Laboratorium nr 3

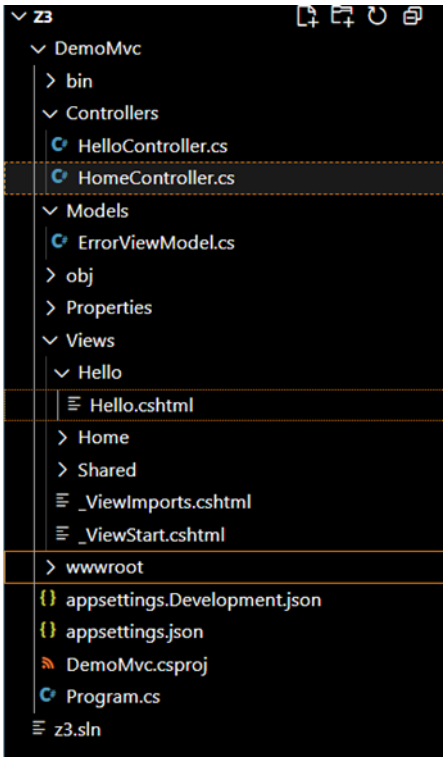
Temat: Tworzenie aplikacji typu MVC i MVVM

1 Zadanie 1- projekt MVC

1.1 Utworzenie projektu MVC

```
(base) PS C:\Users\pawel\Desktop\Desktop\z3> dotnet new mvc
    -n DemoMvc
(base) PS C:\Users\pawel\Desktop\Desktop\z3> cd ./DemoMvc
(base) PS C:\Users\pawel\Desktop\Desktop\z3\DemoMvc> dotnet
    run
```

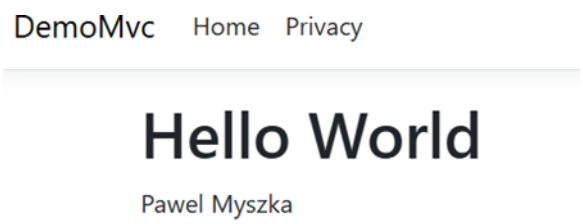
1.2 struktura katalogów:



Rysunek 1: Struktura katalogów

- Controllers – tutaj znajdują się klasy kontrolerów obsługujące żądania HTTP.
- Views – tutaj znajdują się widoki .cshtml, które renderują HTML w przeglądarce.
- Models – tutaj można przechowywać klasy reprezentujące dane aplikacji.

1.3 Po dodaniu kontrolera I prostej akcji index i wpisaniu w wyszukiwarkę: http://localhost:5268/Hello:



Rysunek 2: Widok na adres w przegląarce

1.4 Przepływ żądania w MVC

Po wpisaniu adresu w przeglądarce żądanie HTTP trafia najpierw do mechanizmu routingu, który określa, który kontroler i która akcja powinna je obsłużyć. Następnie kontroler wykonuje odpowiednią logikę i przekazuje dane do widoku, który generuje HTML wyświetlany w przeglądarce.

2 Zadanie 2 – Aplikacja UI bez MVVM (WPF)

2.1 MAUI

MAUI (Multi-platform App UI) to framework firmy Microsoft umożliwiający tworzenie aplikacji działających jednocześnie na Windows, macOS, iOS i Android z jednego wspólnego kodu. Umożliwia tworzenie interfejsu użytkownika (UI) oraz logiki aplikacji w C#, bez konieczności pisania osobnych projektów dla każdej platformy. MAUI pozwala więc na łatwiejsze utrzymanie i rozwój aplikacji wieloplatformowych w porównaniu do tradycyjnych projektów WPF.

2.2 Opis WPF

WPF (Windows Presentation Foundation) to framework firmy Microsoft służący do tworzenia aplikacji desktopowych na system Windows. Pozwala na projektowanie interfejsu użytkownika (UI) przy użyciu języka XAML oraz programowanie logiki aplikacji w C#. WPF jest odpowiedni do tworzenia klasycznych aplikacji desktopowych z rozbudowanym interfejsem graficznym, które nie wymagają wieloplatformowości.

