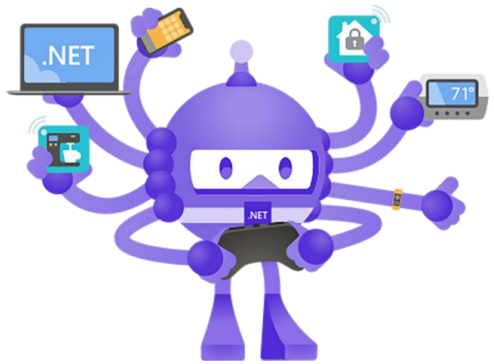
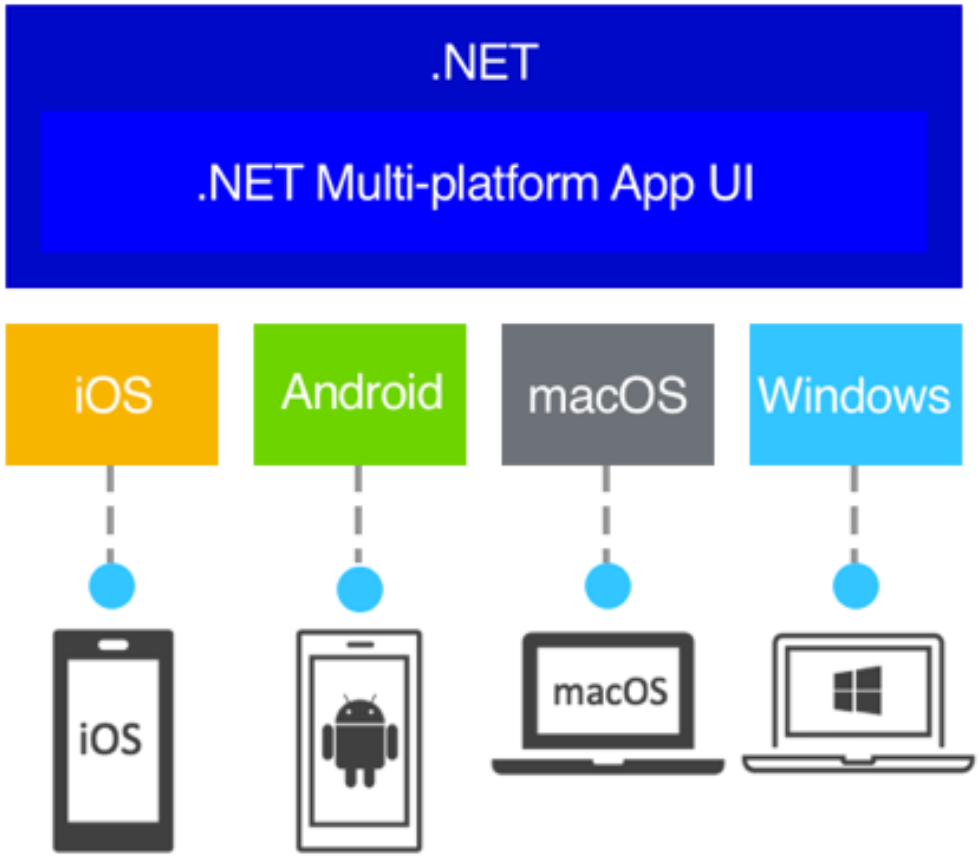
Wprowadzenie do .NET MAUI

Bartosz Miąskowski

Czym jest .NET MAUI?

- .NET Multi-platform App UI (.NET MAUI) to wieloplatformowe rozwiązanie (framework) do tworzenia natywnych aplikacji mobilnych i desktopowych przy użyciu języków C# i XAML.
- Korzystając z .NET MAUI, możesz tworzyć aplikacje na systemy Android, iOS, macOS oraz Windows, opierając się na jednym, wspólnym kodzie źródłowym.





MAUI jako część platformy .NET

.NET MAUI jest elementem platformy .NET, czyli tego samego środowiska, w którym używamy języka C#.

Oznacza to, że:

- logika aplikacji pisana jest w C#
- korzystamy z tych samych zasad programowania
- uczymy się technologii, która pasuje do reszty ekosystemu .NET

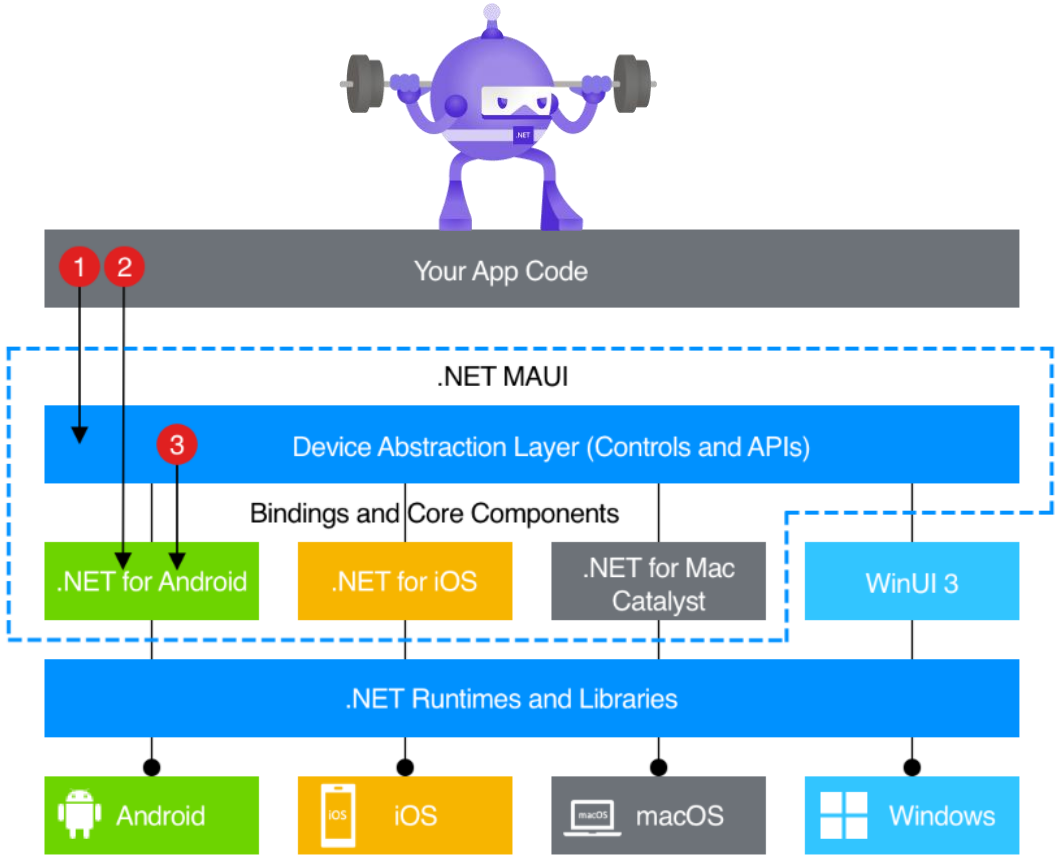
Jak działa .NET MAUI?

- .NET MAUI łączy w jedno różne platformy systemowe: Android, iOS, macOS i Windows.
- Zamiast pisać osobną aplikację dla każdego systemu, programista pracuje z jednym wspólnym API. Dzięki temu możliwe jest podejście – „**napisz raz – uruchom wszędzie**”.
- Kod aplikacji powstaje jeden, a MAUI zajmuje się dopasowaniem go do konkretnej platformy.

Na diagramie:

- Kod aplikacji komunikuje się głównie z warstwą MAUI.
- W razie potrzeby może sięgnąć bezpośrednio do API systemu.
- MAUI przekazuje wszystko do natywnych API platformy.

MAUI działa więc jak pośrednik między aplikacją a systemem operacyjnym.



Jak działa .NET MAUI?

Wszystkie aplikacje MAUI działają na tej samej podstawie – bibliotece klas .NET. To właśnie ona:

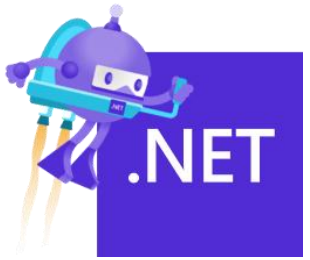
- udostępnia podstawowe typy (np. string, listy, daty),
- ukrywa różnice pomiędzy systemami operacyjnymi,
- sprawia, że logika aplikacji może być wspólna.

Kod aplikacji nie musi wiedzieć, czy działa na telefonie czy komputerze – *tutaj w pewnym uproszczeniu*.

Choć kod aplikacji jest wspólny, środowisko wykonawcze zależy od platformy:

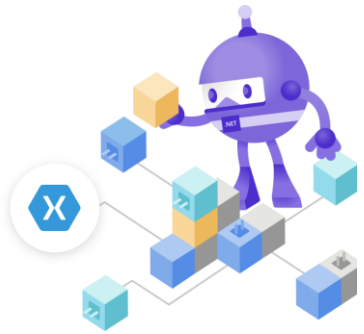
- na Androidzie, iOS i macOS aplikacja działa na środowisku Mono,
- na Windowsie wykorzystywany jest runtime .NET (CLR),

Dla programisty nie ma to większego znaczenia – MAUI ukrywa te różnice.



.NET MAUI – interfejs użytkownika

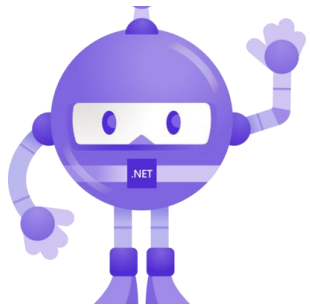
- Każdy system operacyjny inaczej definiuje interfejs użytkownika, inaczej obsługuje kontrolki oraz ma własne zasady wyglądu i zachowania aplikacji. Oznacza to, że bez .NET MAUI tworząc aplikację cross-platformową dla każdej aplikacji musielibyśmy osobno stworzyć zarówno GUI jak i cały code-behind.
- .NET MAUI dostarcza jedno wspólne podejście do budowy interfejsu użytkownika:
  - jeden opis UI,
  - jedna struktura aplikacji,
  - jeden projekt.
- Jednocześnie MAUI zapewnia, że:
  - interfejs jest renderowany natywnie,
  - aplikacja wykonana w MAUI, wygląda jak prawdziwa natywna aplikacja – i nią w rzeczywistości jest.



Kompilacja aplikacji .NET MAUI

Aplikacje MAUI są kompilowane natywnie:

- Android – kompilacja do IL, potem JIT przy uruchomieniu.
- iOS – pełna kompilacja AOT do kodu maszynowego.
- macOS – aplikacja desktopowa przez Mac Catalyst.
- Windows – aplikacja oparta o WinUI 3.

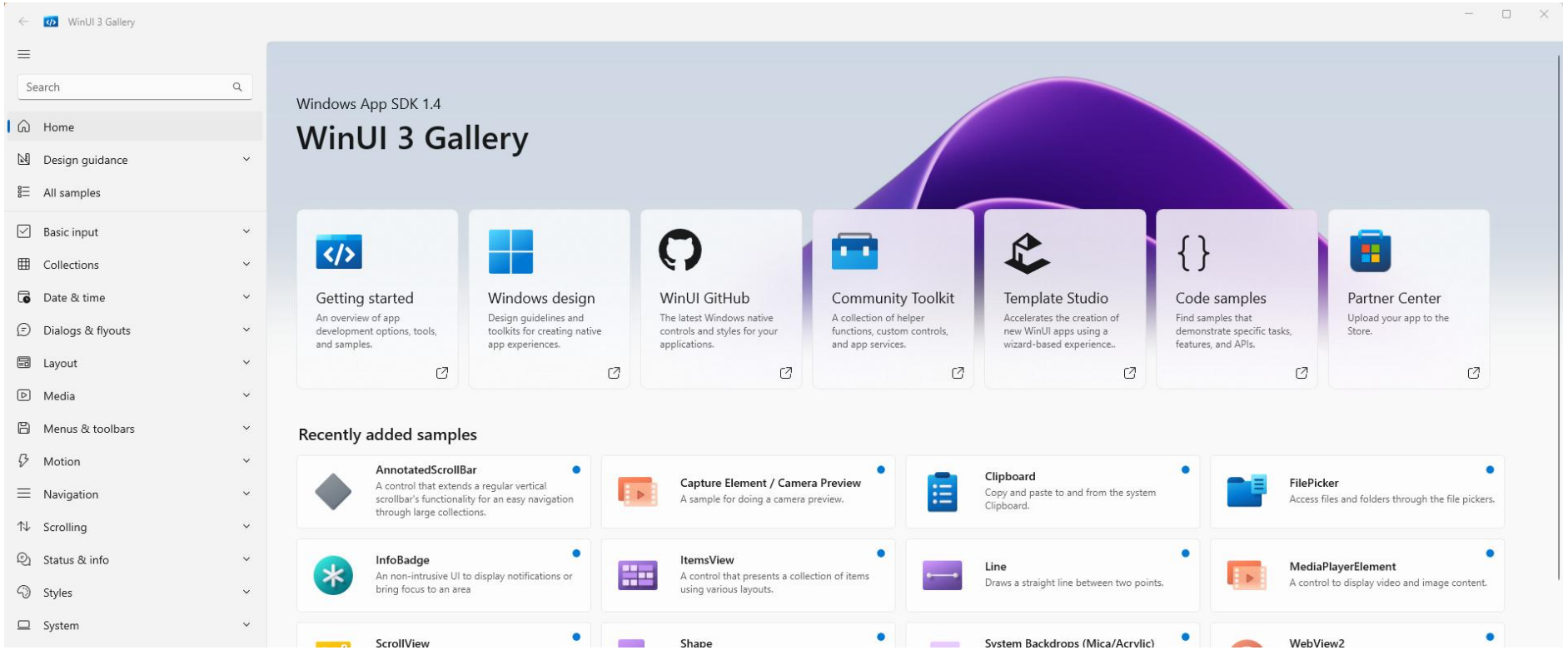


.NET MAUI - Windows

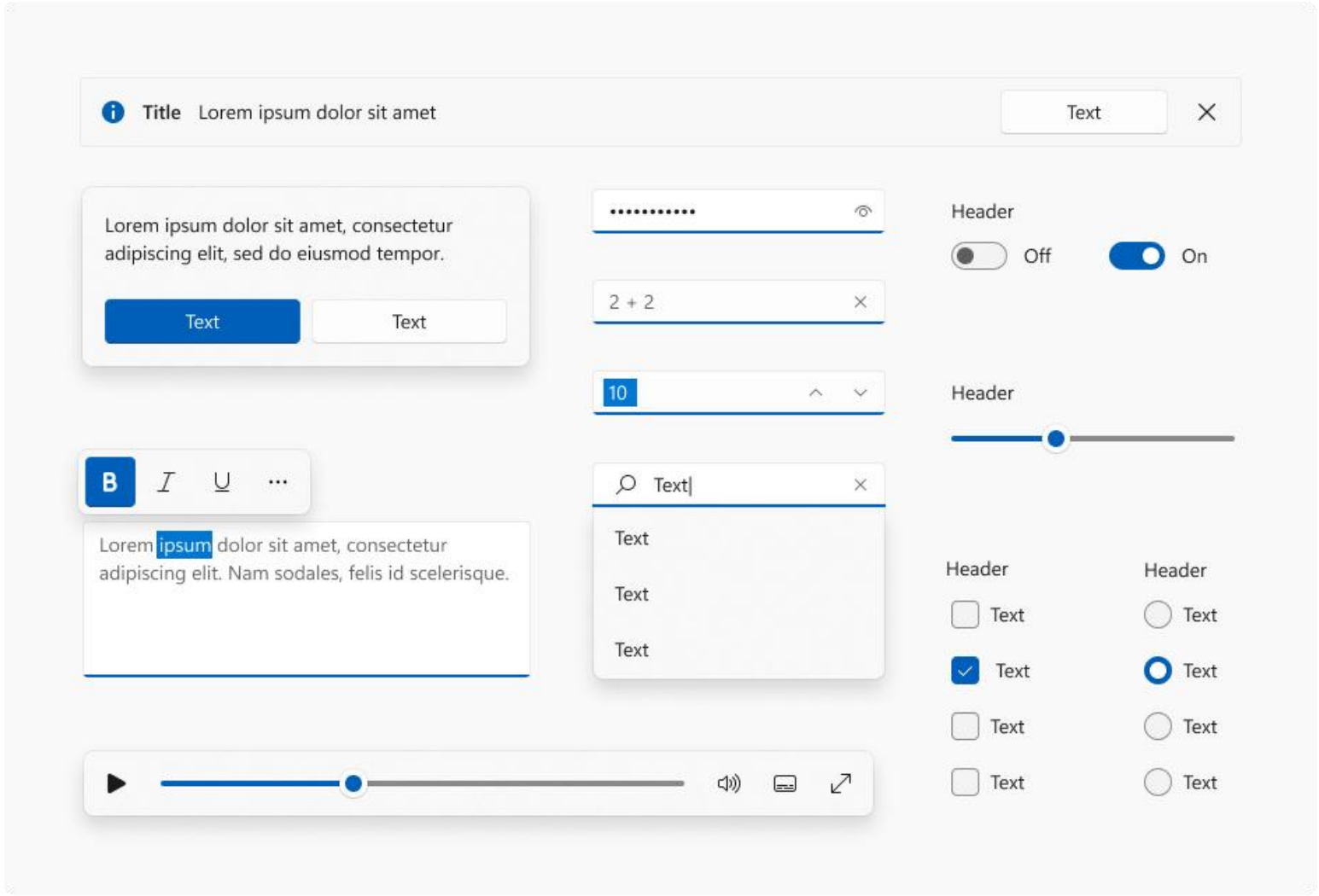
- Gdy budujesz aplikację w .NET MAUI na Windows, jej natywną technologią UI jest WinUI 3 (*część Windows App SDK*). To znaczy, że finalnie Twoje okna, przyciski, pola tekstowe itd. są tworzone jako prawdziwe kontrolki WinUI, a nie „udawane” elementy.

