Cykl życia, wzorce projektowe i MVVM

Bartosz Miąskowski

Czym jest cykl życia aplikacji?

- Aplikacja mobilna nie działa jak aplikacja konsolowa – system może ją w dowolnym momencie wstrzymać, wznowić lub zamknąć.
- Użytkownik przełącza się między aplikacjami, telefon odbiera połączenie, kończy się pamięć – nasza aplikacja musi być na to gotowa.

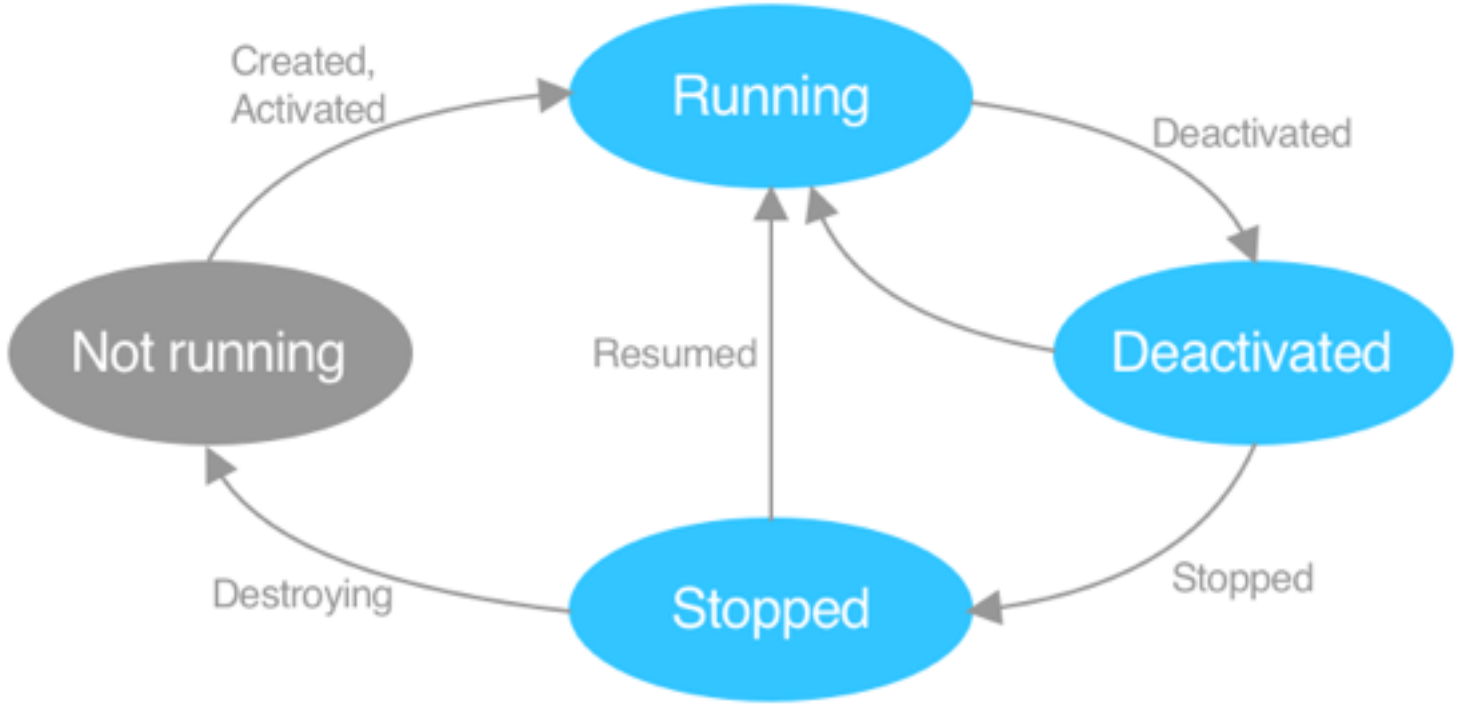
- **Musimy wiedzieć kiedy**:
  - zapisać dane użytkownika,
  - zwolnić zasoby,
  - odświeżyć interfejs.

