Width="180" Height="30"  />
        </StackPanel>
        <Button Content="Palette Hinzufügen" Width="150" Height="30" Grid.ColumnSpan="2"/>
    </Grid
```

Paletten anzeigen

Hier brauchen wir nur eine ListView mit 4 Spalten dies schaut so aus:

PalettenList

```
        <Grid>
        <ListView>
            <ListView.View>
                <GridView>
                    <GridViewColumn Width="Auto" Header="Palettennummer" />
                    <GridViewColumn Width="Auto" Header="Hallennummer" />
                    <GridViewColumn Width="Auto" Header="Zielhallennummer" />
                    <GridViewColumn Width="Auto" Header="Palettengröße"/>
                </GridView>
            </ListView.View>
        </ListView>
    </Grid>
```

Lieferung durchführen

Hier brauchen wir nur eine ComboBox und einen Button

Lieferung

```
<Grid>
        <ComboBox Margin="50" Width="220" Height="30">

        </ComboBox>

        <Button Width="150" Height="30" Cursor="Hand" Margin="319,296,331,124"/>

    </Grid>
```

Die Logik

MainWindow

Das MainWindowViewModel muss nun folgende Elemente besitzen:

- **Ein** object namens CurrentView das in seiner Set Mehode die **OnPropertyChanded** Methode in der **ObservableObject** Klasse auslöst
- **Drei** Commands für **jedes Fenster** bzw. Button
- Für jedes **ViewModel** eine öffentliche **Eigenschaft** die auf das **jeweilige ViewModel** zugreifen kann und sowhol Lesen als auch schreiben erlaubt
- Einen **Konstruktor** für die MainViewModel Klasse in der die Commands die CurrentView auf das jeweilige ViewModel der Fenster ändern

Am Ende sollte es so ausschauen:

MainViewModel

```
class MainViewModel :ObservableObject
    {
        private object _currentView;
        public object CurrentView
        {
            get { return _currentView; }
            set
            {
                _currentView = value;
                OnPropertyChanged();
            }
        }

        //Für jedes Fenster bzw. Button wird ein Command erstellt
        public RelayCommand CreatePalettCommand { get; set; }
        public RelayCommand PalettenListeCommand { get; set; }
        public RelayCommand LiefernCommand { get; set; }



        public CreatePalettViewModel CreateVM { get; set; }
        public LiefernViewModel LiefernVM { get; set; }
        public PalettenListeViewModel ListeVM { get; set; }



        public MainViewModel()
        {
            //Durch die Commands wird immer die CurrentView geändert und somit auch das angezeigte Fenster

            CreateVM = new CreatePalettViewModel();
            CreatePalettCommand =  new RelayCommand(o => { CurrentView = CreateVM; });

            LiefernVM = new LiefernViewModel();
            LiefernCommand = new RelayCommand(o => { CurrentView = LiefernVM; });

            ListeVM = new PalettenListeViewModel();
            PalettenListeCommand = new RelayCommand(o => { CurrentView = ListeVM; });

                        CurrentView = CreateVM;
        }
    }
```

Nun haben wir aber noch nicht die Daten an das MainWindow gebunden dazu muss vor der ersten Border folgendes eingefügt werden:

Main Window Data Binding

```
Ebenfalls muss in der Zeile mit xmlns:local der DateiPfad geändert werden: zu MVVM.ViewModel
Beachte auch hier das anstatt (Manager_2._0) dein Projektname steht

xmlns:local="clr-namespace:Manager_2._0.MVVM.ViewModel"

<!--Hier werden die Daten des ViewModels in die View gebunden-->
<Window.DataContext>
        <local:MainViewModel/>
    </Window.DataContext>
```

Nun können wir mit dem DataBinding für die Radiobuttons und ContentControl beginnen

Das DataBinding ist ganz einfach man schreibt bei den **Radiobuttons** in die Eigenschaften einfach: **Command="{Binding CommandName}"**

Bei der ContentControl muss man in die Eigenschaften folgendes einfügen: **Content="{Binding CurrentView}"**

Wenn alles eingetragen ist sollte das MainWindow.xaml Datei so ausschauen:

MainWindow.xaml