- MAUI działa jak tłumacz / warstwa pośrednia:
  - Ty ustawiasz właściwości MAUI (*np. tekst przycisku, rozmiar, marginesy*).
  - MAUI mapuje je na odpowiedniki w świecie Windows.
  - Pod spodem tworzone są natywne obiekty WinUI 3 i to one są faktycznie renderowane.

WinUI 3 – przykładowe UI



WinUI 3 – przykładowe kontrolki

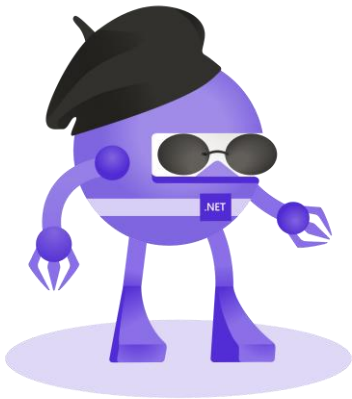


.NET MAUI – macOS

- Na macOS aplikacje .NET MAUI nie są pisane bezpośrednio w AppKit. Zamiast tego MAUI korzysta z technologii Mac Catalyst, dostarczanej przez Apple.

- Mac Catalyst pozwala uruchamiać aplikacje:
  - pisane jak aplikacje iOS (*UIKit*),
  - jako pełnoprawne aplikacje desktopowe macOS.

- Mac Catalyst działa jak most między światem mobilnym a desktopowym Apple:
  - bazuje na UIKit (*znanym z iOS*)
  - rozszerza go o elementy desktopowe (*okna, menu, skróty klawiszowe*)
  - pozwala aplikacji działać natywnie na macOS.



.NET MAUI – iOS

- Na iOS aplikacje .NET MAUI są budowane bezpośrednio w oparciu o UIKit – czyli ten sam framework, z którego korzystają natywne aplikacje pisane w Swift lub Objective-C.

- Aplikacje pisane na iOS są specyficzne, ponieważ Apple narzuca konkretne zasady:
  - brak kompilacji JIT,
  - ścisłe reguły bezpieczeństwa,
  - kontrola wydajności i zachowania aplikacji.
- Dlatego właśnie, aplikacje .NET MAUI na iOS są:
  - kompilowane AOT (*Ahead-Of-Time*),
  - zamieniane z C# bezpośrednio na natywny kod ARM – bardzo wysoka wydajność, pełna zgodność z iOS.

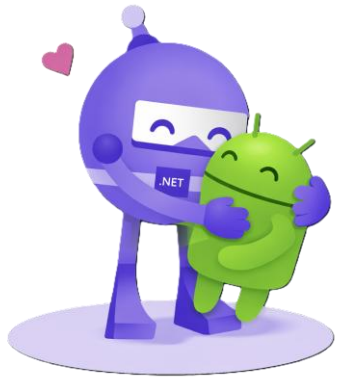
- .NET MAUI daje możliwość bezpośrednio skorzystać z API iOS – nie zamyka drogi do natywnych funkcji systemu.

.NET MAUI – Android

- Na Androidzie aplikacje .NET MAUI są budowane jako prawdziwe, natywne aplikacje Androidowe. Pod spodem wykorzystywane są:
  - natywne widoki Androida,
  - standardowy model aplikacji Android (*ekrany, cykl życia, zdarzenia*).

- Na Androidzie aplikacje MAUI:
  - działają poprzez środowisko Mono,
  - są kompilowane do IL (*Intermediate Language*),
  - przy uruchomieniu korzystają z JIT (*Just-In-Time compilation*).

- Tak samo, jak w przypadku pozostałych platform, możesz sięgnąć bezpośrednio do API Androida z poziomu MAUI.



.NET MAUI – cross-platform

- Cross-platform oznacza, że jedna aplikacja może działać na wielu systemach operacyjnych. Nie chodzi jednak o kopiowanie programu z jednego systemu na drugi, ale o wspólną logikę i strukturę, które są dostosowywane do konkretnej platformy. W MAUI:
  - kod aplikacji jest jeden,
  - platform jest wiele,
  - efekt końcowy zawsze jest natywną aplikacją.

- .NET MAUI jest frameworkiem cross-platform, co oznacza, że jego celem jest udostępnienie jednego wspólnego zestawu narzędzi dla wielu systemów operacyjnych. Aby dana kontrolka mogła znaleźć się w MAUI, musi:
  - istnieć na wszystkich wspieranych platformach w podobnej formie,
  - dać się obsłużyć w spójny sposób,
  - zachowywać się przewidywalnie niezależnie od systemu.

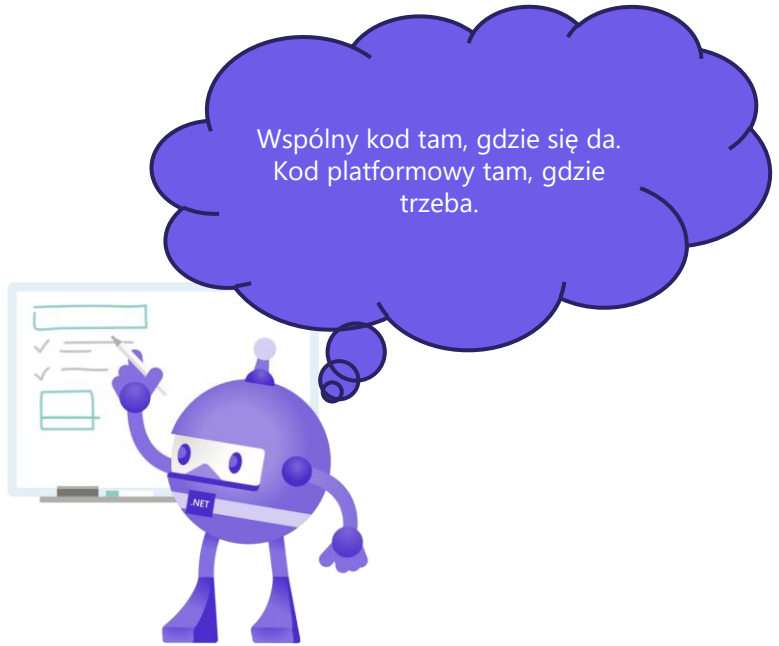
- Jeżeli dany element interfejsu użytkownika jest bardzo specyficzny dla jednego systemu, albo działa zupełnie inaczej na różnych platformach – nie trafia on do wspólnego zestawu kontrolek MAUI.

Obsługa różnic między platformami

- Każdy system operacyjny:
  - ma inne możliwości,
  - ma inne ograniczenia,
  - ma inne zasady projektowania aplikacji.

Dlatego nie wszystko da się zrobić dokładnie tak samo na Androidzie, iOS, Windows i macOS. .NET MAUI zakłada, że te różnice są normalne i nieuniknione.

```
        // Różnice zależne od platformy:
#if WINDOWS
        label.Text = "Cześć z MAUI na Windows";
        label.Margin = new Thickness(24);
#elif ANDROID
        label.Text = "Cześć z MAUI na Androida";
        label.Margin = new Thickness(12);
#elif IOS
        label.Text = "Cześć z MAUI na iOS";
        label.Margin = new Thickness(16);
#endif
```

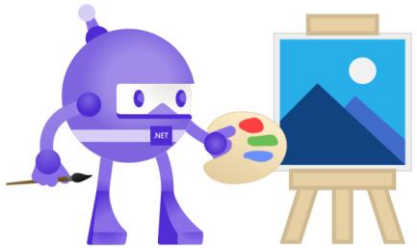


Język XAML

- XAML to język znaczników, który służy do opisywania wyglądu i struktury interfejsu użytkownika. Nie opisujemy w nim *co aplikacja robi*, tylko *jak wygląda*. XAML pozwala określić:
  - jakie elementy znajdują się na ekranie,
  - jak są ułożone względem siebie,
  - jakie mają właściwości (*tekst, kolor, rozmiar*).

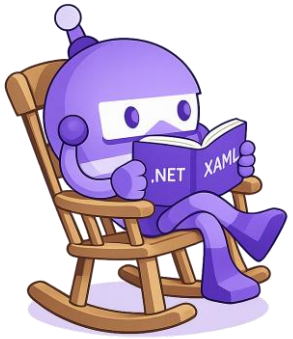
- XAML jest językiem deklaratywnym. Oznacza to, że:
  - opisujemy co ma być na ekranie,
  - a nie jak to tworzyć krok po kroku.

Czytając XAML, można „*zobaczyć*" interfejs aplikacji jeszcze zanim ją uruchomimy.



Język XAML

```xaml
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             x:Class="MyApp.MainPage">
    <VerticalStackLayout Spacing="12" Padding="16">
        <Label
            Text="Witaj w .NET MAUI!"
            FontSize="24"
            HorizontalOptions="Center" />

        <Button
            Text="Kliknij mnie" />
    </VerticalStackLayout>
</ContentPage>
```

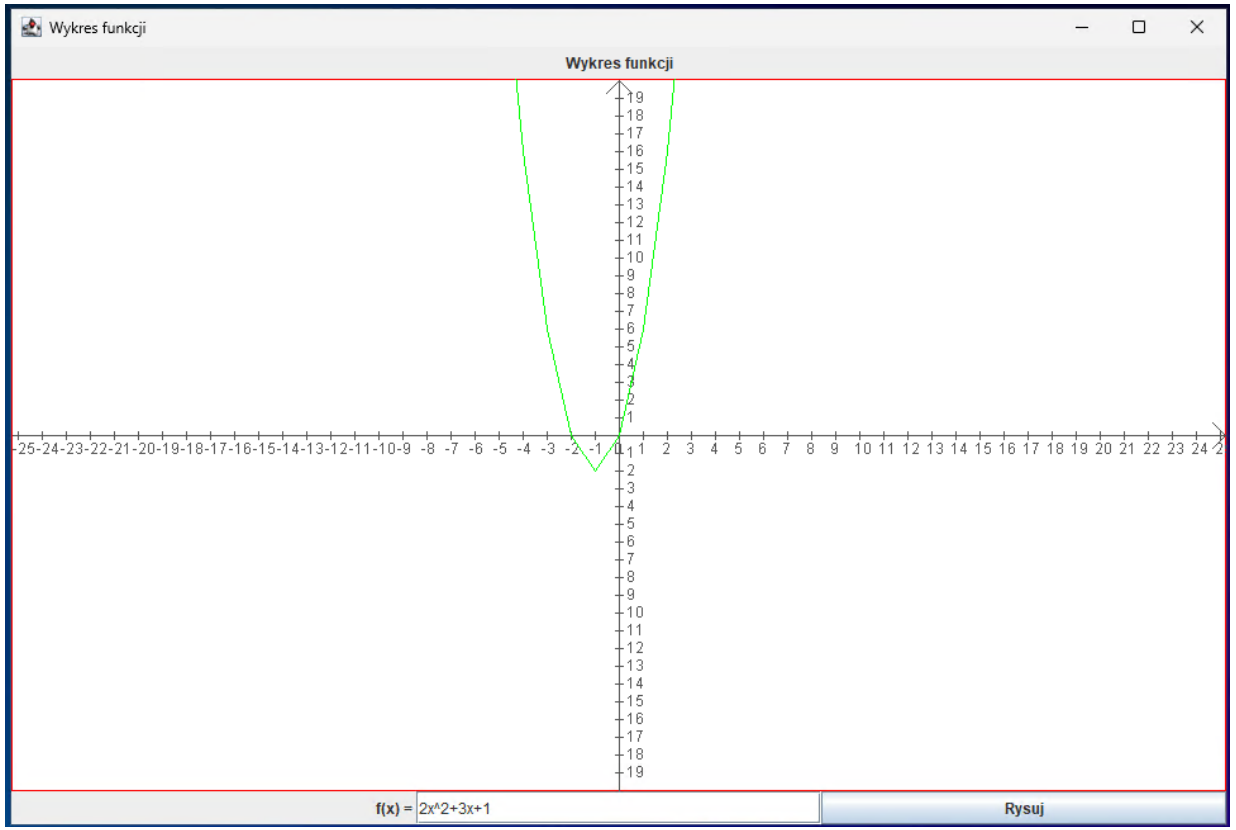


Dlaczego XAML jest taki cudowny?


```python

# Klasa reprezentująca jeden, mały element (dwa guziki oraz tytuł wiadomości)
class Entry(QWidget):
    def __init__(self, main_window, entry, channel_uri):
        super().__init__()
        self.entry_layout = QGridLayout(self)
        self.entry = entry
        self.main_window = main_window

        # Guziki
        btn_show = QPushButton("Pokaż")
        # btn_show.resize(80, 30)
        btn_show.clicked.connect(self.show_entry)
        self.entry_layout.addWidget(btn_show, 0, 0, 0, 1)
        btn_show = QPushButton("Otwórz")
        btn_show.clicked.connect(self.open_entry_in_browser)
        self.entry_layout.addWidget(btn_show, 0, 1, 0, 2)


        # Krótka treść wiadomości
        # (możliwe "przycięcie" wyświetlanych wiadomości do 100 znaków)
        entry_content = f"[{channel_uri}] {entry.Title}"
        # if len(entry_content) > 100:
        #     entry_content = entry_content[0:100]
        # else:
        #     entry_content = entry_content.ljust(100)
        file_name = QLabel(entry_content)
        self.entry_layout.addWidget(file_name, 0, 3, 0, 25)
        self.setLayout(self.entry_layout)

    # Obsługa guzika "Pokaż"
    def show_entry(self):
        full_content = self.entry.Content
        entry_content = full_content[full_content.rfind('>')+1:len(full_content)]
        info_box(entry_content)

    # Obsługa guzika "Otwórz"
    def open_entry_in_browser(self):
        webbrowser.open(self.entry.URL, new=2)

