.NET Multi-platform App UI (.NET MAUI) (3)

Programowanie Wizualne

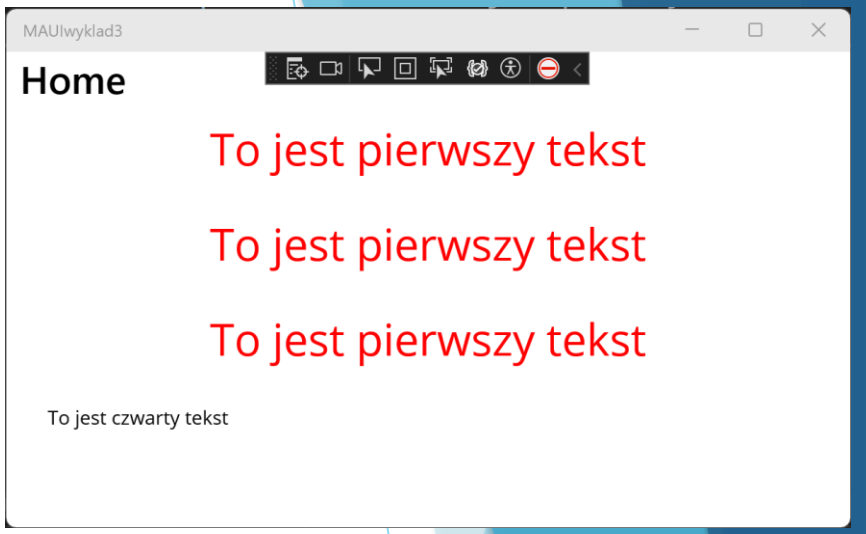
Paweł Wojciechowski

Instytut Informatyki, Politechniki Poznańskiej

2024

Style

Style - idea



- Ustawienie wartości właściwości

```
<VerticalStackLayout
    Padding="30,0"
    Spacing="25">
    <Label FontSize="Large" TextColor="Red" HorizontalOptions="Center">
        To jest pierwszy tekst</Label>
    <Label FontSize="Large" TextColor="Red" HorizontalOptions="Center">
        To jest pierwszy tekst</Label>
    <Label FontSize="Large" TextColor="Red" HorizontalOptions="Center">
        To jest pierwszy tekst</Label>
</VerticalStackLayout>
```

- Zamiast ręcznego ustawiania właściwości każdego z elementów można to zrobić raz – podobnie jak w css – to są właśnie style

Style

- Style są to obiekty tworzone w XAML-u z reguły definiowane w ramach tzw ResourceDictionary
- ResourceDictionary może być definiowany na różnych poziomach od aplikacji po pojedynczy komponent

```
<Label>
    <Label.Resources>
        <ResourceDictionary>
            <Style TargetType="…">

            </Style>
        </ResourceDictionary>
    </Label.Resources>
    To jest czwarty tekst
</Label>
```

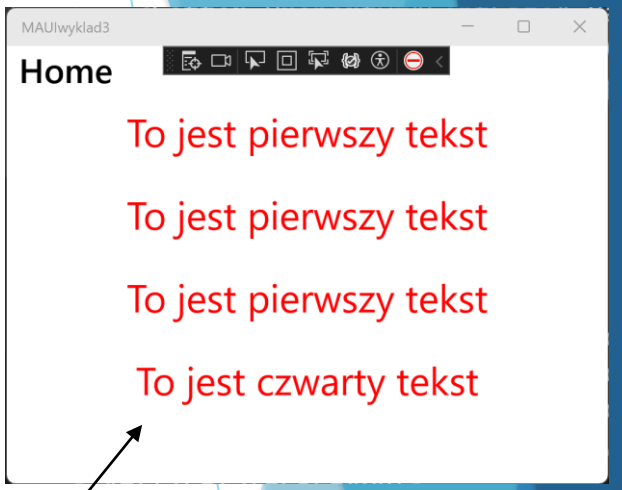
- Styl jest kolekcją obiektówy typu Setter, który ma właściwości:
  - Property – właściwość dla której chcemy ustawić wartość w ramach stylu
  - Value – wartość tej właściwości

Style - przykład

- W ramach definicji stylu trzeba obligatoryjnie podać właściwość TargetType – czyli typ elementu dla którego styl będzie nałożony

```
<ContentPage.Resources>
    <Style TargetType="Label">
        <Setter Property="FontSize" Value="Large"/>
        <Setter Property="TextColor" Value="Red"/>
        <Setter Property="HorizontalOptions" Value="Center"/>
    </Style>
</ContentPage.Resources>
```