```
<Window x:Class="Manager_2._0.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:Manager_2._0.MVVM.ViewModel"
        mc:Ignorable="d"
        Title="MainWindow" Height="550" Width="910"
        WindowStyle="None"
        Background="Transparent"
        AllowsTransparency="True"
        WindowStartupLocation="CenterScreen">
    <!--Hier werden die Daten des ViewModels in die View gebunden-->
    <Window.DataContext>
        <local:MainViewModel/>
    </Window.DataContext>
    <Border Background="#2e3136" CornerRadius="20">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="200"/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="50"/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Border Grid.RowSpan="2" Grid.Column="1" Background="#495057" CornerRadius="0,20,20,0" />
            <TextBlock Text="ForkLiftManager"
                       HorizontalAlignment="Center"
                       VerticalAlignment="Center"
                       Foreground="White"
                       Margin="10,0,0,0"
                       FontWeight="Bold"
                       FontSize="20"/>
            <StackPanel Grid.Row="1">

                <RadioButton Height="50"
                             Content="Auftrag aufgeben"
                             IsChecked="True"
                             Command="{Binding CreatePalettCommand}"/>
                <RadioButton Height="50"
                             Content="Paletten"
                             Command="{Binding PalettenListeCommand}"/>
                <RadioButton Height="50"
                             Content="Lieferung"
                             Command="{Binding LiefernCommand}"/>
            </StackPanel>
            <ContentControl Margin="10"  Grid.Column="1" Grid.Row="1" Content="{Binding CurrentView}" />
        </Grid>
    </Border>
</Window>
```

Paletten erstellen

Wir haben ja beim Design bereits vier TextBoxen erstellt dafür brauchen wir jetzt im ViewModel auch **4 Variablen** mit einem **Getter** und **Setter** um die TextBoxen mit diesen Variablen zu binden.

wir benötigen ebenfalls **einen Command** den wir dem Button binden

Ebenfalls benötigen wir für den Command **eine Methode** die eine **Palette** mit den eingegebenen Eigenschaften erstellt und diese der Paletten liste im PalettenManager hinzufügt

Die CreatePalletViewModel Klasse sollte dann so ausschauen:

CreatePalettViewModel

```
class CreatePalettViewModel
    {

        public string Größe { get; set; }
        public int Halle { get; set; }
        public int Zielhalle { get; set; }
        public int Nummer { get; set; }

        public RelayCommand addPallet { get; set; }

        public CreatePalettViewModel()
        {
            addPallet = new RelayCommand(PaletteHinzufügen);
        }

        private void PaletteHinzufügen(object obj)
        {
            Palette palette = new Palette(Nummer, Halle, Zielhalle, Größe);
            PalettenManager.addPalette(palette);
        }
    }
```

Nun müssen wir wieder den DataContext in der **CreatePalett.xaml** Datei hinzufügen und den Text der TextBoxen auf die Variablen Binden mit **Text="{Binding Größe oder Halle oder Zielhalle oder Nummer}"** den addPalett Command muss mach auch noch auf den Button bidnen.

CreatePalett.xaml

```
        <UserControl.DataContext>
        <local:CreatePalettViewModel/>
    </UserControl.DataContext>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <StackPanel Orientation="Vertical">
            <Label Content="Palettennummer" Foreground="White" HorizontalAlignment="Center"/>
            <TextBox Background="White" Width="180" Height="30" Text="{Binding Nummer}"/>
            <Label Content="Hallennummer" Foreground="White" HorizontalAlignment="Center"/>
            <TextBox Background="White" Width="180" Height="30" Text="{Binding Halle}" />
        </StackPanel>
        <StackPanel Grid.Column="1">
            <Label Content="Zielhalle" Foreground="White" HorizontalAlignment="Center"/>
            <TextBox Background="White" Width="180" Height="30"  Text="{Binding Zielhalle}" />
            <Label Content="Palettengröße" Foreground="White" HorizontalAlignment="Center"/>
            <TextBox Background="White" Width="180" Height="30" Text="{Binding Größe}" />
        </StackPanel>
        <Button Content="Palette Hinzufügen" Width="150" Height="30" Grid.ColumnSpan="2" Command="{Binding
addPallet}" Cursor="Hand"/>
    </Grid>
```

Paletten anzeigen

Nun können wir bereits Paletten erstellen jetzt müssen wir sie noch anzeigen lassen. Das geht mithilfe einer **ListView**.

in die **PalettenListeViewModel** Klasse müssen wir nun eine **ObservableCollection** mit einem **Getter** und **Setter** erstellen und im Konstruktor dieser Collection die in **PalettenManager** gespeichereten PalettenListe zuweisen

PalettenListViewModel