2.3 Utworzenie projektu WPF:

```
(base) PS C:\Users\pawel\Desktop\Desktop\z3> dotnet new wpf
    -n DemoWpf
(base) PS C:\Users\pawel\Desktop\Desktop\z3> cd DemoWpf
(base) PS C:\Users\pawel\Desktop\Desktop\z3\DemoWpf> dotnet
    run
```

2.4 Dodanie elementów UI

W głównym oknie (‘MainWindow.xaml‘) dodano:

- Pole tekstowe **Imię** (TextBox)
- Pole tekstowe **Nazwisko** (TextBox)
- Etykietę (Label/TextBlock) do wyświetlania wyniku
- Przycisk **Połącz** (Button)

Przykładowy kod:

```
private void TxtFirstName_TextChanged(object sender, System.
    Windows.Controls.TextChangedEventArgs e)
{
        firstName = TxtFirstName.Text;
}

private void TxtLastName_TextChanged(object sender, System.
    Windows.Controls.TextChangedEventArgs e)
{
        lastName = TxtLastName.Text;
}
```

2.5 Obsługa zdarzeń w code-behind

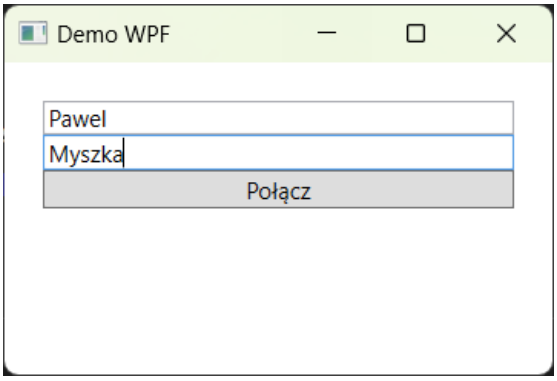
W pliku ‘MainWindow.xaml.cs‘ zaimplementowano:

- Zdarzenia **TextChanged** dla pól tekstowych – zapis wartości do zmiennych.
- Zdarzenie **Click** przycisku – ustawienie w etykiecie połączonego imienia i nazwiska.

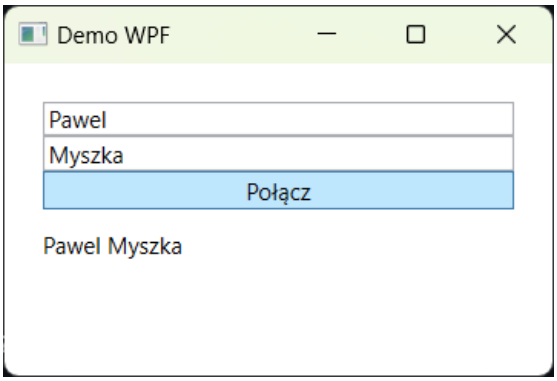
Przykładowy kod:

```
private void BtnCombine_Click(object sender, RoutedEventArgs
    e)
{
        LblFullName.Text = $"{firstName} {lastName}";
}
```

2.6 Przykladowe uruchomienie aplikacji



Rysunek 3: Ekran przed kliknięciem przycisku



Rysunek 4: Ekran po kliknięciu przycisku

3 Zadanie 3 – Refaktoryzacja do MVVM (WPF)

3.1 Dodanie pakietu CommunityToolkit.Mvvm

```
(base) PS D:\stud\sem 5\Desktop\z3\DemoWpf> dotnet add
    package CommunityToolkit.Mvvm
```