- To automatycznie oznacza, że ostatnia etykieta, dla której pierwotnie nie mieliśmy ustawionych żadnych właściwości też będzie miała nadany styl
- Oczywiście wartości właściwości podanych w stylu można nadpisać innym (niżej położonym stylem) albo bezpośrednio w elemencie



Style - rodzaje

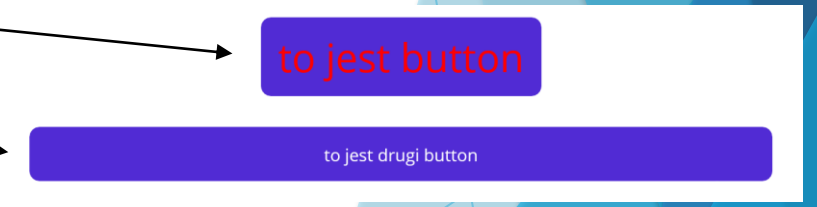
- Są dwa rodzaje styli, różniące się tym, czy mają ustawioną właściwość x:Key
  - Jawne (explicit):
    - Podaje się właściwość x:Key – jest to nazwa stylu

```
<Style x:Key="redLabel" TargetType="Label"></Style>
<Label Style="{StaticResource redLabel}"  …
```

    - Użycie stylu wymaga ustawienia właściwości Style

```
<Button Text="to jest button" Style="{StaticResource redLabel}"></Button>
<Button Text="to jest drugi button"/>
```

    - Można taki styl przypisać do elementów innego typu



  - Niejawne (implicite):
    - Są automatycznie nadawane wszystkim komponentom (elementom) o typie TargetType
    - Style NIE SĄ automatycznie przypisywane do klas dziedziczących po typach wskazanych jako TargetType. Można to zmienić ustawiając właściwość ApplyToDerivedTypes na True

Dziedziczenie styli

- W celu ograniczania duplikowania ustawień właściwości w stylach, mogą one po sobie dziedziczyć
- Dziedziczenie ustawia się właściwością BasedOn

```
<Style x:Key="smallRedLabel" BasedOn="{StaticResource redLabel}" TargetType="Label">
    <Setter Property="FontSize" Value="Small"/>
</Style>
```

- Oczywiście style dziedziczące po innych stylach mogą nadpisywać wartości właściwościom ustawionym w stylu rodzica oraz mogą definiować wartości innym właściwościom
- Style dziedziczące muszą mieć ten sam lub dziedziczący po nim typ ustawiony w ramach TargetType
- Dziedziczyć można tylko po stylach z „tego samego poziomu lub wyższego” – chodzi tutaj o zależności aplikacja->strona->kontrolki na stronie

Style statyczne i dynamiczne

- Raz przypisany styl z wykorzystaniem StaticResource nie będzie ulegał zmianie – jest statyczny
- Wszystkie zmiany (w kodzie) w tym stylu nie będą zmieniały wyglądu/zachowania komponentu
- Użycie zamiast StaticResource DynamicResource pozwoli takie zmiany uwzględnić – uważać jak to się stosuje – „dziwny” przykład na następnym slajdzie
- Wada – wydajność!
- Dziedziczenie styli dynamicznych realizuje się z wykorzystaniem właściwości BaseResourceKey

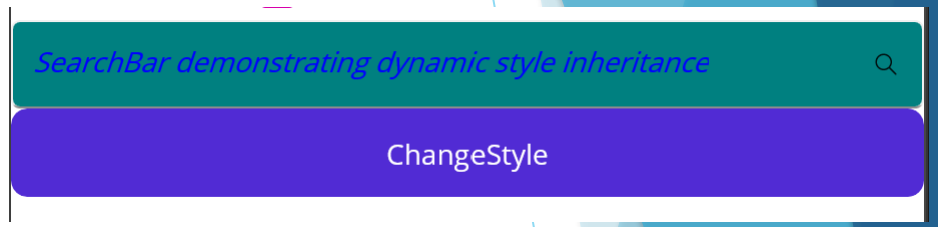
https://learn.microsoft.com/en-us/dotnet/maui/user-interface/styles/xaml?view=net-maui-8.0