Cykl życia aplikacji

Stan	Opis
Created	Aplikacja została utworzona i załadowana do pamięci
Activated	Aplikacja jest widoczna i aktywna na ekranie
Deactivated	Użytkownik przełączył się do innej aplikacji
Stopped	Aplikacja jest w tle, ale wciąż żyje w pamięci
Resumed	Aplikacja wraca na pierwszy plan po wznowieniu
Destroying	Aplikacja jest zamykana i usuwana z pamięci

Aplikacja .NET MAUI przechodzi przez różne stany w zależności od tego, co robi **użytkownik** i **system**.

Cykl życia aplikacji



- Szary owal – aplikacja nie jest załadowana do pamięci.
- Niebieskie owale – aplikacja jest w pamięci.
- Strzałki – zdarzenia wywoływane przez .NET MAUI.

Stopped nie oznacza zamknięcia – aplikacja dalej siedzi w pamięci. System może ją zabić bez ostrzeżenia jeśli potrzebuje RAMu – wtedy Destroying może się nie uruchomić.

Obsługa zdarzeń z cyklu życia


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

    // Tworzymy okno i podpinamy zdarzenia
    protected override Window CreateWindow(IActivationState? activationState)
    {
        Window window = new Window(new AppShell());

        window.Created += (s, e) =>
            Debug.WriteLine(">>> Window Created");

        window.Activated += (s, e) =>
            Debug.WriteLine(">>> Window Activated");

        window.Deactivated += (s, e) =>
            Debug.WriteLine(">>> Window Deactivated");

        window.Stopped += (s, e) =>
            Debug.WriteLine(">>> Window Stopped");

        window.Resumed += (s, e) =>
            Debug.WriteLine(">>> Window Resumed");

        window.Destroying += (s, e) =>
            Debug.WriteLine(">>> Window Detroying");

        return window;
    }
}
```

App.xaml.cs

Zdarzenia subskrybujemy na obiekcie Window. Metoda CreateWindow to miejsce, w którym MAUI tworzy główne okno aplikacji.

Cykl życia strony

- Oprócz cyklu życia całej aplikacji, wyróżniamy również cykl życia strony (ContentPage). Wyróżniamy tutaj dwa zdarzenia:
  - **OnAppearing()** – strona zaraz będzie widoczna na ekranie.
  - **OnDisappearing()** – strona zaraz zniknie z ekranu.

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

    protected override void OnAppearing()
    {
        base.OnAppearing();
        Debug.WriteLine(">>> ProductsPage: OnAppearing");
        // Tu: odśwież listę produktów, uruchom timer
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();
        Debug.WriteLine(">>> ProductsPage: OnDisappearing");
        // Tu: zatrzymaj timer, zapisz stan
    }
}
```

Cykl życia strony

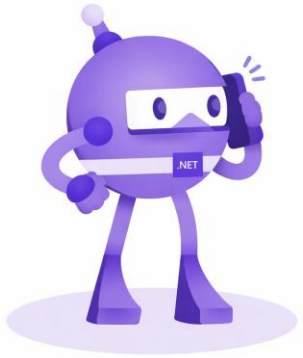
Sytuacja	OnAppearing	OnDisappearing
Nawigacja do strony (Push)	nowa strona	–
Nawigacja wstecz (Pop)	strona pod spodem	zdejmowana strona
Przełączenie zakładki (Tab)	nowa zakładka	stara zakładka
Minimalizacja aplikacji	–	bieżąca strona
Powrót z tła	bieżąca strona	–

- **OnAppearing()** to świetne miejsce na odświeżenie danych, przykładowo kiedy wracamy na daną stronę, a one mogły w tym czasie się zmienić.
- **OnDisappearing()** powinno obsługiwać odsubskrybowanie wszystkich zdarzeń, zasubskrybowanych na danej stronie.

Wzorce projektowe

Definicja:

Wzorzec projektowy (ang. *design pattern*) to opis sprawdzonego rozwiązania powtarzającego się problemu w programowaniu. To nie jest gotowy kod do skopiowania – to schemat myślenia, który można dostosować do konkretnej sytuacji.