3.2 Utworzenie klasy MainViewModel

Utworzono klasę MainViewModel dziedziczącą po ObservableObject:

```
using CommunityToolkit.Mvvm.ComponentModel;
using CommunityToolkit.Mvvm.Input;
```

```
using System.Threading.Tasks;

public partial class MainViewModel : ObservableObject
{
        [ObservableProperty]
        private string firstName;

[ObservableProperty]
\begin{figure}
        \centering
        \includegraphics[width=0.7\linewidth]{C:/Users/pawel
            /Downloads/po_zmianie}
        \caption{}
        \label{fig:pozmianie}
\end{figure}
\begin{figure}
        \centering
        \includegraphics[width=0.7\linewidth]{C:/Users/pawel
            /Downloads/osobno}
        \caption{}
        \label{fig:osobno}
\end{figure}
\begin{figure}
        \centering
        \includegraphics[width=0.7\linewidth]{C:/Users/pawel
            /Downloads/nowy_przycisk}
        \caption{}
        \label{fig:nowyprzycisk}
\end{figure}
\begin{figure}
        \centering
        \includegraphics[width=0.7\linewidth]{C:/Users/pawel
            /Downloads/niedziala}
        \caption{}
        \label{fig:niedziala}
\end{figure}
\begin{figure}
        \centering
        \includegraphics[width=0.7\linewidth]{C:/Users/pawel
            /Downloads/Hello}
        \caption{}
        \label{fig:hello}
\end{figure}
\begin{figure}
        \centering
        \includegraphics[width=0.7\linewidth]{C:/Users/pawel
            /Downloads/folder}
        \caption{}
        \label{fig:folder}
\end{figure}
```

```
\begin{figure}
        \centering
        \includegraphics[width=0.7\linewidth]{C:/Users/pawel
            /Downloads/dziala}
        \caption{}
        \label{fig:dziala}
\end{figure}
\begin{figure}
        \centering
        \includegraphics[width=0.7\linewidth]{C:/Users/pawel
            /Downloads/details}
        \caption{}
        \label{fig:details}
\end{figure}
\begin{figure}
        \centering
        \includegraphics[width=0.7\linewidth]{C:/Users/pawel
            /Downloads/valdiation}
        \caption{}
        \label{fig:valdiation}
\end{figure}
\begin{figure}
        \centering
        \includegraphics[width=0.7\linewidth]{C:/Users/pawel
            /Downloads/razem}
        \caption{}
        \label{fig:razem}
\end{figure}
private string lastName;

public string FullName => $"{FirstName} {LastName}";

[RelayCommand]
        public async Task SaveAsync()
                {
                        await Task.Delay(500);
                        OnPropertyChanged(nameof(FullName));
                }
}
```

3.3 Powiązanie ViewModel z UI

W konstruktorze MainWindow przypisano ViewModel do DataContext:

```
using System.Windows;

namespace DemoWpf
{
        public partial class MainWindow : Window
```

```
        {
                public MainWindow()
                        {
                                InitializeComponent();
                                DataContext = new
                                    MainViewModel();
                        }
        }
}
```

3.4 Zmiany w XAML

Kontrolki zostały powiązane z właściwościami ViewModel za pomocą bindingów:

```
<StackPanel Margin="20">
<TextBox Text="{Binding FirstName, Mode=TwoWay}" />
<TextBox Text="{Binding LastName, Mode=TwoWay}" />
<Button Content="Zapisz" Command="{Binding SaveCommand}" />
<Label Content="{Binding FullName}" />
</StackPanel>
```

3.5 Opis działania

- Wpisanie imienia i nazwiska w TextBox aktualizuje odpowiednie właściwości w ViewModel.
- Właściwość `FullName` wylicza połączone imię i nazwisko.
- Kliknięcie przycisku `Zapisz` wywołuje metodę `SaveAsync()` i powiadamia UI o zmianie `FullName`.
- Dzięki bindingom UI jest całkowicie oddzielone od logiki aplikacji.