Style statyczne i dynamiczne

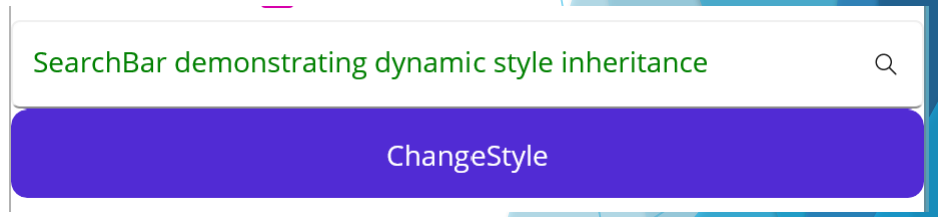
```
<ContentPage.Resources>
        <Style x:Key="baseStyle"
               TargetType="View">
            <Setter Property="VerticalOptions" Value="Center" />
        </Style>
        <Style x:Key="blueSearchBarStyle"
               TargetType="SearchBar"
               BasedOn="{StaticResource baseStyle}">
            <Setter Property="FontAttributes" Value="Italic" />
            <Setter Property="TextColor" Value="Blue" />
        </Style>
        <Style x:Key="greenSearchBarStyle"
               TargetType="SearchBar">
            <Setter Property="FontAttributes" Value="None" />
            <Setter Property="TextColor" Value="Green" />
        </Style>
        <Style x:Key="tealSearchBarStyle"
               TargetType="SearchBar"
               BaseResourceKey="blueSearchBarStyle">
            <Setter Property="BackgroundColor" Value="Teal" />
            <Setter Property="CancelButtonColor" Value="White" />
        </Style>
    </ContentPage.Resources>
    <StackLayout>
        <SearchBar Text="SearchBar demonstrating dynamic style inheritance"
                   Style="{StaticResource tealSearchBarStyle}" />
        <Button Text="ChangeStyle" Clicked="Button_Clicked" />
    </StackLayout>

private void Button_Clicked(object sender, EventArgs e)
{
    Resources["blueSearchBarStyle"] = Resources["greenSearchBarStyle"];
}
```

Zawartość okna po uruchomieniu



Zawartość okna naciśnięciu przycisku



Tu jest dynamiczne dziedziczenie

tu jest StaticResource

Klasy styli

- Klasy umożliwiają przypisanie komponentowi wielu klas styli bez konieczności wprowadzania dziedziczenia

```
<Style TargetType="Label" Class="Tekstzielony">
    <Setter Property="TextColor" Value="Green"/>
</Style>
<Style TargetType="Label" Class="TekstCzerwony">
    <Setter Property="TextColor" Value="Red"/>
</Style>
<Style TargetType="Label" Class="TextDuzy">
    <Setter Property="FontSize" Value="Large"/>
</Style>
```

To jest pierwszy tekst

To jest drugi tekst

To jest trzeci tekst

```
<Label StyleClass="TextDuzy,TekstCzerwony">To jest pierwszy tekst</Label>
<Label StyleClass="TextDuzy,Tekstzielony">To jest drugi tekst</Label>
<Label StyleClass="TextDuzy">To jest trzeci tekst</Label>
```

- Podanie niepoprawnych nazw klas styli nie prowadzi do błędu – może boleć ☺

Style – inne uwagi

- Jak usunąć domyślny styl dla komponentu? Czy w ogóle się da to zrobić?
- Nie można definiować na jednym poziomie dwóch styli jawnych dla tego samego typu obiektu.
- Innym sposobem definiowania styli jest css – dodaje się plik css do projektu i ustawia mu właściwość (we właściwościach pliku) AkcjaKompilacji na MauiCss
- Teoretycznie CSS style sheet jest parsowany w czasie uruchomienia, ale jak zmienić styl po skompilowaniu kodu? Czy to jest w ogóle możliwe?
- Nie wszystkie właściwości da się ustawić w css
- Nie poruszaliśmy w ogóle tematu AppTheme

Visual State

Visual State wprowadzenie

- Stan wizualny (VisualState) to klasa wyglądu komponentu zależąca od jego stanu początkowego. Na przykład Button może mieć kilka różnych stanów: czy jest włączony (Enabled), naciśnięty (Pressed), czy ma Focus.
- Stany zebrane są w grupy stanów, które wzajemnie się wykluczają.
- Każdy stan identyfikowany jest za pomocą nazwy (string)
- Zarządzaniem stanami zajmuje się menadżer Visual State Manager
- W .NET MAUI zdefiniowana jest grupa CommonStates zawierająca stany:
  - Normal
  - Disabled
  - Focused
  - Selected
  - PointOver
- Można definiować własne stany