```
class PalettenListeViewModel
    {
        public ObservableCollection<Palette> Liste { get; set; }
        public PalettenListeViewModel()
        {
            Liste = PalettenManager.GetPalettes();
        }
    }
```

Nun müssen wir in der View wieder einen **DataContext** erstellen und der Listview als **ItemSourece** die erstellte Liste geben

PalettenListe.xaml

```
        <UserControl.DataContext>
        <local:PalettenListeViewModel/>
    </UserControl.DataContext>
    <Grid>
        <ListView ItemsSource="{Binding Liste}">
            <ListView.View>
                <GridView>
                    <GridViewColumn Width="Auto" Header="Palettennummer" DisplayMemberBinding="{Binding
Palettennummer}"/>
                    <GridViewColumn Width="Auto" Header="Hallennummer" DisplayMemberBinding="{Binding
Hallennummer}"/>
                    <GridViewColumn Width="Auto" Header="Zielhallennummer" DisplayMemberBinding="{Binding
Zielhalle}"/>
                    <GridViewColumn Width="Auto" Header="Palettengröße" DisplayMemberBinding="{Binding
PalettenGröße}"/>
                </GridView>
            </ListView.View>
        </ListView>
    </Grid>
```

Nun kann man sehen welche Paletten man hinzugefügt hat

Lieferung durchführen

Es gibt verschiedene Arten wie man eine Palette zum durchführen auswählen kann. Mann kann zum Beispiel ein Textfeld erstellen und dort die Palettennummer eingeben oder man kann direkt bei der Palettenanzeige auf eine Palette doppelklicken. Ich habe es mit einem DropDown Menu gelöst, dass alle Paletten anzeigt.

Dazu muss in das ViewModel für die Liefern Seite ein Command, nochmal eine locale Palettenliste wie in der Ansicht und Palette mit Gettern und Settern

So sollte es dann aussehen:

LiefernViewModel

```
class LiefernViewModel:ObservableObject
    {
        public ObservableCollection<Palette> Liste { get; set; }
        public Palette SelectedPalette { get; set; }

        public RelayCommand Durchführen { get; set; }

        public LiefernViewModel()
        {
            Liste = PalettenManager.GetPalettes();
            Durchführen = new RelayCommand(ChangeHalle);
        }

        private void ChangeHalle(object obj)
        {
            if(SelectedPalette!= null)
            {
                SelectedPalette.Hallennummer = SelectedPalette.Zielhalle;
                SelectedPalette.Zielhalle = 0;
            }
        }
    }
```

Für die **Liefern.xmal** Datei kann man ein Dropdown Menu mithilfe einer **Combobox** erstellen wobei die **ItemSource** auf die Liste gebunden wird.

Liefern.xaml

```
        <UserControl.DataContext>
        <local:LiefernViewModel/>
    </UserControl.DataContext>
        <Grid>
        <ComboBox Margin="50" Width="220" Height="30" x:Name="PalettenAuswahl" ItemsSource="{Binding Liste}"
SelectedItem="{Binding SelectedPalette}">
            <ComboBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal">
                        <TextBlock Text="Nummer: "/>
                        <TextBlock Text="{Binding Palettennummer}" Foreground="Red" Margin="0,0,5,0"/>
                        <TextBlock Text="Von Halle: "/>
                        <TextBlock Text="{Binding Hallennummer}" Foreground="Red" Margin="0,0,5,0"/>
                        <TextBlock Text="In: "/>
                        <TextBlock Text="{Binding Zielhalle}" Foreground="Red" Margin="0,0,5,0"/>
                        <TextBlock Text="Größe: "/>
                        <TextBlock Text="{Binding PalettenGröße}" Foreground="Red"/>
                    </StackPanel>
                </DataTemplate>
            </ComboBox.ItemTemplate>
        </ComboBox>