```

Python – qt5

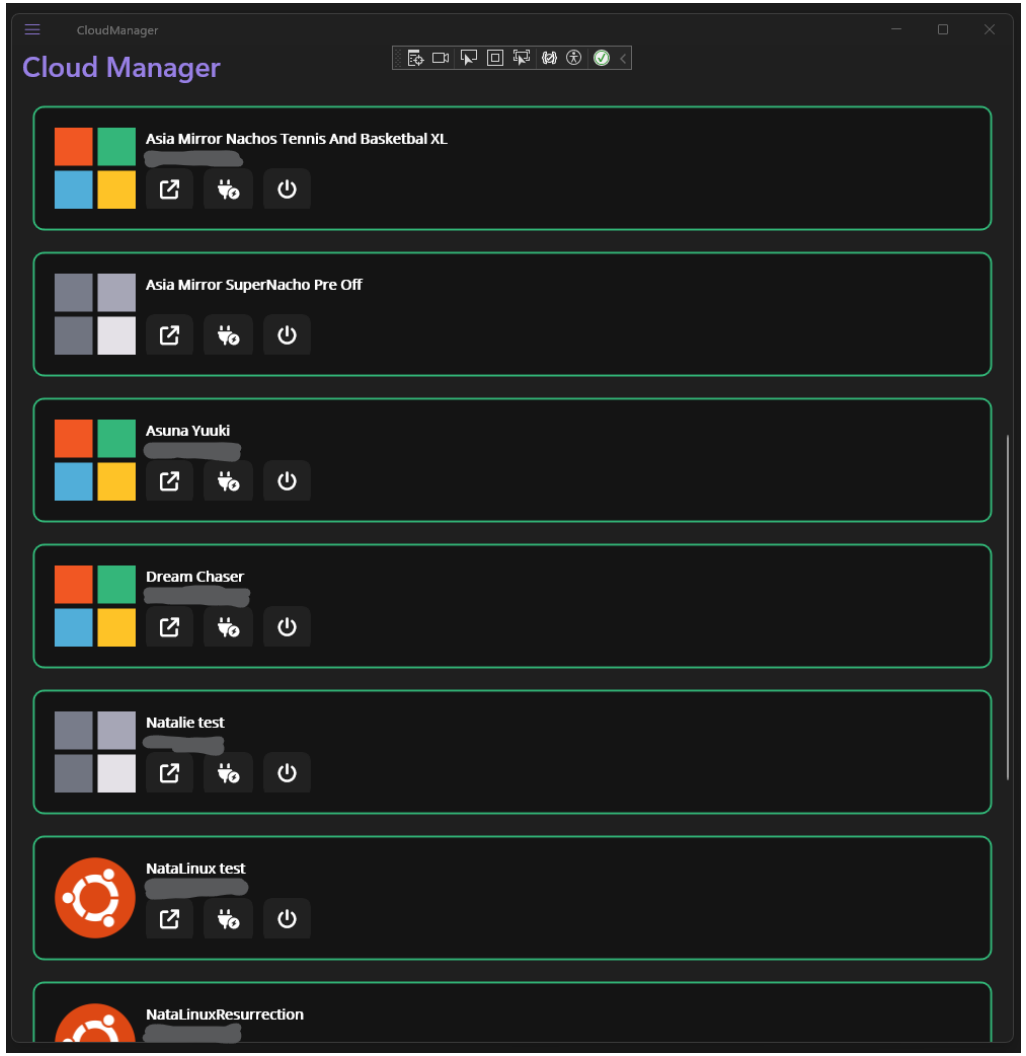
Dlaczego XAML jest taki cudowny?




```java
public class Okno {
    final JFrame okno = new JFrame( title: "Wykres funkcji");                                   //okno dla aplikac
    JLabel etykieta = new JLabel( text: "Wykres funkcji", JLabel.CENTER);                       //etykieta tekstow
    JPanel panelKontrolny = new JPanel();                                                       //panel kontrolny
    JLabel opis1 = new JLabel( text: " f(x) = ", JLabel.RIGHT);                                 //element panelu k
    JTextField poleTextFunkcji = new JTextField("2x^2+3x+1");                                   //element panelu k
    JButton przyciskRysuj = new JButton( text: "Rysuj");                                        //przycisk służący
    PanelWykresuFunkcji panelWykresuFunkcji = new PanelWykresuFunkcji(poleTextFunkcji.getText()); // Tworzy nowy Par
    Container wnetrzeOkna = okno.getContentPane();                                              //kontener okna c



    public Okno() {  1 usage
        okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);                                    //w momencie zamkniecia okna
        etykieta.setPreferredSize(new Dimension( width: 400, height: 25));                      //ustawia wymiar etykiety
        panelKontrolny.setLayout(new GridLayout( rows: 1, cols: 6));                            // Dodaje elementy do panelu k
        panelKontrolny.add(opis1);                                                              // *
        panelKontrolny.add(poleTextFunkcji);                                                    // *
        panelKontrolny.add(przyciskRysuj);                                                      // *
        przyciskRysuj.addActionListener(panelWykresuFunkcji);                                   //przypisuje przyciskowi rysu
        wnetrzeOkna.add(etykieta, BorderLayout.NORTH);                                          //ustawienie elementow wewnatr
        wnetrzeOkna.add(panelWykresuFunkcji, BorderLayout.CENTER);                              //środek okna
        wnetrzeOkna.add(panelKontrolny, BorderLayout.SOUTH);                                    //dół okna
        okno.setSize(new Dimension( width: ((Integer)Toolkit.getDefaultToolkit().getScreenSize().width)*4/5, height: ((Inte
        okno.setVisible(true);
        okno.setLocationRelativeTo(null);                                                       //domyślne wyświetlanie na src
    }
```

Java - Swing

Dlaczego XAML jest taki cudowny?



```
<!-- Definicja warstw -->
<CollectionView Grid.Row="0" ItemsSource="{Binding Servers}" VerticalOptions="FillAndExpand">
    <CollectionView.ItemTemplate>
        <DataTemplate x:DataType="cm:ServerListElement">
            <ContentView StyleClass="ServerListElement">
                <Border StyleClass="ServerListElement" Stroke="#32b778" StrokeThickness="2">
                    <Border.StrokeShape>
                        <RoundRectangle CornerRadius="10" />
                    </Border.StrokeShape>
                    <Grid StyleClass="ServerListElement">
                        <Grid.RowDefinitions>
                            <RowDefinition Height="20" />
                            <RowDefinition Height="20" />
                            <RowDefinition Height="40" />
                        </Grid.RowDefinitions>
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="100" />
                            <ColumnDefinition Width="*" />
                        </Grid.ColumnDefinitions>
                        <Image Grid.RowSpan="3" Source="{Binding PlatformIcon}" Aspect="AspectFi
                        <Label Grid.Column="1" Text="{Binding Name}" FontAttributes="Bold" />
                        <Label Grid.Row="1" Grid.Column="1" Text="{Binding IPAddress}" />
                        <HorizontalStackLayout Grid.Column="1" Grid.Row="2">
                            <Button Command="{Binding OpenRDPCommand}" Text="&#xf08e;" StyleClas
                            <Button Text="&#xe55b;" StyleClass="CodliServerListItemButton" /> <!
                            <Button Text="&#xf011;" StyleClass="CodliServerListItemButton" Margi
                        </HorizontalStackLayout>
                    </Grid>
                </Border>
            </ContentView>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

MAUI – oddzielność widoku i logiki

- W aplikacjach .NET MAUI:
  - XAML odpowiada za wygląd interfejsu,
  - C# odpowiada za logikę aplikacji.

```xaml
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             x:Class="MyApp.MainPage">

    <VerticalStackLayout Padding="20" Spacing="12">

        <Label
            Text="Kliknij przycisk"
            FontSize="20" />

        <Button
            Text="Kliknij mnie"
            Clicked="OnButtonClicked" />

    </VerticalStackLayout>

</ContentPage>
```

Kod XAML, opisujący UI

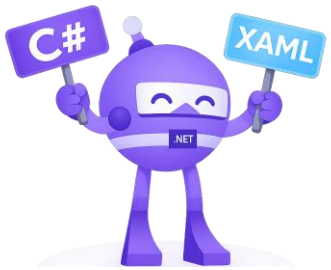
```csharp
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    private void OnButtonClicked(object sender, EventArgs e)
    {
        DisplayAlert("Info", "Przycisk został kliknięty!", "OK");
    }
}
```

Kod C#, opisujący logikę

XAML – centralny fundament UI

- XAML porządkuje aplikację, dzięki niemu:
  - UI jest opisane w jednym miejscu,
  - logika nie miesza się z UI,
  - projekt jest łatwiejszy do zrozumienia.

- W MAUI można tworzyć UI bez używania XAML – bezpośrednio w C# (*tak jak na przykładach z Javą i Pythonem*). Jednakże takie podejście nie jest zbyt wygodne, a tym bardziej czytelne.



XAML jest **podstawowym narzędziem** do tworzenia interfejsu użytkownika w aplikacjach .NET MAUI.

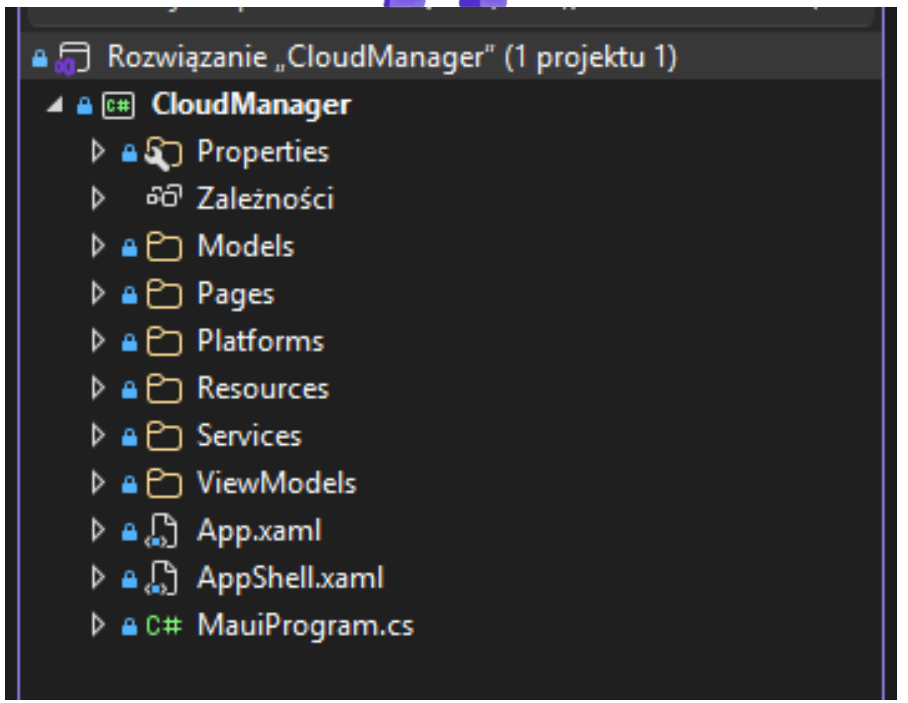
Struktura projektu

- Projekt .NET MAUI od początku jest przygotowany jako aplikacja wieloplatformowa. Już sama struktura projektu pokazuje, że:
  - jedna aplikacja,
  - jeden projekt,
  - wiele platform.
- Wszystko znajduje się w jednym rozwiązaniu, a nie w osobnych projektach dla Androida, Windowsa czy iOS

- Struktura projektu MAUI opiera się na prostym podziale:
  - pliki wspólne – używane na wszystkich platformach,
  - pliki platformowe – używane tylko tam, gdzie są potrzebne.

Dzięki temu:

- większość kodu piszemy tylko raz,
- różnice są jasno wydzielone,
- projekt pozostaje czytelny.

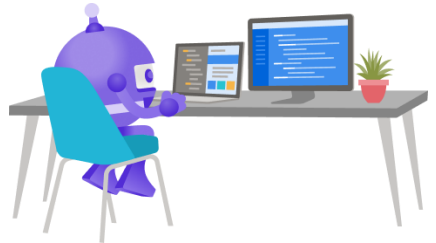


App.xaml

```xml
<?xml version = "1.0" encoding = "UTF-8" ?>
<Application xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:CloudManager"
             x:Class="CloudManager.App">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Resources/Styles/Colors.xaml" />
                <ResourceDictionary Source="Resources/Styles/Styles.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

- **App.xaml** to plik XAML, w którym zwykle trzymamy zasoby wspólne dla całej aplikacji. Czyli rzeczy, które mają wpływ na wygląd i styl w wielu miejscach naraz, np.:
  - kolory
  - style kontrolek (*np. wygląd przycisków*)
  - czcionki, rozmiary, marginesy
  - zasoby, które chcemy używać w wielu ekranach

Można o nim myśleć jak o „bazie ustawień wyglądu” dla całej aplikacji.



App.xaml.cs

- **App.xaml.cs** to plik C#, który odpowiada za uruchomienie aplikacji i wskazanie, co ma się pokazać jako pierwsze.

- Najczęściej to właśnie tutaj ustawiamy:
  - pierwszą stronę aplikacji (MainPage)
  - ewentualnie nawigację (np. AppShell), jeśli projekt jej używa

```
namespace CloudManager
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();
        }

        protected override Window CreateWindow(IActivationState? activationState)
        {
            return new Window(new AppShell());
        }
    }
}
```

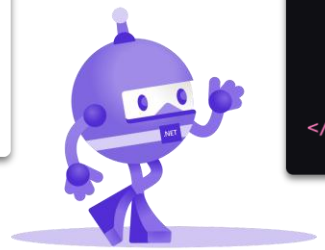
`InitializeComponent()` ładuje to, co jest w App.xaml (zasoby, style itp.), a potem ustawiamy pierwszy ekran.

Widoki – MainPage.xaml i MainPage.xaml.cs

- MainPage to pierwszy ekran, który użytkownik widzi po uruchomieniu aplikacji. To właśnie ten ekran został ustawiony wcześniej w App.xaml.cs jako MainPage.

- **MainPage.xaml** – opisuje wygląd ekranu, definiuje układ kontrolek.
- **MainPage.xaml.cs** – zawiera logikę powiązaną z ekranem, obsługuje zdarzenia, reaguje na działania użytkownika.

```
namespace MyApp;

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    private void OnButtonClicked(object sender, EventArgs e)
    {
        DisplayAlert("Info", "Przycisk został kliknięty!", "OK");
    }
}
```



```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             x:Class="MyApp.MainPage">

    <VerticalStackLayout Padding="20" Spacing="12">

        <Label
            Text="Witaj w .NET MAUI!"
            FontSize="24"
            HorizontalOptions="Center" />

        <Button
            Text="Kliknij mnie"
            Clicked="OnButtonClicked" />

    </VerticalStackLayout>

</ContentPage>
```

MauiProgram.cs

- **MauiProgram.cs** to miejsce, w którym aplikacja mówi:
  - jak ma się uruchomić,
  - jakie funkcje i usługi mają być dostępne,
  - jakie zasoby (np. czcionki) dodajemy,
  - jakie „dodatki” włączamy.

To jest taki punkt konfiguracji całej aplikacji – zanim pokaże się pierwszy ekran.

```
namespace MyApp;

public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();

        builder
            .UseMauiApp<App>()
            .ConfigureFonts(fonts =>
            {
                fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
            });

        return builder.Build();
    }
}
```

Katalog Platforms

- **Platforms** to miejsce w projekcie .NET MAUI, w którym znajdują się pliki specyficzne dla konkretnych systemów operacyjnych. To właśnie tutaj MAUI „schodzi na poziom systemu”.

Każda platforma ma swój własny podkatalog, np.:

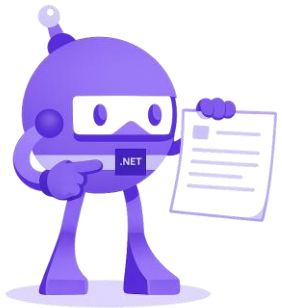
- Android
- iOS
- macOS
- Windows

- Większość aplikacji jest wspólna, więc katalog Platforms ruszamy, tylko jeśli jest taka potrzeba.