Visual State przykład

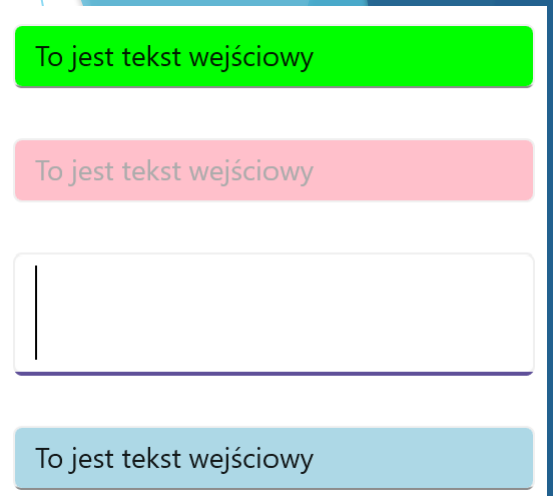
```
<Entry Style="{StaticResource specialEntry}" Text="To jest tekst wejściowy"></Entry>
<Entry Style="{StaticResource specialEntry}" Text="To jest tekst wejściowy" IsEnabled="False"></Entry>
<Entry Style="{StaticResource specialEntry}" Text="To jest tekst wejściowy" ></Entry>
<Entry Style="{StaticResource specialEntry}" Text="To jest tekst wejściowy"></Entry>

 <VisualStateGroup x:Name="CommonStates">
     <VisualState x:Name="Normal">
         <VisualState.Setters>
             <Setter Property="BackgroundColor" Value="Lime" />
         </VisualState.Setters>
     </VisualState>

     <VisualState x:Name="Focused">
         <VisualState.Setters>
             <Setter Property="FontSize" Value="36" />
         </VisualState.Setters>
     </VisualState>

     <VisualState x:Name="Disabled">
         <VisualState.Setters>
             <Setter Property="BackgroundColor" Value="Pink" />
         </VisualState.Setters>
     </VisualState>

     <VisualState x:Name="PointerOver">
         <VisualState.Setters>
             <Setter Property="BackgroundColor" Value="LightBlue" />
         </VisualState.Setters>
     </VisualState>
 </VisualStateGroup>
```



Visual State przykład

- Klasa VisualStateManager definiuje właściwość wiązaną (AttachedProperty) o nazwie VisualStateGroups
- VisualStateGroups jest typu VisualStateGroupList – kolekcja obiektów VisualStateGroup
- VisualStateGroup ma właściwość x:Name – nazwa grupy
- Klasa VisualStateGroup ma właściwość States (jest to tzw. Content Property)- kolekcję obiektów VisualState
- Każdy z obiektów VisualState powinien mieć nazwę x:Name
- Klasa VisualState ma właściwość Setters – kolekcja obiektów Setter
- Obiekt Setter ma dwie właściwości Property i Value, oznaczają one wartość (Value) jaką należy przypisać właściwości (Property)

Visual State – definiowanie w ramach stylu

```
<Style x:Key="specialEntry" TargetType="Entry">
    <Setter Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>
            <VisualStateGroup x:Name="CommonStates">
                <VisualState x:Name="Normal">
                    <VisualState.Setters>
                        <Setter Property="BackgroundColor" Value="Lime" />
                    </VisualState.Setters>
                </VisualState>
                …

            </VisualStateGroup>
        </VisualStateGroupList>
    </Setter>
</Style>
```

Visual State ustawianie stanów dla wielu elementów

- Można utworzyć VisualState dla pojedynczego obiektu, który ustawia właściwości innego elementu (w ramach zakresu)
- Służy do tego właściwość TargetName obiektu Setter, która wzkazuje nazwę (Name) obiektu, którego właściwość ma być ustawiona