        <Button Width="150" Height="30" Cursor="Hand" Command="{Binding Durchführen}" Margin="
319,296,331,124"/>

    </Grid>
```

Jetzt müsste alles Funktionieren und wenn man den Button betätigt kann man in der Palette Ansicht sehen das sich die Zielhalle auf 0 ändert und die Hallennummer verändert wurde.

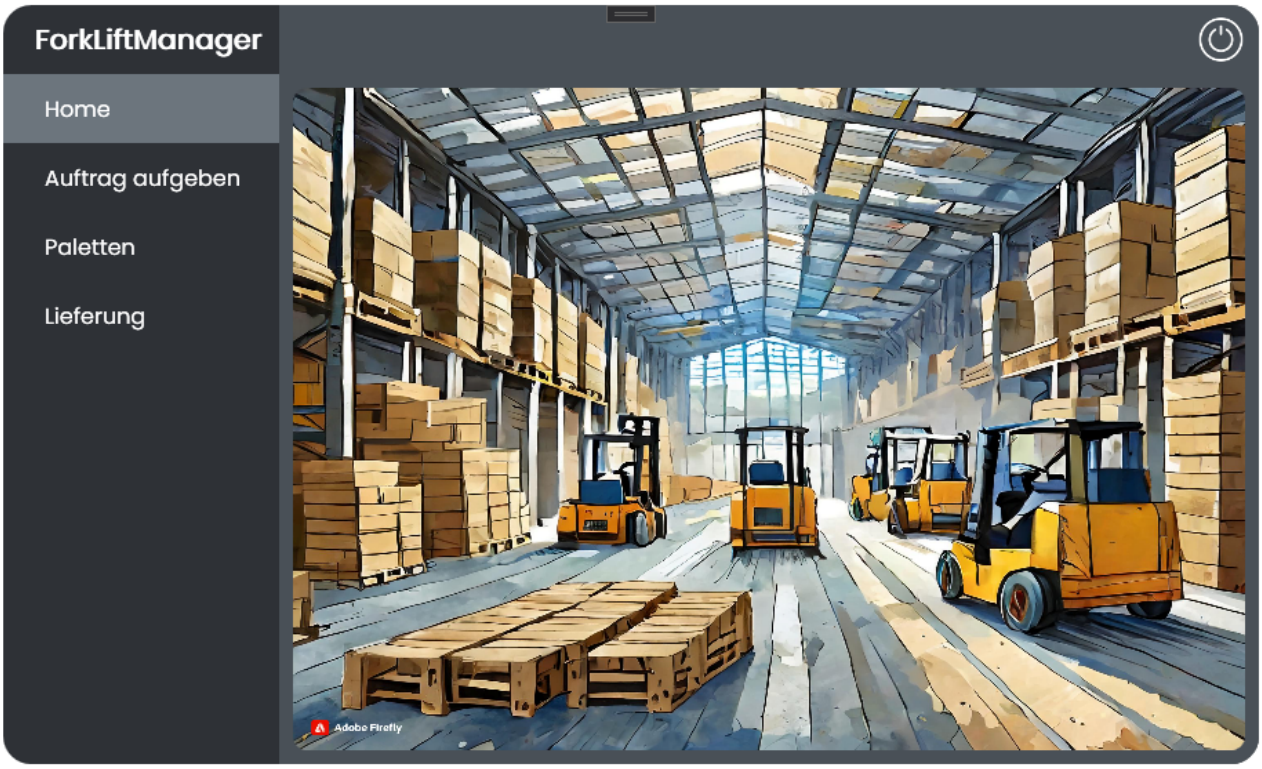
Verbesserungen

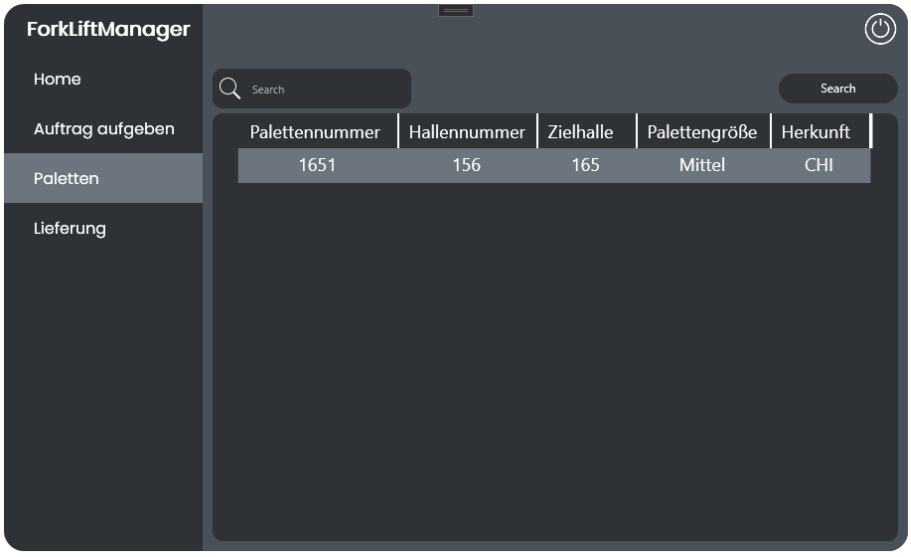
Es gibt natürlich viele Möglichkeiten das Programm zu verbessern/erweitern. Man kann z. B.

- Palettengröße als Enum speichern
- Noch mehr Eigenschaften zu einer Palette hinzufügen
- Bei Doppelklick auf die Liste könnte man ein Fenster öffnen lassen mit den ganzen Paletten Eigenschaften
- Man kann auch eine Liste an Gabelstaplern erstellen und immer wenn eine Palette verändert wird, wird ein Stapler gelöscht