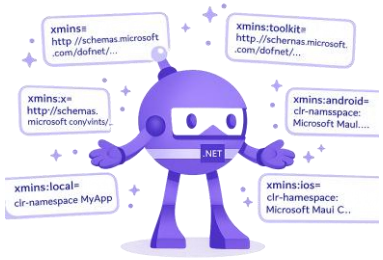
ContentPage

- Każdy plik XAML w MAUI zwykle opisuje jedną stronę aplikacji (*są wyjątki – style, własne kontrolki czy komponenty*). Stroną nazywamy jeden ekran, który widzi użytkownik – jest to **<ContentPage/>**, kontener najwyższego poziomu. Wewnątrz tego kontenera, może znajdować się wyłącznie jeden element.

```
<ContentPage>
    <VerticalStackLayout>
        <!-- zawartość strony -->
    </VerticalStackLayout>
</ContentPage>
```



Przestrzenie nazw w XAML



- W XAML podobnie jak w C#, istnieją przestrzenie nazw. **xmlns** (*XML namespace*) określa, skąd pochodzą elementy używane w XAML. Dzięki temu XAML wie, czym jest np. Button, Label czy Grid.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MyApp.MainPage">
    <VerticalStackLayout>
        <!-- Zawartość strony -->
    </VerticalStackLayout>
</ContentPage>
```

Główna przestrzeń nazw MAUI

xmlns="http://schemas.microsoft.com/dotnet/2021/maui"

Zawiera kontrolki i layouty MAUI

Główna przestrzeń nazw XAML

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

Zawiera elementy techniczne, np. `x:Name`

Powiązanie XAML z C#

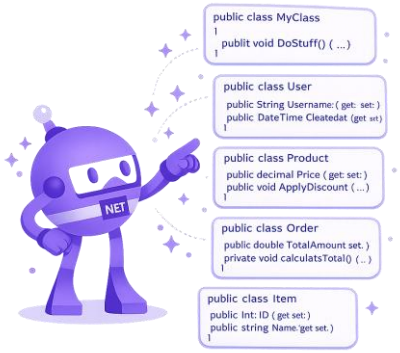
- Do powiązania kodu XAML z klasą C# używamy atrybutu x:Class. Określa on, z jaką klasą C# powiązany jest dany plik XAML.

```xaml
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MyApp.MainPage">
    <VerticalStackLayout>
        <!-- Zawartość strony -->
    </VerticalStackLayout>
</ContentPage>
```

Kod XAML strony MainPage

```csharp
namespace MyApp
{
    public partial class MainPage : ContentPage
    {
        public MainPage()
        {
            InitializeComponent();
        }
    }
}
```

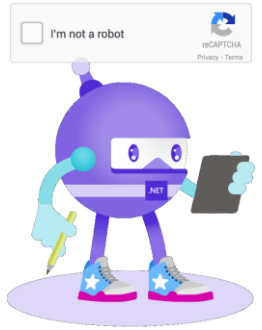
Kod C# strony MainPage



Layouty

- Layout to element, który układa kontrolki na ekranie. Kontrolki same z siebie (*Label, Button, Entry*) nie wiedzą, *gdzie mają być* – to layout decyduje o:
  - kolejności elementów,
  - odstępach,
  - wierszach/kolumnach,
  - dopasowaniu do rozmiaru ekranu.

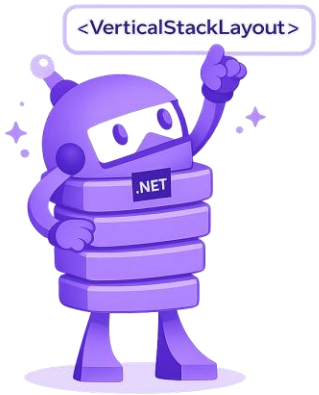
Layout to organizator kontrolek na ekranie.



VerticalStackLayout

Używasz, gdy elementy mają iść pionowo, jeden po drugim – formularze, listy przycisków, proste ekrany.

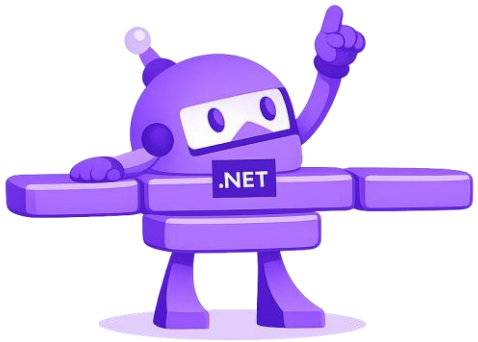
```
<VerticalStackLayout Padding="20" Spacing="10">

    <Label Text="Rejestracja" FontSize="24" />

    <Entry Placeholder="Imię" />
    <Entry Placeholder="Email" />
    <Entry Placeholder="Hasło" IsPassword="True" />

    <Button Text="Utwórz konto" />

</VerticalStackLayout>
```



HorizontalStackLayout

Używasz, gdy elementy mają być w jednym wierszu, np. ikona + tekst, przyciski obok siebie – przykładowo: paski narzędzi, małe grupy elementów.

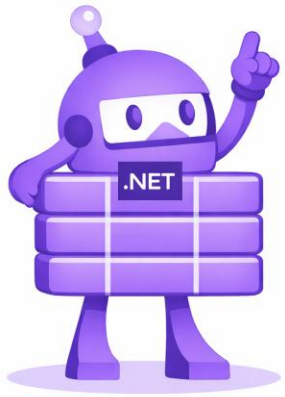
```
<HorizontalStackLayout Padding="20" Spacing="10">

    <Button Text="-" WidthRequest="50" />
    <Label Text="1" FontSize="20" VerticalOptions="Center" />
    <Button Text="+" WidthRequest="50" />

</HorizontalStackLayout>
```



Grid

Używasz, gdy potrzebujesz precyzyjnego układu: wiersze/kolumny, np. ekran logowania, karta produktu, panel z danymi.

```
<Grid Padding="20"
      RowDefinitions="Auto,Auto,Auto"
      ColumnDefinitions="Auto,*"
      RowSpacing="10"
      ColumnSpacing="10">

    <Label Text="Login:" Grid.Row="0" Grid.Column="0" />
    <Entry Placeholder="Wpisz login" Grid.Row="0" Grid.Column="1" />

    <Label Text="Hasło:" Grid.Row="1" Grid.Column="0" />
    <Entry Placeholder="Wpisz hasło" IsPassword="True" Grid.Row="1" Grid.Column="1" />

    <Button Text="Zaloguj się" Grid.Row="2" Grid.ColumnSpan="2" />
</Grid>
```

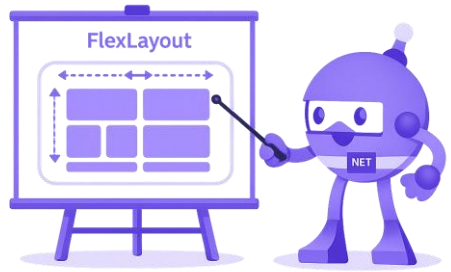


FlexLayout

FlexLayout to layout inspirowany Flexboxem z CSS. Pozwala: dynamicznie układać elementy, zawijać je do kolejnych „wierszy”, dopasowywać się do rozmiaru ekranu.

Idealny, gdy: nie znasz liczby elementów, układ ma być „responsywny”, elementy mają się automatycznie przesuwać.

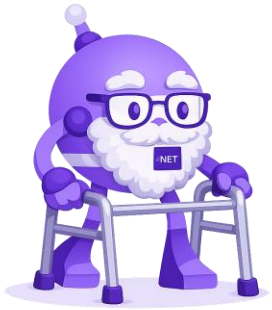
```
<FlexLayout
    Direction="Row"
    Wrap="Wrap"
    JustifyContent="SpaceAround"
    Padding="20">

    <Button Text="A" />
    <Button Text="B" />
    <Button Text="C" />
    <Button Text="D" />
    <Button Text="E" />

</FlexLayout>

<!-- Elementy są układane w jednym wierszu, jeżeli się nie mieszczą, zwijają się -->
```



StackLayout

StackLayout to starszy, klasyczny layout, który układa elementy: pionowo lub poziomo. Działa bardzo podobnie do VerticalStackLayout i HorizontalStackLayout, ale jest mniej wydajny oraz ma mniej optymalną obsługę układu.

```
<StackLayout Padding="20" Spacing="10">

    <Label Text="Stary StackLayout" FontSize="20" />

    <Button Text="Przycisk 1" />
    <Button Text="Przycisk 2" />
    <Button Text="Przycisk 3" />

</StackLayout>
```

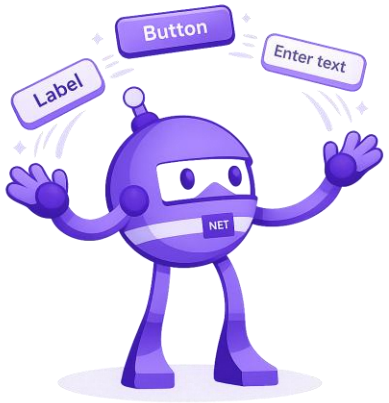


W nowych aplikacjach MAUI, lepiej używać Horizontal i Vertical.

Kontrolki

- .NET MAUI dostarcza wspólny zestaw kontrolek, które:
  - działają na wszystkich wspieranych platformach,
  - są renderowane jako natywne elementy systemowe,
  - mają podobne zachowanie niezależnie od platformy.

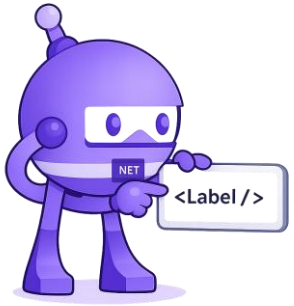
- Każda kontrolka:
  - ma właściwości (np. tekst, kolor, rozmiar),
  - może reagować na zdarzenia (np. kliknięcie),
  - może być umieszczona w layoutach.



Label

Label służy do pokazywania informacji użytkownikowi. Nie służy do wprowadzania danych – tylko do ich prezentacji.

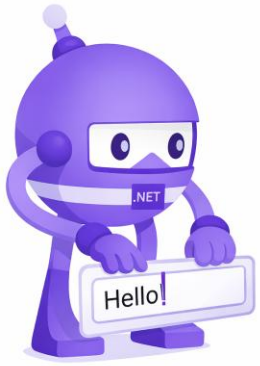
```
<Label
    Text="Witaj w .NET MAUI!"
    FontSize="24"
    HorizontalOptions="Center" />
```



Entry

Entry to pole tekstowe, w którym użytkownik może wpisać dane. Może służyć do: loginów, haseł, krótkich danych tekstowych.

```
<Entry
    Placeholder="Wpisz swoje imię" />
```



Button

- Button pozwala użytkownikowi wykonać akcję, np.:
  - wysłać formularz,
  - zatwierdzić wybór,
  - przejść dalej.

Image

- Image służy do wyświetlania obrazów w aplikacji. Może wyświetlać obrazy z zasobów lokalnych (*pliki dodane do projektu*), z adresu URL oraz generowane w kodzie.
- Kontrolka Image posiada właściwość Aspect, która określa w jaki sposób obraz dostosowuje się do dostępnej przestrzeni *(podobnie jak object-fit w CSS):*
  - **AspectFit** – cały obraz widoczny, może zostać puste miejsce (domyślne).
  - **AspectFill** – wypełnia całą przestrzeń, obraz może zostać przycięty.
  - **Fill** – rozciąga obraz, może zniekształcić proporcje.

```
<Image
    Source="dotnet_bot.png"
    HeightRequest="200"
    Aspect="AspectFit" />
```

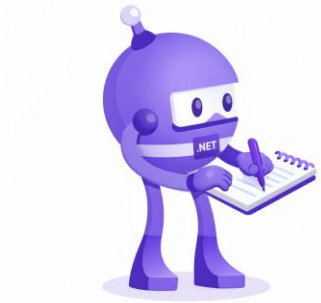
Obrazki lokalne najczęściej umieszczamy w katalogu **Resources/Images**, wtedy są automatycznie dostępne na wszystkich platformach.

Editor

- Editor to wieloliniowe pole tekstowe – w odróżnieniu od Entry, pozwala użytkownikowi wpisać dłuższy tekst, np. komentarz, opis, notatkę.
- Przydatne właściwości:
  - **Placeholder** – tekst podpowiedzi, wyświetlany gdy pole jest puste.
  - **MaxLength** – maksymalna liczba znaków, jaką może wpisać użytkownik.
  - **IsReadOnly** – jeżeli true, użytkownik może jedynie odczytywać tekst.
  - **AutoSize** – kontrolka może automatycznie się rozszerzać, wraz z wpisywanym tekstem.

```
<Editor
    Placeholder="Wpisz swój komentarz..."
    HeightRequest="150"
    MaxLength="500" />
```

Entry – jedna linia tekstu, do krótkich danych (*login, e-mail, hasło*)
Editor – wiele linii, do dłuższych treści (*opis, wiadomość, notatka*)

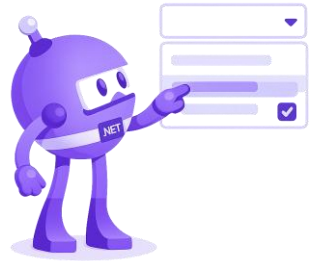


Picker

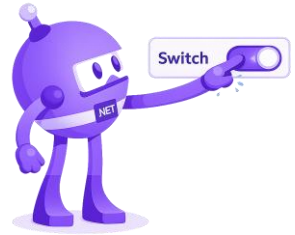
- **Picker** to lista rozwijana, z której użytkownik wybiera jedną opcję. Przydatny, gdy mamy z góry określony zbiór wartości do wyboru – np. kategoria, miasto, rola użytkownika.
- Przydatne właściwości:
  - **Title** – tekst wyświetlany, gdy nic nie jest wybrane (*podpowiedź*).
  - **ItemsSource** – źródło danych (*lista elementów do wyboru*).
  - **SelectedItem** – aktualnie wybrany element.
  - **SelectedIndex** – indeks wybranego elementu (*licząc od 0*).

```
<Picker Title="Wybierz język programowania">
    <Picker.ItemsSource>
        <x:Array Type="{x:Type x:String}">
            <x:String>C#</x:String>
            <x:String>Java</x:String>
            <x:String>Python</x:String>
        </x:Array>
    </Picker.ItemsSource>
</Picker>
```

Uwaga!
W prawdziwych aplikacjach **ItemsSource najczęściej ustawiamy z poziomu C#** – np. przypisujemy listę obiektów. Wpisywanie elementów bezpośrednio w XAML sprawdza się tylko przy małych, stałych zbiorach.



Switch



- Switch to przełącznik, który posiada dwa stany: włączony (true) lub wyłączony (false). Używamy go, gdy użytkownik ma podjąć decyzję typu tak/nie, np. włączenie powiadomień, tryb ciemny, zgoda na regulamin.
- Przydatne właściwości:
  - **IsToggled** – aktualny stan przełącznika (true lub false).
  - **OnColor** – kolor tła, gdy Switch jest włączony.
  - **ThumbColor** – kolor "kółka" przełącznika.
- Switch posiada zdarzenie **Toggled**, wywoływane za każdym razem, gdy użytkownik zmieni stan przełącznika. W metodzie obsługującej zdarzenie możemy sprawdzić nowy stan.

```
<HorizontalStackLayout Spacing="10">
    <Label
        Text="Tryb ciemny"
        VerticalOptions="Center" />
    <Switch
        IsToggled="False"
        OnColor="Green"
        ThumbColor="White" />
</HorizontalStackLayout>
```

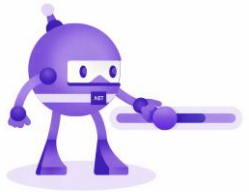
Uwaga!
Switch prawie zawsze umieszczamy obok Label, który opisuje co ten przełącznik robi – sam Switch nie wyświetla żadnego tekstu.

CheckBox

- CheckBox to pole wyboru – użytkownik może je zaznaczyć lub odznaczyć. W odróżnieniu od Switch, CheckBox stosujemy tam, gdzie użytkownik zaznacza jedną lub więcej opcji z listy – np. wybór zainteresowań, akceptacja warunków, lista zakupów.
- Przydatne właściwości:
  - **IsChecked** – czy pole jest zaznaczone (true lub false).
  - **Color** – kolor znacznika po zaznaczeniu.
- CheckBox posiada zdarzenie **CheckedChanged**, które jest wywoływane, gdy użytkownik zaznaczy lub odznaczy pole.