Visual State ustawianie stanów dla wielu elementów - przykład

```
<StackLayout>
    <Label Text="What is the capital of France?" />
    <Entry x:Name="entry" Placeholder="Enter answer" />
    <Button Text="Reveal answer">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroupList>
                <VisualStateGroup x:Name="CommonStates">
                    <VisualState x:Name="Normal" />
                    <VisualState x:Name="Pressed">
                        <VisualState.Setters>
                            <Setter Property="Scale"
                                    Value="0.5" />
                            <Setter TargetName="entry"
                                    Property="Entry.Text"
                                    Value="Paris" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateGroupList>
        </VisualStateManager.VisualStateGroups>
    </Button>
</StackLayout>
```

W dokumentacji jest błąd, a właściwie to teoretycznie go nie ma, ale kod nie działa. Trzeba dodać tę linię.
W dokumentacji jest też napisane, że tekst z entry wraca po puszczeniu przycisku. Nie wraca.

https://learn.microsoft.com/en-us/dotnet/maui/user-interface/visual-states?view=net-maui-8.0

Triggers

Wyzwalacze - Triggers

- Wyzwalacze jest to mechanizm pozwalający na zmianę wyglądu elementów kontrolnych w reakcji na zdarzenia albo zmianę danych
- Podobnie jak style mogą być zdefiniowane w XAML-u na poziomie kontrolki, strony albo aplikacji.
- Definicja składa się z sekcji Trigger i Settera (-ów)
- Wartość właściwości jest ustawiana tylko dopóki warunki zawarte w triggerze są spełnione

```
<Entry>
    <Entry.Triggers>
        <Trigger TargetType="Entry"
                 Property="Text"
                 Value="coś">
            <Setter Property="BackgroundColor"
                    Value="Yellow"/>
        </Trigger>
    </Entry.Triggers>
</Entry>
```

Wyzwalacze – Triggers (2)

- Wyzwalacze mogą być definiowane na poziomie styli

```
<Style TargetType="Entry" >
    <Style.Triggers>
        <Trigger TargetType="Entry"
                 Property="IsFocused"
                 Value="True">
            <Setter Property="BackgroundColor"
                    Value="Aqua"/>
            <Setter Property="FontSize"
                    Value="Large"/>
        </Trigger>
    </Style.Triggers>
</Style>
```

Wyzwalacze – DataTriggers

- Wyzwalacze danych (?) pozwalają na definiowanie zależności od danych
- Odbywa się do przez wiązanie - Binding

```
<Entry x:Name="testEntry">
    <Entry.Triggers>
        <Trigger TargetType="Entry"
      Property="Text"
      Value="coś">
            <Setter Property="BackgroundColor"
       Value="Yellow"/>
        </Trigger>
    </Entry.Triggers>
</Entry>

<Button Text="to jest drugi button" IsEnabled="True">
    <Button.Triggers>
        <DataTrigger TargetType="Button"
                     Binding="{Binding Source={x:Reference testEntry}, Path=Text}"
                     Value="ktoś">
            <Setter Property="IsEnabled"
                    Value="False"/>
        </DataTrigger>
    </Button.Triggers>
</Button>
```

UWAGA:
prawdopodobnie DataTriggers mają bug polegający na tym, że po zakończeniu spełniania warunku, wartość właściwości nie wraca do poprzedniej wartości.

Wyzwalacze – EnterActions i ExitActions

- W ramach wyzwalaczy można zdefiniować kolekcję obiektów (metod Invoke tych obiektów) które będą wywoływane:
  - Na początku, kiedy warunki wyzwolenia wyzwalacza zostaną spełnione – EnterActions
  - Na końcu, kiedy warunki przestaną być spełnione – ExitActions
- Każdy z tych kolekcji może przyjmować obiekty dziedziczące po klasie abstrakcyjnej TriggerAction

```
public class EnableControlTriggerAction : TriggerAction<Button>
{
    public bool IsButtonEnabled { get; set; }
    protected override void Invoke(Button sender)
    {
        sender.IsEnabled = IsButtonEnabled;
    }
}
```

Wyzwalacze – EnterActions i ExitActions (2)

```
<Button Text="to jest drugi button" IsEnabled="True">
    <Button.Triggers>
        <DataTrigger TargetType="Button"
                     Binding="{Binding Source={x:Reference testEntry}, Path=Text}"
                     Value="ktoś">
            <DataTrigger.EnterActions>
                <local:EnableControlTriggerAction IsButtonEnabled="False"/>
            </DataTrigger.EnterActions>
            <DataTrigger.ExitActions>
                 <local:EnableControlTriggerAction IsButtonEnabled="True"/>
            </DataTrigger.ExitActions>
        </DataTrigger>
    </Button.Triggers>
</Button>
```