Mit **Styles** kann man natürlich **jedes Element** der Anwendung verändern.

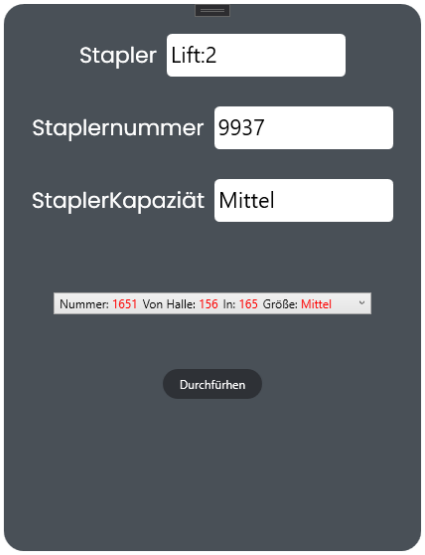
Hier ist ein Beispiel wie das Programm auch aussehen könnte:






ForkLiftManager
Home
Auftrag aufgeben
Paletten
Lieferung
Palettennummer
Zielhalle
Hallennummer
Palettengröße
Herkunft
Beschreibung
add Palette


ForkLiftManager
Home
Auftrag aufgeben
Paletten
Lieferung
Gabelstapler
Name
Nummer
Kapazität
Fortschritt
Lift:1 6298 Klein
Lift:2 9937 Mittel
Lift:3 2362 Groß
Lift:4 5020 Mittel
Lift:5 3912 Klein
Lift:6 6684 Groß
Lift:7 4602 Klein
Lift:8 4235 Mittel
Lift:9 3596 Groß


Stapler
Lift:2
Staplernummer
9937
StaplerKapaziät
Mittel
Nummer: 1651 Von Halle: 156 In: 165 Größe: Mittel
Durchführen