4 Zadanie 4 – Dodanie IsNotBusy / CanExecute

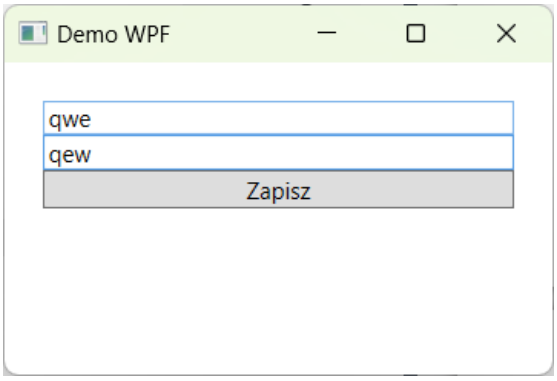
4.1 Zmiany w MainViewModel

W ViewModel dodano nowe właściwości i zmodyfikowano komendę:

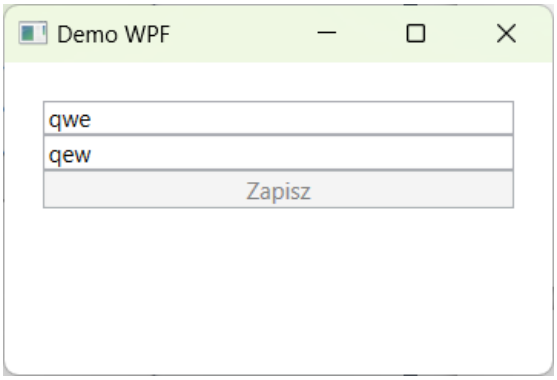
```
[ObservableProperty]
private bool isBusy = false;

public bool IsNotBusy => !IsBusy;

[RelayCommand(CanExecute = nameof(IsNotBusy))]
        public async Task SaveAsync()
        {
```

```
                IsBusy = true;
                await Task.Delay(3000);
                IsBusy = false;
                OnPropertyChanged(nameof(FullName));
        }
```

4.2 Przykładowe uruchomienie



Rysunek 5: Przycisk aktywny



Rysunek 6: Przycisk nieaktywny (IsBusy = true)

5 Zadanie 5 – Walidacja pola Imię

5.1 Zmiany w MainViewModel

Dodano właściwość ValidationMessage oraz zmodyfikowano metodę SaveAsync():

```
[ObservableProperty]
private string validationMessage = "";

[RelayCommand(CanExecute = nameof(IsNotBusy))]
        public async Task SaveAsync()
        {
        if (string.IsNullOrWhiteSpace(FirstName))
        {
                ValidationMessage = "Imi Ź jest wymagane";
                return;
        }

        ValidationMessage = "";
        IsBusy = true;
        await Task.Delay(500);
        IsBusy = false;
        OnPropertyChanged(nameof(FullName));
        }
```

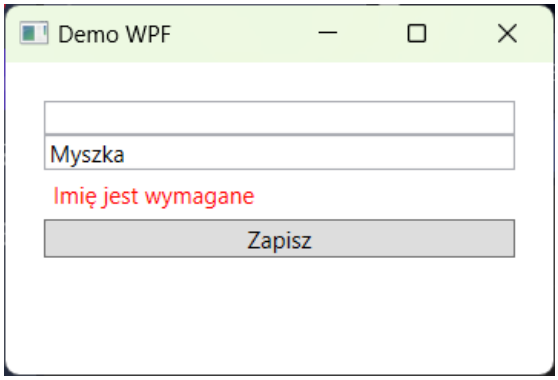
5.2 Zmiany w XAML

Dodano Label powiązany z właściwością `ValidationMessage`:

```
<Label Content="{Binding ValidationMessage}" Foreground="Red
    " />
```

Label automatycznie wyświetla komunikat walidacyjny w momencie, gdy pole FirstName jest puste.

5.3 Przykładowe uruchomienie



Rysunek 7: Pole Imię puste – komunikat walidacyjny wyświetlony

6 Zadanie 6 – ValueConverter

6.1 Konwerter BoolInverseConverter

Utworzono konwerter odwracający wartość logiczną:

```
using System;
using System.Globalization;
using System.Windows.Data;

namespace DemoWpf
{
        public class BoolInverseConverter : IValueConverter
        {
                public object Convert(object value, Type
                    targetType, object parameter, CultureInfo
                     culture)
                => value is bool b ? !b : false;
        }
}
```

6.2 Użycie konwertera w XAML

```
<Window.Resources>
<local:BoolInverseConverter x:Key="BoolInverseConverter"/>
</Window.Resources>

<Button Content="Zapisz"
Command="{Binding SaveCommand}"
IsEnabled="{Binding IsBusy, Converter={StaticResource
    BoolInverseConverter}}"/>
```

Przycisk Zapisz wykorzystuje konwerter do dynamicznego ustawienia IsEnabled.