Efekt identyczny – dodano setter i usunięto EnterActions

```
<Button Text="to jest drugi button" IsEnabled="True">
     <Button.Triggers>
         <DataTrigger TargetType="Button"
                      Binding="{Binding Source={x:Reference testEntry}, Path=Text}"
                      Value="ktoś">
             <Setter Property="IsEnabled"
                     Value="False"/>
             <DataTrigger.ExitActions>
                  <local:EnableControlTriggerAction IsButtonEnabled="True"/>
             </DataTrigger.ExitActions>
         </DataTrigger>
     </Button.Triggers>
</Button>
```

Wyzwalacze – EnterActions i ExitActions (3)

- Wygodne rozwiązanie, które sprawia, że błąd z DataTriggerem nie musi być tak uciążliwy
- Trzeba jednak pamiętać, że w ten sposób jest więcej pracy – trzeba zaimplementować klasę, mamy dłuższy zapis w XAML-u
- Problemem jest również to, że w przypadku EnterActions i ExitActions nie ma automatycznego powrotu do wartości sprzed wyzwolenia triggera. Możemy ją sami zapamiętać. Tylko, że korzystając z ResourceDictionary jedna instancja będzie współdzielona między wszystkimi obiektami, które z takiej akcji korzystają.

Wyzwalacze – MultiTriggers

- Działa podobnie do Triggera ale ma kolekcję warunków do spełnienia.
- Warunki definiuje się we właściwości Conditions

```
<Entry x:Name="email" Text="" />
<Entry x:Name="phone" Text="" />
<Button Text="Save">
    <Button.Triggers>
        <MultiTrigger TargetType="Button">
            <MultiTrigger.Conditions>
                <BindingCondition Binding="{Binding Source={x:Reference email},
                                  Path=Text.Length}"
                                  Value="0" />
                <BindingCondition Binding="{Binding Source={x:Reference phone},
                                  Path=Text.Length}"
                                  Value="0" />
            </MultiTrigger.Conditions>
            <Setter Property="IsEnabled" Value="False" />
            <!-- multiple Setter elements are allowed -->
        </MultiTrigger>
    </Button.Triggers>
</Button>
```

Uwaga na tego typu odwołania. Jeśli nie zostanie ustawiona wartość właściwości Text pole Length może nie działać

https://learn.microsoft.com/en-us/dotnet/maui/fundamentals/triggers?view=net-maui-8.0

Inne wyzwalacze

- EventTrigger – reakcja na wystąpienie zdarzenia (nie ma wartości)
  - Nie obsługują EnterActions i ExitActions
  - Implementacja akcji takiego triggera jest obiektem klasy TriggerAction<T>
- Wyzwalacze stanów – StateTriggers:
  - Adaptive Triggers
  - Device State Triggers
  - Orientation State Triggers

Nawigacja po stronach

Nawigacja po stronach

- Powłoka interfejsu użytkownika (Shell) zawiera środowisko nawigacji oparte na identyfikatorze URI, które używa tras (Route). Dodatkowo zapewnia również przechodzenie w tył.
- W klasie Shell definiując (dodając) nową stronę w xaml, dodawaliśmy obiekt typu ShellContent. Właściwość Route jest adresem URL.

```
<FlyoutItem>
    <ShellContent
        Title="Home"
        ContentTemplate="{DataTemplate local:MainPage}"
        Route="MainPage" />
    <ShellContent
        Title="Dynamic style"
        ContentTemplate="{DataTemplate local:DynamicStylePage}"
        Route="DynamicStylePage" />
    <ShellContent
        Title="Visual State"
        ContentTemplate="{DataTemplate local:VisualState}"
        Route="VisualState" />
</FlyoutItem>
```

Nawigacja po stronach (2)

- Przejście do innej strony można również definiować w konstruktorze klasy Shell wykorzystując klasę Routing i jej metodę statyczną RegisterRoute

```
Routing.RegisterRoute("NazwaRoute", typeof(VisualState));
Routing.RegisterRoute(nameof(VisualState), typeof(VisualState));
```