```
<VerticalStackLayout Spacing="10">
    <HorizontalStackLayout Spacing="8">
        <CheckBox IsChecked="True" Color="Blue" />
        <Label Text="Akceptuję regulamin"
               VerticalOptions="Center" />
    </HorizontalStackLayout>

    <HorizontalStackLayout Spacing="8">
        <CheckBox Color="Blue" />
        <Label Text="Chcę otrzymywać newsletter"
               VerticalOptions="Center" />
    </HorizontalStackLayout>
</VerticalStackLayout>
```

Slider



- Slider pozwala użytkownikowi wybrać wartość liczbową z określonego zakresu, przesuwając suwak. Przydatny wszędzie tam, gdzie użytkownik ustawia poziom czegoś – np. głośność, jasność, rozmiar czcionki, cena w filtrze.
- Przydatne właściwości:
  - **Minimum** – dolna granica zakresu.
  - **Maximum** – górna granica zakresu.
  - **Value** – aktualna wartość suwaka.
  - **MinimumTrackColor** – kolor paska po lewej stronie suwaka (już "przesunięta" część).
  - **MaximumTrackColor** – kolor paska po prawej stronie suwaka.
  - **ThumbColor** – kolor kółka suwaka.
- Slider posiada zdarzenie **ValueChanged**, wywoływane przy każdym przesunięciu suwaka.

```
<Slider
    x:Name="suwak"
    Minimum="0"
    Maximum="100"
    Value="50"
    ValueChanged="OnSliderValueChanged" />
<Label x:Name="wartoscLabel" FontSize="20" />
```

ActivityIndicator

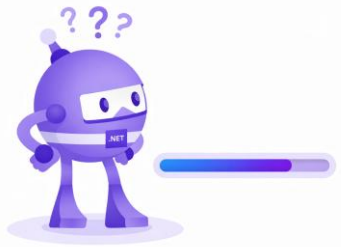
- ActivityIndicator to animowany wskaźnik ładowania – kręcące się "kółko", które informuje użytkownika, że aplikacja coś robi w tle. Nie pokazuje postępu (*nie wiemy ile procent gotowe*) – tylko sygnalizuje, że trwa jakieś działanie, np. pobieranie danych z serwera, logowanie, ładowanie listy.
- Przydatne właściwości:
  - **IsRunning** – czy wskaźnik jest aktywny (true = animacja się kręci, false = ukryty).
  - **Color** – kolor animacji.

```
<VerticalStackLayout Padding="20" Spacing="10">
    <ActivityIndicator
        IsRunning="True"
        Color="Blue" />
    <Label
        Text="Ładowanie danych..."
        HorizontalOptions="Center" />
</VerticalStackLayout>
```

```
loader.IsRunning = true;
// ... operacja w tle ...
loader.IsRunning = false;
```

Uwaga!
ActivityIndicator najczęściej sterujemy z poziomu C# – włączamy go przed rozpoczęciem operacji i wyłączamy po jej zakończeniu.

ProgressBar



- ProgressBar to pasek postępu – pokazuje użytkownikowi, postęp w realizacji danego zadania. W odróżnieniu od ActivityIndicator, tutaj znamy konkretny stopień ukończenia. Stosujemy go np. przy pobieraniu pliku, przesyłaniu danych, wypełnianiu formularza wieloetapowego.
- Przydatne właściwości:
  - **Progress** – aktualna wartość postępu, zawsze w zakresie od 0.0 (0%) do 1.0 (100%).
  - **ProgressColor** – kolor wypełnionej części paska.

```
<VerticalStackLayout Padding="20" Spacing="10">
    <ProgressBar
        Progress="0.7"
        ProgressColor="Green" />
    <Label
        Text="Postęp: 70%"
        HorizontalOptions="Center" />
</VerticalStackLayout>
```

```
// Wartość ustawiamy ręcznie w kodzie, np. po każdym kroku operacji.
pasekPostepu.Progress = 0.25; // 25%
pasekPostepu.Progress = 0.5;  // 50%
pasekPostepu.Progress = 1.0;  // 100%

// Można też użyć animowanej (asynchronicznej) zmiany wartości:
await pasekPostepu.ProgressTo(0.75, 500, Easing.Linear);
```

DatePicker

- DatePicker pozwala użytkownikowi wybrać datę z kalendarza systemowego. Po kliknięciu na kontrolkę, system wyświetla natywny selektor daty – *wygląda on inaczej na każdej platformie*, ale działa tak samo. Stosujemy go np. przy wyborze daty urodzenia, terminu rezerwacji, daty wydarzenia.
- Przydatne właściwości:
  - **Date** – aktualnie wybrana data.
  - **MinimumDate** – najwcześniejsza data, którą można wybrać.
  - **MaximumDate** – najpóźniejsza data, którą można wybrać.
  - **Format** – sposób wyświetlania daty (*np. "dd.MM.yyyy" da „29.05.2021"*).
- Kontrolka posiada zdarzenie **DateSelected**, wywoływane, gdy użytkownik zmieni datę.


```
<DatePicker
    Format="dd.MM.yyyy"
    MinimumDate="01/01/2000"
    MaximumDate="12/31/2030"
    DateSelected="OnDateSelected" />
```

TimePicker

- TimePicker pozwala użytkownikowi wybrać godzinę. Podobnie jak DatePicker, po kliknięciu wyświetla natywny selektor czasu. Stosujemy go np. przy ustawianiu alarmu, godziny spotkania, czasu rezerwacji.
- Przydatne właściwości:
  - **Time** – aktualnie wybrana godzina (*typ TimeSpan, nie DateTime*).
  - **Format** – sposób wyświetlania godziny (*np. "HH:mm" da "08:30", "hh:mm tt" da "08:30 AM"*).

```
<TimePicker
    Time="08:30:00"
    Format="HH:mm" />
```

```
TimeSpan wybranaGodzina = zegarek.Time;
int godziny = wybranaGodzina.Hours;
int minuty = wybranaGodzina.Minutes;
```

Uwaga!
TimePicker nie ma zdarzenia analogicznego do DateSelected. Zmianę godziny najczęściej odczytujemy w momencie zatwierdzenia formularza (np. po kliknięciu przycisku), albo **obsługujemy przez binding do właściwości Time**.

SearchBar

- SearchBar to pole wyszukiwania z wbudowaną ikoną lupy. Znajdziemy go w większości aplikacji – służy do filtrowania list, wyszukiwania produktów, kontaktów, wiadomości. Wygląda jak Entry, ale ma dodatkowe elementy: ikonę wyszukiwania, przycisk anulowania oraz dedykowane zdarzenia.
- Przydatne właściwości:
  - **Text** – aktualnie wpisany tekst.
  - **Placeholder** – tekst podpowiedzi, wyświetlany gdy pole jest puste.
  - **CancelButtonColor** – kolor przycisku anulowania (X).
- Zdarzenia:
  - **SearchButtonPressed** – wywoływane, gdy użytkownik kliknie przycisk szukaj na klawiaturze. Wyszukiwanie uruchamia się dopiero po zatwierdzeniu.
  - **TextChanged** – wywoływane przy każdej zmianie tekstu, litera po literze. Pozwala na filtrowanie w czasie rzeczywistym.

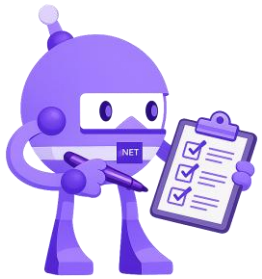
```
<SearchBar
    Placeholder="Szukaj produktu..."
    SearchButtonPressed="OnSearchPressed"
    TextChanged="OnSearchTextChanged" />
```

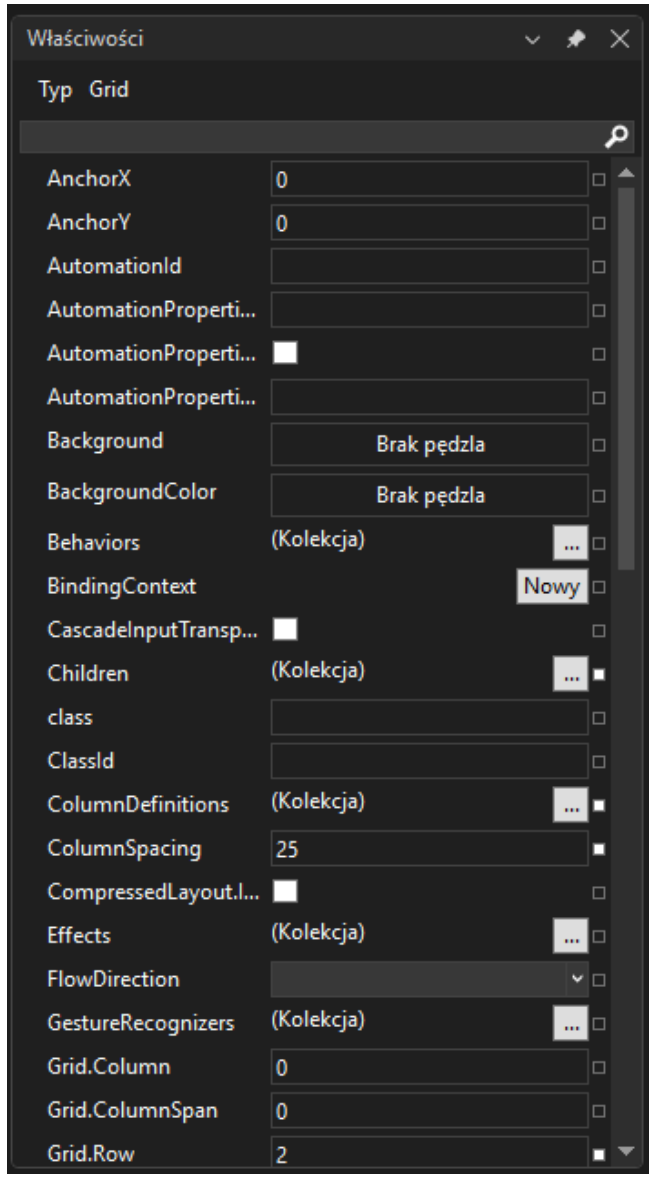
```
private void OnSearchPressed(object sender, EventArgs e)
{
    var searchBar = (SearchBar)sender;
    string szukanyTekst = searchBar.Text;
    // filtrowanie listy...
}

private void OnSearchTextChanged(object sender,
TextChangedEventArgs e)
{
    string nowyTekst = e.NewTextValue;
    // filtrowanie listy na bieżąco...
}
```

Uwaga!
SearchButtonPressed to wyszukiwanie "na żądanie" – użytkownik wpisuje frazę i zatwierdza.
TextChanged to filtrowanie na żywo (*podpowiedzi*) – wyniki zmieniają się z każdą literą.

Wspólne właściwości kontrolek

- **Większość kontrolek w MAUI dziedziczy po klasie View**, dzięki czemu wiele właściwości jest wspólnych i działa tak samo, niezależnie z jaką kontrolką pracujemy.





Margin i Padding

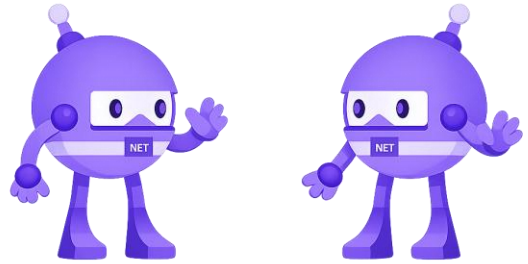
- Margin i Padding to dwie właściwości kontrolujące odstępy – wyglądają podobnie, ale działają w przeciwnych kierunkach:
  - **Margin** to odstęp na zewnątrz kontrolki – odsuwa ją od sąsiednich elementów.
  - **Padding** to odstęp wewnątrz kontrolki – odsuwa zawartość od krawędzi kontrolki.

```
<Button
    Text="Kliknij mnie"
    Margin="20"
    Padding="15" />
```

```
<Button
    Text="Kliknij mnie"
    Margin="10,20"
    Padding="5,15" />
```

```
<Button
    Text="Kliknij mnie"
    Margin="5,10,15,20"
    Padding="5,10,15,20" />
```

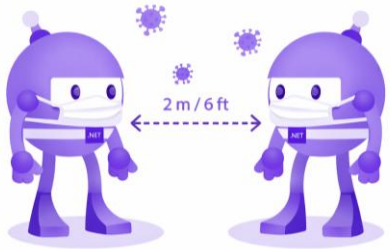
Uwaga!
Padding mają tylko te kontrolki, które mogą zawierać inne elementy – np. Button (*zawiera tekst*), Layout (*zawiera kontrolki*), Frame. Nie ustawimy Padding na Slider czy CheckBox, bo nie mają "zawartości wewnętrznej". Margin natomiast ma każda kontrolka.



Spacing

- Spacing to właściwość layoutów (nie pojedynczych kontrolek), która określa odstęp między elementami potomnymi. Zamiast ustawiać Margin na każdej kontrolce z osobna, wystarczy jeden Spacing w layoucie – i wszystkie elementy wewnątrz będą równomiernie oddalone od siebie.
  - Spacing działa z VerticalStackLayout, HorizontalStackLayout i StackLayout.
  - Grid zamiast Spacing używa RowSpacing i ColumnSpacing, które działają analogicznie.
  - FlexLayout w ogóle nie używa Spacing.

```
<VerticalStackLayout Spacing="15">
    <Button Text="Pierwszy" />
    <Button Text="Drugi" />
    <Button Text="Trzeci" />
</VerticalStackLayout>
```



HorizontalOptions i VerticalOptions

- Każda kontrolka zajmuje pewne miejsce w layoucie. HorizontalOptions i VerticalOptions określają, jak kontrolka zachowuje się w dostępnej przestrzeni – gdzie się wyrównuje i czy ją wypełnia.
- Dostępne wartości:
  - **Start** – kontrolka przylega do lewej krawędzi (Horizontal) lub górnej krawędzi (Vertical).
  - **Center** – kontrolka ustawia się na środku.
  - **End** – kontrolka przylega do prawej krawędzi (Horizontal) lub dolnej krawędzi (Vertical).
  - **Fill** – kontrolka rozciąga się, żeby wypełnić całą dostępną przestrzeń (wartość domyślna).

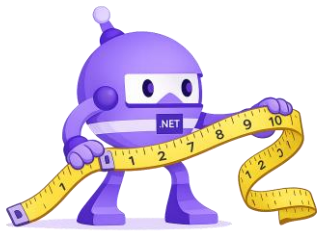
```
<VerticalStackLayout Padding="20" Spacing="10">
    <Button Text="Start"
            HorizontalOptions="Start"
            BackgroundColor="Blue" />
    <Button Text="Center"
            HorizontalOptions="Center"
            BackgroundColor="Green" />
    <Button Text="End"
            HorizontalOptions="End"
            BackgroundColor="Orange" />
    <Button Text="Fill"
            HorizontalOptions="Fill"
            BackgroundColor="Red" />
</VerticalStackLayout>
```

W tym przykładzie wszystkie cztery przyciski są w VerticalStackLayout – układ ustawia je jeden pod drugim, a **HorizontalOptions decyduje o tym, gdzie każdy z nich ląduje w poziomie**. Pierwszy przylega do lewej, drugi jest na środku, trzeci po prawej, a czwarty rozciąga się na całą szerokość.

Uwaga!
Działanie tych właściwości zależy od rodzaju layoutu nadrzędnego. **W VerticalStackLayout kontrolki układają się pionowo – więc HorizontalOptions wpływa na ich pozycję**, ale **VerticalOptions nie ma efektu** (bo każda kontrolka dostaje dokładnie tyle miejsca w pionie, ile potrzebuje). W Grid natomiast obie właściwości działają, bo komórka ma określony rozmiar w obu kierunkach.

WidthRequest i HeightRequest

- **WidthRequest** i **HeightRequest** pozwalają określić żądany rozmiar kontrolki – odpowiednio szerokość i wysokość, wyrażone w jednostkach niezależnych od urządzenia (*device-independent pixels*). Słowo "Request" (żądanie) nie jest przypadkowe – **to sugestia, a nie gwarancja**. Layout nadrzędny **może przydzielić kontrolce inny rozmiar, jeśli nie ma wystarczająco dużo miejsca**.
- Tych właściwości **nie** używamy gdy chcemy, żeby interfejs elastycznie dopasowywał się do różnych rozmiarów ekranów. Sztywne rozmiary mogą wyglądać dobrze na jednym urządzeniu, ale źle na innym. W takich przypadkach lepiej polegać na HorizontalOptions="Fill" i właściwościach layoutu (np. proporcjach w Grid).
- Width i Height to wartości tylko do odczytu – to rzeczywisty rozmiar kontrolki po tym, jak layout ją rozmieścił. Nie możemy ich ustawić w XAML.

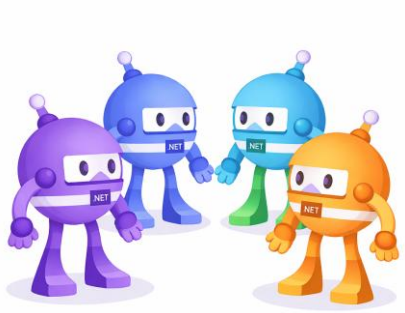