6.3 Cel zastosowania konwertera

Konwerter BoolInverseConverter odwraca wartość `IsBusy`, aby przycisk „Zapisz” był aktywny tylko wtedy, gdy aplikacja nie wykonuje operacji. Dzięki temu logika pozostaje w ViewModel, a UI automatycznie reaguje, bez ręcznej obsługi w code-behind.

7 Zadanie 7 – Dependency Injection

7.1 Instalacja pakietu

```
(base) PS D:\stud\sem 5\Desktop\z3\DemoWpf> dotnet add
    package Microsoft.Extensions.DependencyInjection
```

7.2 Konfiguracja w WPF

W pliku App.xaml.cs skonfigurowano kontener DI:

```
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Windows;

namespace DemoWpf
{
        public partial class App : Application
        {
                public new static App Current => (App)
                    Application.Current;
                public IServiceProvider Services { get; }

                public App()
                {
                        var serviceCollection = new
                            ServiceCollection();
                        ConfigureServices(serviceCollection)
                            ;
                        Services = serviceCollection.
                            BuildServiceProvider();
                }

                private void ConfigureServices(
                    IServiceCollection services)
                {
                        services.AddSingleton<MainViewModel
                            >();
                        services.AddSingleton<MainWindow>();
                }

                protected override void OnStartup(
                    StartupEventArgs e)
                {
                        var mainWindow = Services.
                            GetRequiredService<MainWindow>();
                        mainWindow.Show();
                        base.OnStartup(e);
                }
        }
}
```

7.3 Dodanie ViewModel do okna

W MainWindow.xaml.cs dodano ViewModel:

```
public partial class MainWindow : Window
{
        public MainWindow(MainViewModel vm)
        {
                InitializeComponent();
                DataContext = vm;
        }
}
```

7.4 Wnioski

Dependency Injection (DI) pozwala wstrzykiwać zależności, takie jak ViewModel, bez tworzenia ich ręcznie w kodzie okna. Dzięki temu kod jest bardziej modularny, testowalny i zgodny z wzorcem MVVM.

8 Zadanie 8 – Testy jednostkowe ViewModelu

8.1 Utworzenie projektu testowego

Projekt testowy NUnit został utworzony w katalogu DemoTests z targetem net9.0:

```
(base) PS D:\stud\sem 5\Desktop\z3> dotnet new nunit -n
    DemoTests -f net9.0
(base) PS D:\stud\sem 5\Desktop\z3> cd DemoTests
(base) PS D:\stud\sem 5\Desktop\z3\DemoTests> dotnet restore

(base) PS D:\stud\sem 5\Desktop\z3> dotnet test
```

8.2 Przykładowe testy

W pliku MainViewModelTests.cs dodano testy sprawdzające:

- Poprawne generowanie pełnego imienia `FullName` po zmianie pól `FirstName` i `LastName`.
- Walidację pola `FirstName` – gdy pozostaje puste, `ValidationMessage` przyjmuje odpowiednią wartość.

```
using NUnit.Framework;
using DemoWpf;

namespace DemoTests
```

```
{
        public class MainViewModelTests
        {
                [Test]
                public void
                    FullName_Updates_OnFirstOrLastChange()
                {
                        var vm = new MainViewModel();
                        vm.FirstName = "Anna";
                        vm.LastName = "Nowak";
                        Assert.That(vm.FullName, Is.EqualTo
                            ("Anna Nowak"));
                }

                [Test]
                public void
                    ValidationMessage_Shows_WhenFirstNameEmpty
                    ()
                {
                        var vm = new MainViewModel();
                        vm.FirstName = "";

                        vm.SaveCommand.Execute(null);

                        Assert.That(vm.ValidationMessage, Is
                            .EqualTo("Imi Ź jest wymagane"));
                }
        }
}
```

Wszystkie testy zakończyły się powodzeniem, co potwierdza poprawne działanie logiki ViewModelu niezależnie od interfejsu użytkownika.