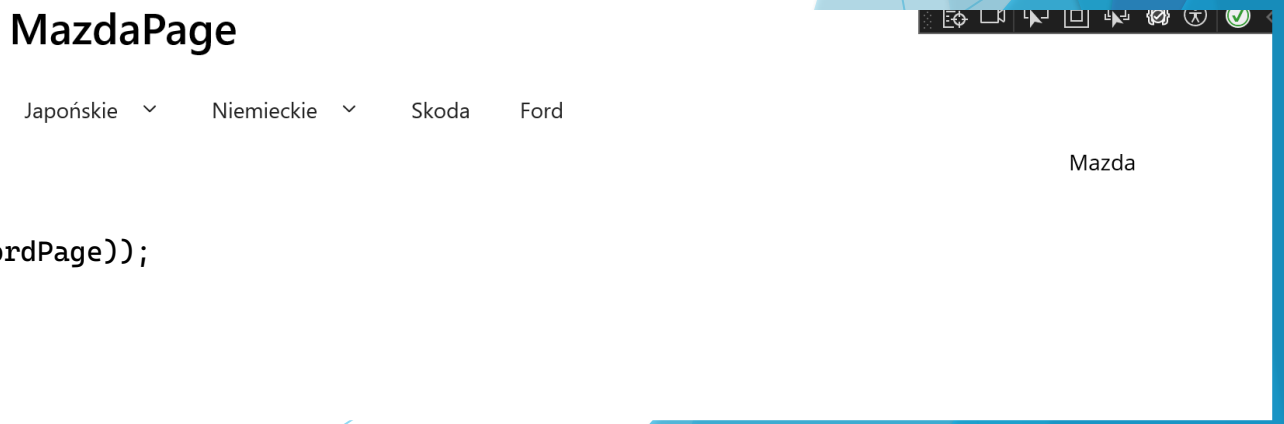
- Dostęp do tych stron (do tych zdefiniowanych tylko w konstruktorze klasy Shell również) z dowolnego miejsca w aplikacji jest możliwy na podstawie ścieżki globalnej.
- Nawigację do żądanej strony wywołuje się wykorzystując metodę asynchroniczną GoToAsync obiektu Shell (można go pobrać z właściwości Shell.Current).

Nawigacja po stronach (3)

- Istnieje obiekt stosu nawigacji (Navigation Stack)
- Podczas nawigacji podając ścieżkę bezwzględną nadpisujemy aktualny stos nawigacji:
  - //japan/honda/accord
- Albo można podać ścieżkę względną (wg. dokumentacji):
  - route – Hierarchia zostanie przeszukana w celu znalezienia transy route w górę od bieżącej pozycji – znaleziona strona zostanie dodana do stosu nawigacji
  - /route – j.w. ale w dół od bieżącej pozycji
  - //route – tak samo jak route ale znaleziona strona zastąpi stos nawigacyjny
  - ///route – tak samo jak /route ale znaleziona strona zastąpi stos nawigacyjny
- Można używać tras typu „..” żeby wykonać nawigację wstecz, albo „../trasa” albo nawet „../../trasa”

Nawigacja po stronach (4) - przykład

```
<FlyoutItem Route="Cars">
    <Tab Title="Japońskie" Route="japan">
        <ShellContent Title="Honda" Route="HondaPage" ContentTemplate="{DataTemplate local:HondaPage}"/>
        <ShellContent Title="Mazda" Route="MazdaPage" ContentTemplate="{DataTemplate local:MazdaPage}"/>
        <ShellContent Title="Toyota" Route="ToyotaPage" ContentTemplate="{DataTemplate local:ToyotaPage}"/>
    </Tab>
    <Tab Title="Niemieckie" Route="german">
        <ShellContent Title="BMW" Route="BMWPage" ContentTemplate="{DataTemplate local:BMWPage}"/>
        <ShellContent Title="Mercedes" Route="MercedesPage" ContentTemplate="{DataTemplate local:MercedesPage}"/>
        <ShellContent Title="Opel" Route="OpelPage" ContentTemplate="{DataTemplate local:OpelPage}"/>
    </Tab>
    <ShellContent Title="Skoda" Route="SkodaPage" ContentTemplate="{DataTemplate local:SkodaPage}"/>
    <ShellContent Title="Ford" Route="FordPage" ContentTemplate="{DataTemplate local:FordPage}"/>
</FlyoutItem>
<ShellContent
    Title="O programie"
    Route="CarsPage"
    ContentTemplate="{DataTemplate local:MainPage}" />
```




```
public AppShell()
{
    InitializeComponent();
    Routing.RegisterRoute("japan/HondaPage/Accord", typeof(Pages.AccordPage));
}
```