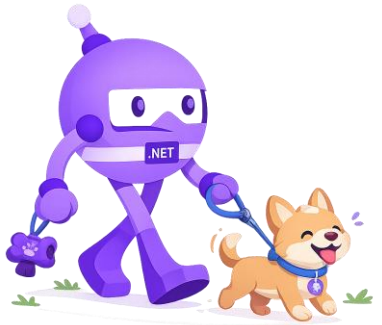


Wzorce projektowe

- Dlaczego warto stosować wzorce projektowe?
  - **Wspólny język w zespole** Zamiast: *"zróbmy taką klasę, która będzie jedyna w całej aplikacji i będzie trzymać konfigurację„*, mówisz: *„użyjmy Singletona"* - Wszyscy wiedzą o co chodzi.
  - **Sprawdzone rozwiązania** Nie wymyślasz koła na nowo. Tysiące programistów przed Tobą miało ten sam problem i wypracowali najlepsze podejście.
  - **Łatwiejsze utrzymanie kodu** Kod oparty na wzorcach jest przewidywalny – nowy programista w zespole szybciej go zrozumie.
  - **Łatwiejsze rozszerzanie** Wzorce są projektowane z myślą o zmianach – dodanie nowej funkcjonalności nie wymaga przepisywania połowy aplikacji.

Klasyfikacja wzorców projektowych

Rodzina	Czym się zajmuje?	Przykłady
Kreacyjne *(Creational)*	Jak tworzyć obiekty?	Singleton, Factory, Builder
Strukturalne *(Structural)*	Jak łączyć obiekty w większe struktury?	Adapter, Decorator, Facade
Behawioralne (Behavioral)	Jak obiekty ze sobą komunikują?	Observer, Strategy, Command



Singleton – tylko jedna instancja

Problem:

Potrzebujemy dokładnie jednego obiektu danego typu w całej aplikacji – np. połączenie z bazą danych, konfiguracja, logger. Tworzenie wielu instancji powodowałoby konflikty lub marnowanie zasobów.

Rozwiązanie:

Klasa sama kontroluje, że istnieje tylko jedna jej instancja, i udostępnia do niej globalny punkt dostępu.

```
public class AppSettings
{
    private static AppSettings? _instance;

    // Prywatny konstruktor – nikt z zewnątrz nie utworzy new AppSettings()
    private AppSettings() { }

    public static AppSettings Instance
    {
        get
        {
            _instance ??= new AppSettings();
            return _instance;
        }
    }

    public string ApiUrl { get; set; } = "https://api.sklep.pl";
    public string AppVersion { get; set; } = "1.0.0";
}

// Użycie – wszędzie ten sam obiekt:
var url = AppSettings.Instance.ApiUrl;
```

Uwaga!
W MAUI za rejestrację serwisów jako **Singleton** odpowiedzialny jest mechanizm **DI** (Dependency Injection).

Factory – tworzenie obiektów bez wiedzy o typie

Problem:

Chcemy tworzyć obiekty, ale konkretny typ zależy od sytuacji.
Nie chcemy rozsiewać `if/switch + new` po całym kodzie.

Rozwiązanie:

Wydzielamy logikę tworzenia obiektów do osobnej metody lub klasy.

```csharp
// Wspólny interfejs
public interface INotificationSender
{
    void Send(string message);
}

public class EmailSender : INotificationSender
{
    public void Send(string message) => /* wyślij e-mail */;
}

public class SmsSender : INotificationSender
{
    public void Send(string message) => /* wyślij SMS */;
}

public class PushSender : INotificationSender
{
    public void Send(string message) => /* wyślij push */;
}

// Factory – decyduje, który typ utworzyć
public static class NotificationFactory
{
    public static INotificationSender Create(string type) => type switch
    {
        "email" => new EmailSender(),
        "sms"   => new SmsSender(),
        "push"  => new PushSender(),
        _       => throw new ArgumentException($"Nieznany typ: {type}")
    };
}

// Użycie:
var sender = NotificationFactory.Create("email");
sender.Send("Zamówienie potwierdzone!");
```

Adapter – tłumacz między interfejsami

Problem:

Mamy dwie klasy, które powinny ze sobą współpracować, ale mają niekompatybilne interfejsy. Nie chcemy (lub nie możemy) zmieniać żadnej z nich.

Rozwiązanie:

Tworzymy klasę pośredniczącą, która "tłumaczy" jeden interfejs na drugi.