```
<VerticalStackLayout Padding="20" Spacing="10">
    <BoxView
        Color="Blue"
        WidthRequest="200"
        HeightRequest="100"
        HorizontalOptions="Center" />
    <Button
        Text="Mały przycisk"
        WidthRequest="150"
        HeightRequest="50"
        HorizontalOptions="Center" />
</VerticalStackLayout>
```

Właściwości wizualne

- Większość kontrolek wyświetlających tekst lub mających tło udostępnia zestaw właściwości wizualnych. Pozwalają one szybko zmienić wygląd elementu bez konieczności definiowania stylów.
  - **BackgroundColor** – kolor tła kontrolki. Przyjmuje nazwy kolorów (*Red, Blue*) lub wartości heksadecymalne (*#FF5733*).
  - **TextColor** – kolor tekstu. Działa na kontrolkach wyświetlających tekst (*Label, Button, Entry, Editor, SearchBar itd.*).
  - **FontSize** – rozmiar czcionki, wyrażony w jednostkach niezależnych od urządzenia.
  - **FontAttributes** – styl czcionki. Trzy możliwe wartości:
    - **None** (domyślny),
    - **Bold** (pogrubienie),
    - **Italic** (kursywa).

    Można łączyć: FontAttributes="Bold,Italic".

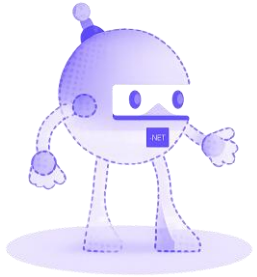


```
<VerticalStackLayout Padding="20" Spacing="10">
    <Label
        Text="Nagłówek strony"
        FontSize="28"
        FontAttributes="Bold"
        TextColor="DarkBlue" />
    <Button
        Text="Zatwierdź"
        FontSize="18"
        FontAttributes="Bold"
        TextColor="White"
        BackgroundColor="Green" />
</VerticalStackLayout>
```

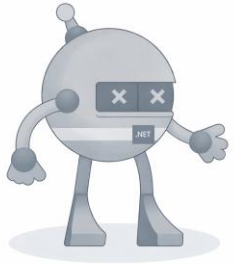
IsVisible i IsEnabled

- Te dwie właściwości kontrolują dostępność elementu w interfejsie – ale robią to na różne sposoby.
  - **IsVisible** decyduje, czy kontrolka w ogóle pojawia się na ekranie. Gdy ustawimy IsVisible=„false", kontrolka znika z interfejsu – nie jest widoczna i nie zajmuje miejsca w layoucie. Pozostałe kontrolki przesuwają się, jakby tej kontrolki w ogóle nie było.
  - **IsEnabled** decyduje, czy użytkownik może z nią wchodzić w interakcję. Gdy ustawimy IsEnabled="False", kontrolka jest widoczna, ale wyszarzona i nie reaguje na interakcje użytkownika. Przycisk nie da się kliknąć, pole tekstowe nie przyjmuje tekstu, suwak się nie przesuwa.

```
<Label
    Text="Ta wiadomość jest ukryta"
    IsVisible="False" />
```



```
<Button
    Text="Wyślij"
    IsEnabled="False" />
```



Opacity

- Opacity kontroluje przezroczystość kontrolki. Przyjmuje wartości od 0.0 (*całkowicie przezroczysta, niewidoczna*) do 1.0 (*całkowicie nieprzezroczysta, domyślna*). W odróżnieniu od IsVisible="False", kontrolka z Opacity="0" nadal zajmuje miejsce w layoucie i nadal reaguje na interakcje użytkownika.

```
<VerticalStackLayout Padding="20" Spacing="10">
    <Label Text="Opacity 1.0 (domyślna)" Opacity="1.0" />
    <Label Text="Opacity 0.7" Opacity="0.7" />
    <Label Text="Opacity 0.4" Opacity="0.4" />
    <Label Text="Opacity 0.1" Opacity="0.1" />
</VerticalStackLayout>
```

```
// Opacity można swobodnie animować
await etykieta.FadeTo(0, 500);   // zniknięcie w 500ms
await etykieta.FadeTo(1, 500);   // pojawienie się w 500ms
```

Atrybut x:Name

- x:Name pozwala nadać kontrolce unikalną nazwę, dzięki której możemy się do niej odwołać z poziomu kodu C#. Bez x:Name kontrolka istnieje w interfejsie, ale nasz kod nie ma do niej dostępu – nie możemy odczytać jej wartości, zmienić tekstu ani zareagować na jej stan.
- Zasady nazewnictwa:
  - Nazwa musi być unikalna w obrębie jednej strony – dwie kontrolki nie mogą mieć tego samego x:Name.
  - **Stosujemy konwencję camelCase**.

```
<VerticalStackLayout Padding="20" Spacing="15">
    <Entry
        x:Name="poleImie"
        Placeholder="Wpisz swoje imię" />
    <Button
        Text="Przywitaj się"
        Clicked="OnPrzywitaj" />
    <Label
        x:Name="etykietaPowitanie"
        FontSize="20" />
</VerticalStackLayout>
```

```
private void OnPrzywitaj(object sender, EventArgs e)
{
    string imie = poleImie.Text;
    etykietaPowitanie.Text = $"Cześć, {imie}!";
}
```

Data binding – wiązanie logiki i interfejsu

- **Data binding** to mechanizm, który **łączy dane z interfejsem użytkownika**.
  Dzięki niemu:
  - interfejs **sam aktualizuje się**, gdy zmienią się dane,
  - nie trzeba ręcznie przepisywać wartości między kontrolkami.

- Bez bindingu musielibyśmy:
  - obsługiwać każde zdarzenie ręcznie,
  - pobierać tekst z jednej kontrolki,
  - ustawiać go w innej kontrolce w kodzie.

Data binding

Przykład – chcemy aby po wpisaniu tekstu w pole Entry, został on automatycznie wyświetlony w Label.

Co się tutaj dzieje?

- Entry - zapisuje tekst do UserName.
- Label – wyświetla aktualną wartość UserName.

zmiana w jednym miejscu to **automatyczna zmiana w drugim**.

Nie piszemy:

- żadnych eventów,
- żadnych OnTextChanged,
- żadnego ręcznego przypisywania.

```
<VerticalStackLayout Padding="20" Spacing="12">
    <Entry
        Placeholder="Wpisz imię"
        Text="{Binding UserName}" />
    <Label
        Text="{Binding UserName}"
        FontSize="20" />
</VerticalStackLayout>
```

```
public partial class MainPage : ContentPage
{
    public string UserName { get; set; } = "";

    public MainPage()
    {
        InitializeComponent();
        BindingContext = this;
    }
}
```

Zasoby i style

- W wcześniejszych przykładach ustawialiśmy kolory, rozmiary czcionek i inne wartości bezpośrednio na każdej kontrolce. To działa, ale przy większych aplikacjach prowadzi do problemów – te same wartości powtarzają się w wielu miejscach. Jeśli chcemy zmienić główny kolor aplikacji z niebieskiego na zielony, musimy ręcznie poprawić dziesiątki kontrolek.
- Zasoby i style rozwiązują ten problem. Zasób to wartość zdefiniowana raz i używana w wielu miejscach – np. kolor, rozmiar, grubość ramki. Styl to zestaw właściwości, który możemy jednocześnie zastosować do wielu kontrolek – np. "wszystkie przyciski mają być niebieskie, pogrubione, z fontem 16".

Definiowanie zasobów

- Zasoby definiujemy wewnątrz ResourceDictionary – słownika, w którym każdy zasób ma swój klucz (x:Key) i wartość. Klucz to unikalna nazwa, po której będziemy się odwoływać do zasobu. Wartością może być praktycznie cokolwiek – kolor, tekst, liczba, grubość marginesu.
- **Konwencja nazewnictwa kluczy, to PascalCase**.

```
<ContentPage.Resources>
    <ResourceDictionary>
        <Color x:Key="KolorGlowny">#1E90FF</Color>
        <Color x:Key="KolorTekstu">#333333</Color>
        <x:Double x:Key="RozmiarTytulu">24</x:Double>
        <x:Double x:Key="RozmiarTresc">16</x:Double>
        <Thickness x:Key="StandardowyMargines">10,15,10,15</Thickness>
    </ResourceDictionary>
</ContentPage.Resources>
```

Mamy tu pięć zasobów – dwa kolory, dwa rozmiary czcionki i jedną wartość marginesu.
Każdy ma swój klucz, po którym będziemy go wywoływać.

StaticResource – użycie zasobu

- Sam zasób zdefiniowany w ResourceDictionary nic nie robi – trzeba go jeszcze zastosować na kontrolce. Do tego służy rozszerzenie znaczników StaticResource. Podajemy klucz zasobu, a MAUI podstawia odpowiednią wartość.

```
<ContentPage.Resources>
    <Color x:Key="KolorGlowny">#1E90FF</Color>
    <Color x:Key="KolorTekstu">#333333</Color>
    <x:Double x:Key="RozmiarTytulu">24</x:Double>
    <Thickness x:Key="StandardowyMargines">10,15,10,15</Thickness>
</ContentPage.Resources>

<VerticalStackLayout Padding="20" Spacing="15">
    <Label
        Text="Witaj w aplikacji"
        TextColor="{StaticResource KolorGlowny}"
        FontSize="{StaticResource RozmiarTytulu}" />
    <Button
        Text="Rozpocznij"
        BackgroundColor="{StaticResource KolorGlowny}"
        TextColor="White"
        Margin="{StaticResource StandardowyMargines}" />
    <Label
        Text="Opis aplikacji"
        TextColor="{StaticResource KolorTekstu}" />
</VerticalStackLayout>
```

Uwaga!
Słowo "Static" oznacza, że **wartość zasobu jest pobierana raz – w momencie ładowania strony**. Istnieje też DynamicResource, które aktualizuje się automatycznie gdy zasób się zmieni w trakcie działania aplikacji..

Zasoby lokalne i globalne

- Zasoby mogą być zdefiniowane na różnych poziomach – i od tego zależy, które kontrolki mogą z nich korzystać. Im wyżej w hierarchii umieścimy zasób, tym szerzej jest dostępny.

```xaml
<VerticalStackLayout>
    <VerticalStackLayout.Resources>
        <Color x:Key="KolorLokalny">Red</Color>
    </VerticalStackLayout.Resources>

    <Label TextColor="{StaticResource KolorLokalny}"
           Text="Mam dostęp" />
</VerticalStackLayout>

<!-- Tu już KolorLokalny nie jest dostępny -->
```

Zasób kontrolki – dostępny tylko dla tej kontrolki i jej dzieci.

```xaml
<ContentPage.Resources>
    <Color x:Key="KolorStrony">Blue</Color>
</ContentPage.Resources>
```

Zasób strony – dostępny dla wszystkich kontrolek na danej stronie.

```xaml
<!-- W pliku App.xaml -->
<Application.Resources>
    <Color x:Key="KolorAplikacji">Green</Color>
</Application.Resources>
```

Zasób globalny (**App.xaml**) – dostępny w całej aplikacji.

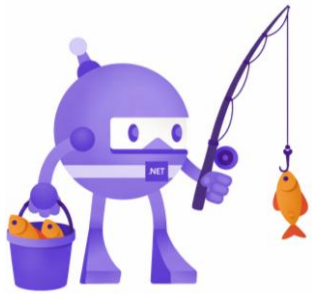
Uwaga!
Gdy kontrolka odwołuje się do StaticResource, **MAUI szuka klucza od dołu do góry** – najpierw w zasobach najbliższego rodzica, potem strony, a na końcu w App.xaml. **Używa pierwszego znalezionego wyniku**.

Style XAML

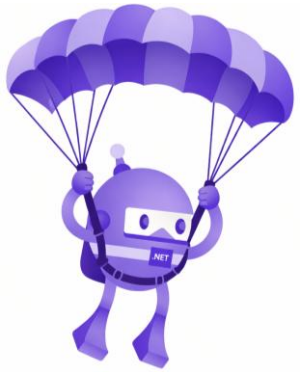
- Zasób to pojedyncza wartość – jeden kolor, jeden rozmiar. Styl idzie o krok dalej – to zestaw właściwości zgrupowanych pod jedną nazwą. Zamiast ustawiać na każdym przycisku osobno BackgroundColor, TextColor, FontSize i Padding, definiujemy styl raz i stosujemy go jednym odwołaniem.
- Budowa stylu:
  - **x:Key** – nazwa stylu, po której się do niego odwołujemy.
  - **TargetType** – typ kontrolki, do której styl może być zastosowany (Button, Label, Entry itd.).
  - **Setter** – pojedyncza właściwość i jej wartość.

  Styl może mieć dowolną liczbę Setterów.

```
<ContentPage.Resources>
    <Style x:Key="StylPrzycisku" TargetType="Button">
        <Setter Property="BackgroundColor" Value="#1E90FF" />
        <Setter Property="TextColor" Value="White" />
        <Setter Property="FontSize" Value="16" />
        <Setter Property="Padding" Value="10,8" />
    </Style>
</ContentPage.Resources>

<Button Text="Zapisz" Style="{StaticResource StylPrzycisku}" />
<Button Text="Anuluj" Style="{StaticResource StylPrzycisku}" />
<Button Text="Usuń" Style="{StaticResource StylPrzycisku}" />
```



Style XAML – niejawne



- W poprzednim przykładzie musieliśmy na każdym przycisku napisać: Style="{StaticResource StylPrzycisku}". Przy trzech przyciskach nie ma z tym większego problemu, ale przy dwudziestu robi się uciążliwe. Styl niejawny rozwiązuje ten problem – stosuje się automatycznie do wszystkich kontrolek danego typu, bez konieczności jawnego odwoływania się do niego. Różnica jest jedna, styl niejawny nie posiada x:Key.
- Jeżeli chcemy wyłączyć styl niejawny na konkretnej kontrolce, wystarczy nadpisać bezpośrednio jego właściwości – wartość ustawiona bezpośrednio, zawsze wygrywa ze stylem.

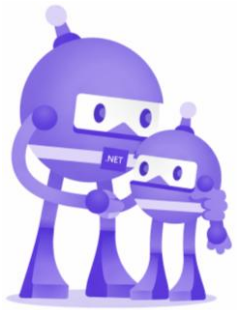
```
<Style TargetType="Button">
    <Setter Property="BackgroundColor" Value="#1E90FF" />
    <Setter Property="TextColor" Value="White" />
</Style>

<!-- Styl zastosowany automatycznie -->
<Button Text="Zapisz" />
<Button Text="Anuluj" />
<Button Text="Usuń" />
```

Styl niejawny działa na tym samym poziomie co zasoby – jeśli zdefiniujemy go w ContentPage.Resources, obejmie wszystkie kontrolki danego typu na tej stronie. Jeśli w App.xaml – w całej aplikacji.

Style XAML – dziedziczenie

- Czasem potrzebujemy kilku wariantów stylu – np. wszystkie przyciski mają wspólną czcionkę i padding, ale przyciski akcji głównej są niebieskie, a przyciski usuwania czerwone. Zamiast kopiować te same Settery do dwóch osobnych stylów, możemy jeden styl oprzeć na drugim za pomocą właściwości BasedOn. Styl potomny dziedziczy wszystkie Settery rodzica i dodaje lub nadpisuje wybrane.
- Zasady dziedziczenia:
  - Styl potomny może dodawać nowe Settery, których rodzic nie ma.
  - Styl potomny może nadpisywać Settery rodzica – ustawia tę samą Property na inną Value.
  - BasedOn działa tylko między stylami o tym samym TargetType.
  - Łańcuch dziedziczenia może mieć wiele poziomów – styl C oparty na B, który jest oparty na A.