9 Zadanie 9 – Details Page

9.1 Dodanie nowego okna DetailsPage

Utworzono nowe okno DetailsPage.xaml w projekcie WPF, które służy do wyświetlania szczegółów.

9.2 Obsługa przycisku „Szczegóły”

W MainWindow.xaml dodano przycisk:

```
<Button Content="Szczeg Ćy" Click="BtnDetails_Click"/>
```

W MainWindow.xaml.cs zaimplementowano obsługę kliknięcia:

```
private void BtnDetails_Click(object sender, RoutedEventArgs
    e)
```

```
{
        var detailsWindow = new DetailsPage((MainViewModel)
            DataContext);
        detailsWindow.Show();
                }
```

Kliknięcie przycisku otwiera nowe okno DetailsPage, przekazując obecny MainViewModel jako DataContext.

9.3 Wyświetlanie danych w DetailsPage

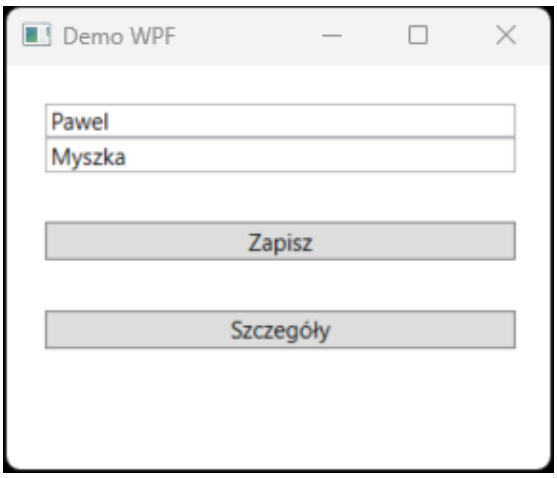
W DetailsPage.xaml ustawiono bindingi do właściwości ViewModelu:

```
<Window x:Class="DemoWpf.DetailsPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
    presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Details"
Width="280" Height="200"
MinWidth="280" MinHeight="150">
<StackPanel Margin="20">
<Label Content="{Binding FullName}" FontWeight="Bold"
    FontSize="16"/>
<Label Content="Tutaj mo esz wy Żwietli  szczeg Ćy."
    FontSize="14"/>
</StackPanel>
</Window>
```

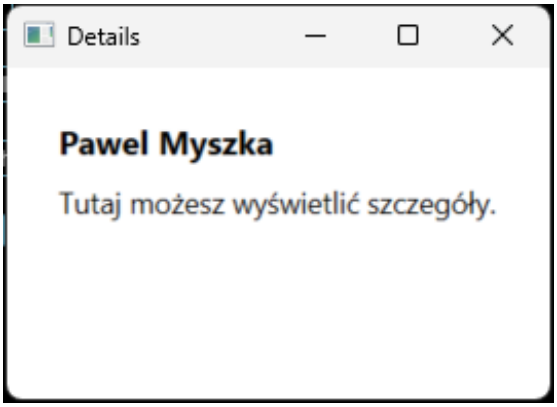
9.4 Efekt działania

Po kliknięciu przycisku Szczegóły użytkownik widzi nowe okno. Dzięki użyciu tego samego ViewModelu co w głównym oknie, dane pozostają zsynchronizowane.

9.5 Przykładowe uruchomienie aplikacji



Rysunek 8: Nowy przycisk w oknie głównym aplikacji.



Rysunek 9: Okno DetailsPage.