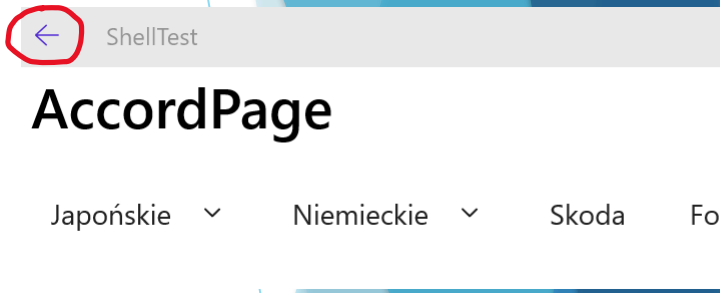
Uwagi do przykładu

- Dodając przycisk nawigacji na stronie HondaPage z przekierowaniem do Accord z kodem na naciśnięcie takiego przycisku:

```
await Shell.Current.GoToAsync("Accord");
```

  powoduje przejście do strony Accord i pojawienie się strzałki powrotu.

- Taki sam efekt uzyska się dodając przycisk nawigacji np. do strony Mercedesa, oczywiście z tą różnicą, że powrót spowoduje przejście do strony, z której do strony Accord przyszliśmy
- Nie widzę żadnej różnicy czy wywołana zostanie strona Accord, czy /Accord
- Przejście do strony CarsPage wymaga użycia // lub ///



Przykład z dokumentacji

- Dla tak zdefiniowanych tras:

```
Routing.RegisterRoute("monkeys/details", typeof(MonkeyDetailPage));
Routing.RegisterRoute("bears/details", typeof(BearDetailPage));
Routing.RegisterRoute("cats/details", typeof(CatDetailPage));
Routing.RegisterRoute("dogs/details", typeof(DogDetailPage));
Routing.RegisterRoute("elephants/details", typeof(ElephantDetailPage));
```

różnica przy wyszukiwaniu details a /details może wystąpić – nie sprawdzałem.

Przekazywanie danych w ramach nawigacji

- W ramach nawigacji można przekazać parametry do strony jak w url-u. np:

```
Shell.Current.GoToAsync($"//japan/HondaPage/Accord?licznik={count}");
```

- Odebranie takich danych dla następującej konfiguracji:
  - Strona odbierająca: AccordPage
  - Dodany viewmodel o nazwie: TestViewModel
  - Wykorzystany CommunityToolkit.Mvvm
  - Automatyczne dependency injection – w konstruktorze aplikacji MauiProgram rejestrujemy viewmodel i stronę

```
builder.Services.AddSingleton<viewmodel.TestViewModel>();
builder.Services.AddSingleton<Pages.AccordPage>();
```

  - Viewmodel ma atrybut QueryProperty, gdzie mamy:
    - Nazwę właściwości
    - Nazwę parametru

```
[QueryProperty(nameof(Count), "licznik")]
 public partial class TestViewModel:ObservableObject
 {
     [ObservableProperty]
     int count = 0;
 }
```

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ShellTest.Pages.AccordPage"
             xmlns:viewmodel="clr-namespace:ShellTest.viewmodel"
             x:DataType="viewmodel:TestViewModel"
             Title="AccordPage">
    <VerticalStackLayout>

        <HorizontalStackLayout>
            <Label Text="Aktualny licznik: "/>
            <Label Text="{Binding Count}"/>
        </HorizontalStackLayout>
    </VerticalStackLayout>
</ContentPage>
```

```
public AccordPage(TestViewModel vm)
{
    InitializeComponent();
    BindingContext = vm;
}
```

Przekazywanie danych w ramach nawigacji - obiekty

- Obiekty trudniej przekazać jako atrybut w wywołaniu URL(I).

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

- Dodajemy właściwość do viewmodel

```
[ObservableProperty]
Person person;
```

- Dodajemy QueryProperty:

```
[QueryProperty(nameof(Person), "osoba")]
```

- I modyfikujemy wywołanie:

```
Dictionary<string, object> dict = new Dictionary<string, object>();
dict.Add(
    "osoba", new Model.Person() {  Id=1, Name="Grzegorz Brzęczyszczykiewicz"}
    );
Shell.Current.GoToAsync($"//japan/HondaPage/Accord?licznik={count}",
                        dict);
```