```
<ContentPage.Resources>
    <Style x:Key="StylBazowy" TargetType="Button">
        <Setter Property="FontSize" Value="16" />
        <Setter Property="Padding" Value="12,8" />
        <Setter Property="TextColor" Value="White" />
    </Style>

    <Style x:Key="StylGlowny" TargetType="Button"
           BasedOn="{StaticResource StylBazowy}">
        <Setter Property="BackgroundColor" Value="#1E90FF" />
    </Style>

    <Style x:Key="StylUsuwania" TargetType="Button"
           BasedOn="{StaticResource StylBazowy}">
        <Setter Property="BackgroundColor" Value="#FF4444" />
    </Style>
</ContentPage.Resources>

<VerticalStackLayout Padding="20" Spacing="10">
    <Button Text="Zapisz"
            Style="{StaticResource StylGlowny}" />
    <Button Text="Edytuj"
            Style="{StaticResource StylGlowny}" />
    <Button Text="Usuń"
            Style="{StaticResource StylUsuwania}" />
</VerticalStackLayout>
```



Style XAML - klasy

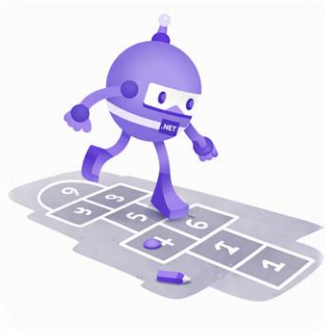
- W XAML możemy również korzystać z klas stylów, działają one podobnie jak w CSS. Kontrolka może mieć przypisaną jedną lub wiele klas, a każda klasa to osobny styl.

```xaml
<ContentPage.Resources>
    <Style Class="pogrubiony" TargetType="Label">
        <Setter Property="FontAttributes" Value="Bold" />
    </Style>
    <Style Class="duzy" TargetType="Label">
        <Setter Property="FontSize" Value="24" />
    </Style>
    <Style Class="niebieski" TargetType="Label">
        <Setter Property="TextColor" Value="Blue" />
    </Style>
</ContentPage.Resources>
```

Zamiast x:Key używamy atrybutu Class

```xaml
<!-- Stosowanie jednej klasy -->
<Label Text="Pogrubiony tekst" StyleClass="pogrubiony" />

<!-- Stosowanie wielu klas -->
<Label Text="Duży, pogrubiony, niebieski"
       StyleClass="pogrubiony,duzy,niebieski" />
```

Stosowanie klasy na kontrolce



Style XAML – organizacja

- W małych aplikacjach możemy trzymać wszystkie zasoby w App.xaml. Ale gdy projekt rośnie, ten plik szybko staje się nieczytelny – setki linii kolorów, czcionek, stylów przycisków, etykiet, wpisanych w jedno miejsce. Dlatego w praktyce dzielimy zasoby na osobne pliki tematyczne, umieszczone w katalogu Resources/Styles.

```
Resources/
└── Styles/
    ├── Colors.xaml        <- kolory aplikacji
    ├── Fonts.xaml         <- rozmiary i rodziny czcionek
    ├── Styles.xaml        <- style kontrolek
    └── ButtonStyles.xaml  <- style konkretnego typu (opcjonalnie)
```

```xml
<?xml version="1.0" encoding="utf-8" ?>
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml">

    <Color x:Key="Primary">#1E90FF</Color>
    <Color x:Key="Secondary">#6C757D</Color>
    <Color x:Key="Danger">#FF4444</Color>
    <Color x:Key="TextDark">#333333</Color>
    <Color x:Key="TextLight">#FFFFFF</Color>
    <Color x:Key="Background">#F5F5F5</Color>

</ResourceDictionary>
```

Colors.xaml jako osobny ResourceDictionary

Style XAML – organizacja

- Osobne pliki nie łączą się automatycznie – musimy je scalić w App.xaml za pomocą MergedDictionaries.
- Kolejność ma znaczenie – jeśli ten sam klucz pojawia się w dwóch plikach, wygrywa plik podany później. Dlatego Styles.xaml (który może odwoływać się do kolorów i czcionek) powinien być na końcu.

```xaml
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="Resources/Styles/Colors.xaml" />
            <ResourceDictionary Source="Resources/Styles/Fonts.xaml" />
            <ResourceDictionary Source="Resources/Styles/Styles.xaml" />
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

Łączenie stylów w App.xaml

Style C#

- Wszystko co zdefiniowaliśmy w XAML można też napisać w czystym C#. Podejście to pozwala na stosowanie stylów generowanych dynamicznie, lub gdy są zależne od warunków, których nie da się łatwo wyrazić w XAML.

```xaml
<Style x:Key="StylPrzycisku" TargetType="Button">
    <Setter Property="BackgroundColor" Value="#1E90FF" />
    <Setter Property="TextColor" Value="White" />
    <Setter Property="FontSize" Value="16" />
    <Setter Property="Padding" Value="10,8" />
</Style>
```

Styl napisany w XAML

```csharp
// Dodawanie stylu do zasobów, z poziomu C#
Resources.Add("StylPrzycisku", stylPrzycisku);

// Dodawanie stylu, jako styl niejawny - bez klucza
Resources.Add(stylPrzycisku);

// Tworzenie przycisku, który używa danego stylu
var przycisk = new Button
{
    Text = "Zapisz",
    Style = (Style)Resources["StylPrzycisku"]
};
```

Stosowanie stylu napisanego w C#

```csharp
var stylPrzycisku = new Style(typeof(Button))
{
    Setters =
    {
        new Setter { Property = Button.BackgroundColorProperty,
                     Value = Color.FromArgb("#1E90FF") },
        new Setter { Property = Button.TextColorProperty,
                     Value = Colors.White },
        new Setter { Property = Button.FontSizeProperty,
                     Value = 16.0 },
        new Setter { Property = Button.PaddingProperty,
                     Value = new Thickness(10, 8) }
    }
};
```

Ten sam styl, ale napisany w C#

Style CSS

- .NET MAUI pozwala stylować aplikację za pomocą CSS. To alternatywa dla stylów XAML-owych, szczególnie atrakcyjna dla programistów mających doświadczenie z web developmentem. CSS w MAUI nie zastępuje w pełni stylów XAML – wspiera najważniejsze właściwości, ale nie wszystkie. Traktujemy go jako uzupełnienie, nie zamiennik.
- Selektory CSS w MAUI:
  - **Selektor typu** – po nazwie kontrolki (małe litery): stacklayout, button, label. Działa jak styl niejawny w XAML – stosuje się do wszystkich kontrolek danego typu.
  - **Selektor bazowy** – z prefiksem „^”, np. ^contentpage. Wybiera kontrolki po klasie bazowej, nie tylko po dokładnym typie. To selektor specyficzny dla MAUI, nie istnieje w standardowym CSS.
  - **Selektor klasy** – z kropką: .mainPageTitle. Stosujemy do kontrolek, które mają ustawioną właściwość StyleClass.
  - **Selektor identyfikatora** – z hashem: #listView. Wybiera kontrolkę po StyleId lub x:Name.


```css
stacklayout {
    margin: 20;
    -maui-spacing: 6;
}

^contentpage {
    background-color: lightgray;
}

.mainPageTitle {
    font-style: bold;
    font-size: 14;
}

.mainPageSubtitle {
    margin-top: 15;
}

#listView {
    background-color: lightgray;
}
```

Style CSS

- Plik CSS dodajemy do projektu z akcją budowania MauiCss, a następnie ładujemy go w App.xaml za pomocą elementu StyleSheet.

```
<Application ...>
    <Application.Resources>
        <StyleSheet Source="/Resources/styles.css" />
    </Application.Resources>
</Application>
```

Rejestrowanie stylów CSS w App.xaml

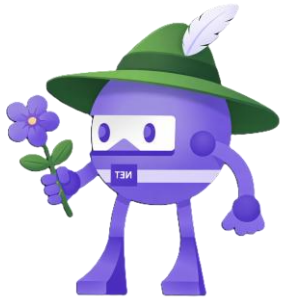
```
<Label Text="Tytuł strony"
       StyleClass="mainPageTitle" />
<Label Text="Podtytuł"
       StyleClass="mainPageSubtitle" />
```

Stosowanie klas CSS na kontrolkach MAUI

```
<ItemGroup>
    <MauiCss Include="Resources\styles.css" />
</ItemGroup>
```

Dodawanie akcji kompilacji w pliku projektu (.csproj)

Visual Studio powinno dodać tę sekcję automatycznie.



Style CSS

- CSS może być używany w MAUI, ale tylko w ograniczonym zakresie – nie ma tutaj tak wielu możliwości, jak w przypadku HTML.
- CSS w .NET MAUI obsługuje:
  - Kolory (color, background-color),
  - czcionki (font-size, font-style, font-family),
  - marginesy i paddingi (margin, padding),
  - wyrównanie (text-align),
  - widoczność (visibility, opacity)
  - oraz specyficzne właściwości MAUI z prefiksem -maui- (np. -maui-spacing, -maui-bar-background-color dla Shell).
- CSS w MAUI nie obsługuje:
  - Zmienne CSS,
  - dziedziczenie (inherit),
  - skrócone właściwości jak font czy border,
  - pseudoklasy (:hover, :active),
  - media queries.
  - Nie da się też targetować elementów Span.

[Style apps using Cascading Style Sheets - .NET MAUI | Microsoft Learn]

Uwaga!
CSS w MAUI jest parsowany w trakcie działania aplikacji, nie w trakcie kompilacji. To oznacza, że błędy w CSS nie spowodują błędu kompilacji – po prostu style się nie zastosują. Warto o tym pamiętać przy debugowaniu.

Nawigacja w .NET MAUI

- Do tej pory wszystkie przykłady działały na jednym ekranie – jedna strona, jeden zestaw kontrolek. Ale realne aplikacje składają się z wielu ekranów: lista produktów, szczegóły produktu, koszyk, ustawienia, profil użytkownika. Użytkownik musi móc się między nimi przemieszczać.
- .NET MAUI oferuje dwa podejścia do nawigacji:
  - **NavigationPage** – klasyczna nawigacja oparta na stosie stron. Prosta, bezpośrednia, dobra do liniowych przepływów (np. lista -> szczegóły -> edycja). Działa jak stos – nowe strony "kładziemy na wierzch", a cofając się zdejmujemy je ze stosu.
  - **Shell** – nowoczesny system nawigacji z wbudowaną obsługą zakładek, menu bocznego i nawigacji przez URI. Bardziej rozbudowany, lepszy do złożonych aplikacji z wieloma sekcjami. Domyślnie generowany w nowym projekcie MAUI.

NavigationPage – stos stron

- NavigationPage zarządza nawigacją za pomocą stosu (stack). Nowa strona trafia na wierzch stosu (push), a cofanie zdejmuje ją ze stosu (pop). Użytkownik zawsze widzi stronę, która jest aktualnie na szczycie.

1. Aby móc korzystać z tego mechanizmu, należy opakować stronę startową projektu w NavigationPage – to wystarczy, żeby zyskać nawigację i automatyczny pasek nawigacji z przyciskiem cofania.
2. Przejście na nową stronę odbywa się poprzez stworzenie nowej instancji tej strony (obiektu) oraz ułożenie go na stosie, przy użyciu metody PushAsync().
3. Cofanie się odbywa się poprzez ściągnięcie aktualnej strony ze stosu – użytkownik wraca do poprzedniej.

```
public App()
{
    InitializeComponent();
    MainPage = new NavigationPage(new MainPage());
}
```

1

```
private async void OnSzczegolyClicked(object sender, EventArgs e)
{
    await Navigation.PushAsync(new SzczegolyPage());
}
```

2

```
private async void OnCofnijClicked(object sender, EventArgs e)
{
    await Navigation.PopAsync();
}
```

3

NavigationPage – przekazywanie danych

- Sama nawigacja to za mało – zazwyczaj chcemy przekazać informację do nowej strony. Np. użytkownik klika produkt na liście i oczekuje, że na stronie szczegółów zobaczy dane tego konkretnego produktu, a nie pustą stronę. Najprostszy sposób to przekazanie danych przez konstruktor nowej strony.

```
private async void OnProduktClicked(object sender, EventArgs e)
{
    string nazwaProdukt = "Laptop Lenovo ThinkPad";
    double cena = 4599.99;

    await Navigation.PushAsync(
        new SzczegolyPage(nazwaProdukt, cena));
}
```

Wysyłanie danych do innej strony

```
public partial class SzczegolyPage : ContentPage
{
    public SzczegolyPage(string nazwa, double cena)
    {
        InitializeComponent();

        labelNazwa.Text = nazwa;
        labelCena.Text = $"{cena:F2} zł";
    }
}
```

Odbieranie danych na innej stronie

Shell

- Shell to nowoczesny system nawigacji w .NET MAUI – bardziej rozbudowany niż NavigationPage, ale też wygodniejszy przy większych aplikacjach. Zamiast ręcznie zarządzać stosem stron, definiujemy strukturę aplikacji deklaratywnie w jednym pliku – AppShell.xaml.
- Shell automatycznie generuje pasek nawigacyjny, zakładki, menu boczne i obsługuje przejścia między stronami.

Shell

- Kluczowe elementy Shell:
  - **ShellContent** – pojedyncza strona w aplikacji, najniższy poziom hierarchii.
  - **Tab** – zakładka, może zawierać jeden lub więcej ShellContent.
  - **TabBar** – pasek zakładek na dole ekranu.
  - **FlyoutItem** – element menu bocznego (wysuwanego z lewej strony).
- Hierarchia:
  - Shell może zawierać FlyoutItem lub TabBar.
  - FlyoutItem i TabBar mogą zawierać Tab.
  - Tab zawiera ShellContent.
  - ShellContent wskazuje na konkretną stronę (ContentPage).

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MojaAplikacja"
    x:Class="MojaAplikacja.AppShell">

    <ShellContent
        Title="Strona główna"
        Icon="home.png"
        ContentTemplate="{DataTemplate local:MainPage}" />

</Shell>
```

Plik AppShell.xaml

Uwaga!
Shell i NavigationPage to dwa osobne podejścia – **nie mieszamy ich ze sobą**. Jeśli korzystamy z Shell, nawigację obsługujemy przez mechanizmy Shell, a nie przez Navigation.PushAsync.

Nawigacja z zakładkami - TabBar

- TabBar tworzy pasek zakładek na dole ekranu – użytkownik przełącza się między sekcjami jednym kliknięciem. To najpopularniejszy wzorzec nawigacji w aplikacjach mobilnych – znamy go z większości aplikacji na telefonie.

```xml
<Shell xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:local="clr-namespace:MojaAplikacja"
       x:Class="MojaAplikacja.AppShell">

    <TabBar>
        <Tab Title="Główna" Icon="home.png">
            <ShellContent
                ContentTemplate="{DataTemplate local:MainPage}" />
        </Tab>
        <Tab Title="Szukaj" Icon="search.png">
            <ShellContent
                ContentTemplate="{DataTemplate local:SearchPage}" />
        </Tab>
        <Tab Title="Profil" Icon="profile.png">
            <ShellContent
                ContentTemplate="{DataTemplate local:ProfilePage}" />
        </Tab>
    </TabBar>

</Shell>
```

AppShell.xaml z trzema zakładkami

Trzy zakładki – każda prowadzi do osobnej strony. Ikony i tytuły wyświetlają się automatycznie na pasku zakładek.

Nawigacja z zakładkami – TabBar

- Jeśli każda zakładka zawiera tylko jeden ShellContent, możemy pominąć element Tab, i skorzystać z skróconego zapisu. Efekt jest identyczny – Shell automatycznie opakowuje każdy ShellContent w Tab.
- Zwróć uwagę na ContentTemplate="{DataTemplate local:MainPage}" – Shell nie tworzy strony od razu, tylko czeka aż użytkownik przejdzie na daną zakładkę. Dzięki temu aplikacja uruchamia się szybciej, bo nie ładuje wszystkich stron na raz.