10 Zadanie 10 – Stylizacja i zasoby

10.1 Ustawienie koloru głównego odnośnie od trybu jasny/ciemny i koloru dla przycisków

W projekcie utworzono plik Resources/Styles.xaml, w którym zdefiniowano kolory i style:

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

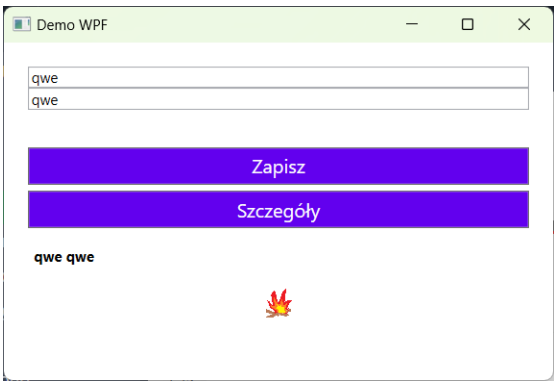
```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<Color x:Key="PrimaryColor">#FF6200EE</Color>
<SolidColorBrush x:Key="PrimaryBrush" Color="{StaticResource
    PrimaryColor}"/>

<Style TargetType="Button">
<Setter Property="Margin" Value="5"/>
<Setter Property="FontSize" Value="16"/>
<Setter Property="Padding" Value="8,4"/>
<Setter Property="Background" Value="{StaticResource
   PrimaryBrush}"/>
<Setter Property="Foreground" Value="White"/>
</Style>

<Color x:Key="LightBackgroundColor">#FFFFFFFF</Color>
<Color x:Key="DarkBackgroundColor">#FF1E1E1E</Color>
<SolidColorBrush x:Key="BackgroundBrush" Color="{
   DynamicResource LightBackgroundColor}"/>

</ResourceDictionary>
```

Styl został połączony w `App.xaml`, dzięki czemu jest dostępny globalnie w całej aplikacji.

10.2 Dodanie ikony

W folderze `Assets` umieszczono plik ikony `ikona.png` i ustawiono jego *Build Action* na `Resource`. W MainWindow.xaml dodano obrazek:

```
<Image Source="Assets/ikona.png" Width="32" Height="32"
   Margin="0,10,0,0"/>
```



Rysunek 10: Efekt zastosowania stylów i ikony w aplikacji.

11 Zadanie 11 – Lifecycle (WPF)

11.1 Implementacja w WPF

W App.xaml.cs obsłużono zdarzenia:

- `OnStartup` – log w konsoli: `Console.WriteLine("[INFO] Aplikacja uruchomiona")`
- `OnExit` – log w konsoli: `Console.WriteLine("[INFO] Aplikacja zamykana")`

Aby konsola była widoczna podczas debugowania, w pliku `DemoWpf.csproj` dodano w sekcji `Debug` zmianę typu wyjściowego na `Exe`:

```
        <PropertyGroup Condition=" '$(Configuration)|$(
            Platform)' == 'Debug|AnyCPU' ">
        <OutputType>Exe</OutputType>
        </PropertyGroup>
```

11.2 Efekt działania

```
(base) PS D:\stud\sem 5\Desktop\z3> dotnet run --project "d
    :\stud\sem 5\Desktop\z3\DemoWpf\DemoWpf.csproj" -c Debug
[INFO] Aplikacja uruchomiona
[INFO] Aplikacja zamykana
```

11.3 Wnioski

W WPF aplikacja działa dopóki użytkownik jej nie zamknie, a logi lifecycle pozwalają obserwować momenty startu i zakończenia. W przeciwieństwie do MAUI, nie ma natywnego stanu „usypiania” aplikacji przez system.

12 Podsumowanie

W ramach laboratorium zrealizowano tworzenie aplikacji w architekturze MVC (aplikacja webowa) oraz MVVM (aplikacja desktopowa WPF). Aplikacja WPF została przeprojektowana z podejścia opartego na code-behind do wzorca MVVM z wykorzystaniem biblioteki CommunityToolkit.Mvvm. Zastosowano data binding, walidację danych, konwertery wartości oraz Dependency Injection do zarządzania zależnościami. Logika ViewModelu została przetestowana za pomocą testów jednostkowych NUnit. Aplikację wzbogacono o stylizację globalną, nawigację między oknami oraz obsługę lifecycle. MVVM zapewnia separację logiki od UI, co zwiększa testowalność i utrzymywalność kodu.