```
<TabBar>
    <ShellContent Title="Główna" Icon="home.png"
        ContentTemplate="{DataTemplate local:MainPage}" />
    <ShellContent Title="Szukaj" Icon="search.png"
        ContentTemplate="{DataTemplate local:SearchPage}" />
    <ShellContent Title="Profil" Icon="profile.png"
        ContentTemplate="{DataTemplate local:ProfilePage}" />
</TabBar>
```

Nawigacja z zakładkami – TabBar

- Tab może zawierać więcej niż jeden ShellContent. Wtedy na górze strony pojawia się dodatkowy pasek pozwalający przełączać się między podstronami w ramach tej samej zakładki.
  - Zakładka "Główna" ma dwie podstrony – Aktualności i Polecane.
  - Użytkownik widzi na dole zakładki "Główna" i "Profil„.
  - Po wejściu w "Główna" na górze pojawia się dodatkowy przełącznik między Aktualności a Polecane.
  - Zakładka "Profil" ma tylko jedną stronę, więc dodatkowy pasek się nie pojawia.

```
<TabBar>
    <Tab Title="Główna" Icon="home.png">
        <ShellContent Title="Aktualności"
            ContentTemplate="{DataTemplate local:NewsPage}" />
        <ShellContent Title="Polecane"
            ContentTemplate="{DataTemplate local:FeaturedPage}" />
    </Tab>
    <Tab Title="Profil" Icon="profile.png">
        <ShellContent
            ContentTemplate="{DataTemplate local:ProfilePage}" />
    </Tab>
</TabBar>
```

Menu boczne – FlyoutItem

- Flyout to wysuwane menu boczne – użytkownik otwiera je przesunięciem palca od lewej krawędzi ekranu lub kliknięciem ikony "hamburgera" (trzy poziome kreski).
- Stosujemy je, gdy aplikacja ma wiele sekcji, które nie zmieściłyby się wygodnie w pasku zakładek – np. sklep, ustawienia, pomoc, kontakt, historia zamówień.

Każdy FlyoutItem to pozycja w menu bocznym. Po kliknięciu menu się zamyka i użytkownik trafia na wybraną stronę.

```
<Shell xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:local="clr-namespace:MojaAplikacja"
       x:Class="MojaAplikacja.AppShell">

    <FlyoutItem Title="Główna" Icon="home.png">
        <ShellContent
            ContentTemplate="{DataTemplate local:MainPage}" />
    </FlyoutItem>
    <FlyoutItem Title="Produkty" Icon="products.png">
        <ShellContent
            ContentTemplate="{DataTemplate local:ProductsPage}" />
    </FlyoutItem>
    <FlyoutItem Title="Koszyk" Icon="cart.png">
        <ShellContent
            ContentTemplate="{DataTemplate local:CartPage}" />
    </FlyoutItem>
    <FlyoutItem Title="Ustawienia" Icon="settings.png">
        <ShellContent
            ContentTemplate="{DataTemplate local:SettingsPage}" />
    </FlyoutItem>

</Shell>
```

Menu boczne z zakładkami

- FlyoutItem może zawierać Tab – wtedy po wybraniu pozycji z menu użytkownik widzi stronę z zakładkami na dole.

```
<FlyoutItem Title="Sklep" Icon="shop.png">
    <Tab Title="Produkty" Icon="products.png">
        <ShellContent
            ContentTemplate="{DataTemplate local:ProductsPage}" />
    </Tab>
    <Tab Title="Kategorie" Icon="categories.png">
        <ShellContent
            ContentTemplate="{DataTemplate local:CategoriesPage}" />
    </Tab>
</FlyoutItem>
<FlyoutItem Title="Ustawienia" Icon="settings.png">
    <ShellContent
        ContentTemplate="{DataTemplate local:SettingsPage}" />
</FlyoutItem>
```

Sekcja "Sklep" w menu bocznym zawiera dwie zakładki – Produkty i Kategorie. Sekcja "Ustawienia" to pojedyncza strona bez zakładek.

Shell – nawigacja przez URI

- Shell oferuje nawigację opartą na URI – adresach stron, podobnych do adresów URL w przeglądarce. Zamiast tworzyć obiekt strony i kłaść go na stos, podajemy ścieżkę jako tekst.
- To podejście znane z aplikacji internetowych – każda strona ma swoją "trasę" (route), a nawigacja to przejście pod odpowiedni adres.

```
// Przejście do strony
await Shell.Current.GoToAsync("szczegoly");

// Cofnięcie do poprzedniej strony
await Shell.Current.GoToAsync("..");
```

Dwie kropki .. działają jak w systemie plików – oznaczają "wróć o jeden poziom wyżej", czyli do poprzedniej strony.

Shell – przekazywanie parametrów przez URI

- Przekazywanie parametrów
  - Parametry dodajemy po znaku zapytania, rozdzielone ampersandem – dokładnie jak w adresach URL.

```
await Shell.Current.GoToAsync($"szczegoly?id={produktId}&nazwa={nazwa}");
```

- Odbieranie parametrów
  - Atrybut QueryProperty łączy parametr z URI (np. "id") z właściwością klasy (np. ProduktId). Shell automatycznie ustawia wartość właściwości przed wyświetleniem strony.

```
[QueryProperty(nameof(ProduktId), "id")]
[QueryProperty(nameof(ProduktNazwa), "nazwa")]
public partial class SzczegolyPage : ContentPage
{
    public string ProduktId { get; set; }
    public string ProduktNazwa { get; set; }
```

```
    public SzczegolyPage()
    {
        InitializeComponent();
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();
        labelNazwa.Text = ProduktNazwa;
        labelId.Text = $"ID: {ProduktId}";
    }
}
```

Nawigacja względna i bezwzględna

- Ścieżka względna – przechodzi "w głąb" od aktualnej pozycji.

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

- Ścieżka bezwzględna – zaczyna się od „//”, przechodzi do konkretnego miejsca niezależnie od tego, gdzie jesteśmy.

```
await Shell.Current.GoToAsync("//glowna");
```

Rejestrowanie tras

- Strony zdefiniowane bezpośrednio w AppShell.xaml (jako ShellContent) mają trasy automatycznie – Shell generuje je na podstawie struktury. Ale nie każda strona powinna być widoczna w menu czy zakładkach.
- Strona szczegółów produktu, ekran edycji, formularz dodawania – to strony, do których nawigujemy dynamicznie, a nie przez stałą pozycję w interfejsie. Takie strony musimy zarejestrować ręcznie.

```
public partial class AppShell : Shell
{
    public AppShell()
    {
        InitializeComponent();

        Routing.RegisterRoute("szczegoly", typeof(SzczegolyPage));
        Routing.RegisterRoute("edycja", typeof(EdycjaPage));
        Routing.RegisterRoute("dodaj", typeof(DodajPage));
    }
}
```

Pierwszy parametr to nazwa trasy (ten sam tekst, którego użyjemy w GoToAsync), drugi to typ strony, na którą trasa prowadzi.

Rejestrowanie tras – pełny przepływ

1. Rejestrujemy trasę w AppShell.xaml.cs.

```
Routing.RegisterRoute("szczegoly", typeof(SzczegolyPage));
```

2. Nawigujemy do strony SzczegolyPage z parametrami.

```
await Shell.Current.GoToAsync($"szczegoly?id={produktId}");
```

3. Strona docelowa odbiera parametry.

```
[QueryProperty(nameof(ProduktId), "id")]
public partial class SzczegolyPage : ContentPage
{
    public string ProduktId { get; set; }
}
```

Rejestrowanie tras – kiedy?

- Strony w AppShell.xaml (ShellContent, Tab, FlyoutItem) – trasy generowane automatycznie, nie rejestrujemy.
- Strony, do których nawigujemy przez GoToAsync, ale nie są w AppShell.xaml – musimy zarejestrować ręcznie.

Uwaga!
Nazwy tras muszą być unikalne w całej aplikacji. Jeśli zarejestrujemy dwie trasy pod tą samą nazwą, Shell nie będzie wiedział, którą stronę otworzyć. Dobrą praktyką jest **nazywanie tras tak samo jak strony, tylko małymi literami**:

- SzczegolyPage - "szczegoly",
- EdycjaPage - "edycja".

Rys historyczny – skąd wzięło się .NET MAUI?

- .NET MAUI nie powstał od zera. Jego bezpośrednim poprzednikiem był Xamarin, a dokładniej Xamarin.Forms – framework do tworzenia aplikacji mobilnych w C#.

- Xamarin pozwalał:
  - pisać aplikacje na Androida i iOS,
  - używać jednego języka (C#),
  - współdzielić dużą część kodu.

To była pierwsza realna próba „cross-platform” w świecie .NET.

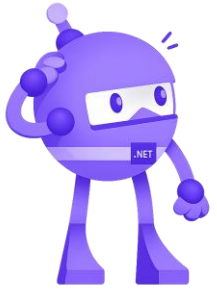
Xamarin.Forms – co robił dobrze?

- Xamarin.Forms wprowadził:
  - jeden wspólny interfejs użytkownika,
  - renderowanie natywnych kontrolek,
  - XAML jako język opisu UI,
  - ideę „shared code”.

Dla wielu aplikacji to wystarczało oraz działało całkiem nieźle.

- Z czasem pojawiły się problemy:
  - osobne projekty dla platform,
  - skomplikowana struktura,
  - różne wersje frameworków,
  - ograniczona spójność z nowoczesnym .NET.

- Xamarin był:
  - skuteczny,
  - trudny w utrzymaniu,
  - coraz mniej pasował do nowego ekosystemu .NET.

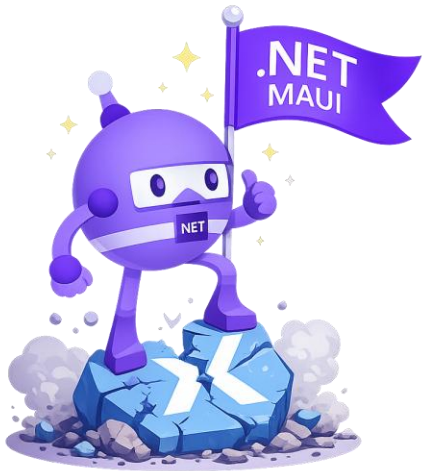


Przejście na .NET MAUI

- Microsoft postanowił:
  - zintegrować rozwój aplikacji UI bezpośrednio z .NET,
  - uprościć strukturę projektów,
  - połączyć mobilne i desktopowe aplikacje w jeden model.

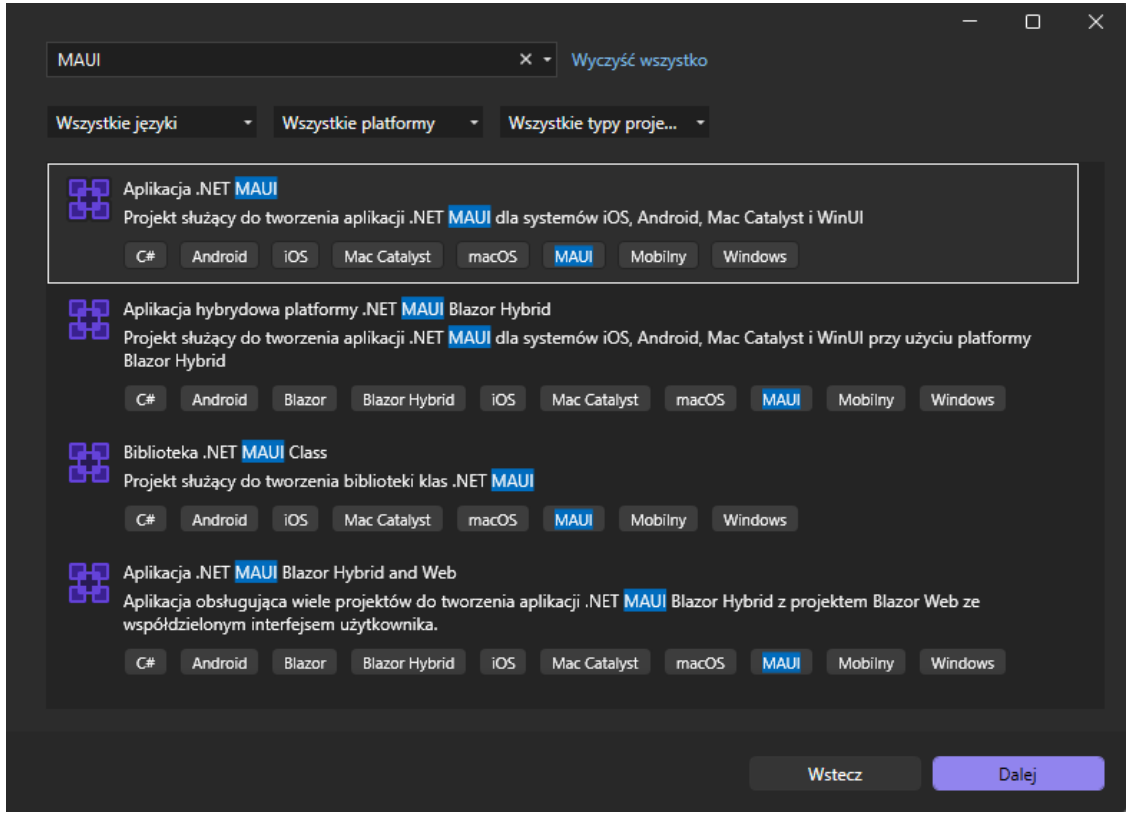
Tak powstał .NET MAUI – jeden projekt, jeden ekosystem, jeden sposób tworzenia aplikacji.

- Co zmieniło się w MAUI względem Xamarin?
  - Jeden projekt zamiast wielu,
  - lepsza integracja z .NET,
  - wspólne API dla mobile i desktop,
  - uproszczona obsługa platform,
  - nowoczesne podejście do UI.



Inne warianty .NET MAUI

- Oprócz MAUI w swojej podstawowej formie, istnieją inne warianty, które łączą MAUI z technologiami webowymi.



Szablony MAUI dostępne w Visual Studio 2026

.NET MAUI Blazor

- .NET MAUI Blazor pozwala budować interfejs aplikacji mobilnej i desktopowej za pomocą technologii webowych – HTML *(przy użyciu składni Razor)*, CSS i C# (przez framework Blazor). Zamiast XAML-owych kontrolek, piszemy komponenty Blazor, które renderują się wewnątrz kontrolki BlazorWebView – osadzonej przeglądarki w aplikacji natywnej.
- Dla kogo to jest?
  - Dla programistów, którzy znają już HTML i CSS i chcą tworzyć aplikacje mobilne bez nauki XAML.
  - Dla zespołów, które mają gotowe komponenty webowe i chcą je wykorzystać w aplikacji natywnej.
- **Kluczowa różnica** – interfejs nie korzysta z natywnych kontrolek systemu. Przycisk w MAUI Blazor to element HTML renderowany w WebView, nie natywny Button Androida czy iOS. Aplikacja wygląda tak samo na każdej platformie, ale traci natywny charakter.

```razor
<div style="padding: 20px;">
    <h1>Witaj w aplikacji!</h1>
    <input @bind="imie" placeholder="Wpisz imię" />
    <button @onclick="Przywitaj">Przywitaj</button>
    <p style="font-size: 18px;">@wynik</p>
</div>

@code {
    private string imie = "";
    private string wynik = "";

    private void Przywitaj()
    {
        wynik = $"Cześć, {imie}!";
    }
}
```

.NET MAUI Blazor Hybrid

- MAUI Blazor Hybrid łączy oba światy – część interfejsu budujemy w XAML z natywnymi kontrolkami, a część w HTML/CSS przez Blazor.
- W ramach jednej aplikacji możemy mieć natywną nawigację i pasek zakładek (XAML), a wewnątrz konkretnych stron osadzić komponenty Blazor (HTML/CSS).
- To podejście pragmatyczne – pozwala szybko dostarczyć aplikację mobilną, wykorzystując kod, który już istnieje.

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MojaAplikacja.DashboardPage">

    <BlazorWebView HostPage="wwwroot/index.html">
        <BlazorWebView.RootComponents>
            <RootComponent Selector="#app"
                           ComponentType="{x:Type local:Dashboard}" />
        </BlazorWebView.RootComponents>
    </BlazorWebView>

</ContentPage>
```

Strona XAML z komponentem Razor