Analogia:

Adapter do gniazdka – europejska wtyczka nie pasuje do brytyjskiego gniazdka, ale adapter rozwiązuje problem bez przerabiania wtyczki ani gniazdka.

```
// Stara biblioteka – nie możemy jej zmienić
public class StaryCsvExporter
{
    public string ExportujDoCsv(string[] wiersze)
    {
        return string.Join("\n", wiersze);
    }
}

// Nasz system oczekuje takiego interfejsu
public interface IDataExporter
{
    string Export(List<Product> products);
}

// Adapter – łączy stary kod z nowym interfejsem
public class CsvExporterAdapter : IDataExporter
{
    private readonly StaryCsvExporter _staryCsv = new();

    public string Export(List<Product> products)
    {
        var wiersze = products
            .Select(p => $"{p.Name};{p.Price}")
            .ToArray();

        return _staryCsv.ExportujDoCsv(wiersze);
    }
}

// Użycie – nasz kod nie wie, że pod spodem jest stara biblioteka:
IDataExporter exporter = new CsvExporterAdapter();
var csv = exporter.Export(products);
```

Observer – powiadamianie o zmianie

Problem:

Jeden obiekt zmienia stan, a wiele innych obiektów musi o tym wiedzieć i zareagować – ale nie chcemy ich ze sobą ściśle wiązać.

Rozwiązanie:

Obiekt (Subject) prowadzi listę obserwatorów i powiadamia ich automatycznie o zmianach.

Analogia:

Newsletter – zapisujesz się na listę, dostajesz powiadomienia. Wydawca nie musi wiedzieć kim jesteś.


```
// Subject – ten, kto powiadamia
public class Magazyn
{
    private int _stan;
    private List<IObserver> _observers = new();

    public void Subskrybuj(IObserver observer)
        => _observers.Add(observer);

    public void Wypisz(IObserver observer)
        => _observers.Remove(observer);

    public int Stan
    {
        get => _stan;
        set
        {
            _stan = value;
            // Powiadom wszystkich obserwatorów
            foreach (var obs in _observers)
                obs.Reaguj($"Stan magazynu: {_stan}");
        }
    }
}

public interface IObserver
{
    void Reaguj(string komunikat);
}

public class SklepInternetowy : IObserver
{
    public void Reaguj(string komunikat)
        => Console.WriteLine($"Sklep: {komunikat}");
}

// Użycie:
var magazyn = new Magazyn();
magazyn.Subskrybuj(new SklepInternetowy());
magazyn.Stan = 50;  // Sklep dostaje powiadomienie
```

Command – akcja jako obiekt

Problem:

Chcemy traktować akcje (operacje) jako obiekty – żeby je przekazywać, kolejkować, cofać, albo podpinać dynamicznie do UI.

Rozwiązanie:

Opakowujemy akcję w obiekt implementujący wspólny interfejs z metodą `Execute()`.

Analogia:

Pilot do telewizora – każdy przycisk to "komenda". Pilot nie wie jak działa telewizor, wie tylko że ma wcisnąć `Execute`. Można łatwo przeprogramować przycisk na inną akcję.

```
// Interfejs komendy
public interface ICommand
{
    void Execute();
    bool CanExecute();  // Czy komenda jest w ogóle dostępna?
}

// Konkretne komendy
public class SaveCommand : ICommand
{
    private readonly Document _doc;

    public SaveCommand(Document doc) => _doc = doc;

    public bool CanExecute() => _doc.HasChanges;
    public void Execute() => _doc.Save();
}

public class PrintCommand : ICommand
{
    private readonly Document _doc;

    public PrintCommand(Document doc) => _doc = doc;

    public bool CanExecute() => _doc.PageCount > 0;
    public void Execute() => _doc.Print();
}

// Użycie – przycisk nie wie CO robi, wie tylko ŻE ma wykonać komendę
ICommand command = new SaveCommand(document);
if (command.CanExecute())
    command.Execute();
```

Wzorce architektoniczne

- Wzorce projektowe (singleton, observer, factory) rozwiązują konkretne, małe problemy – jak stworzyć obiekt, jak powiadomić o zmianie.

- Istnieje również wyższy poziom – wzorce architektoniczne, które definiują całą strukturę aplikacji:
  - MVC – Model View Controller
  - MVP – Model View Presenter
  - MVVM – Model View ViewModel

- Wzorce architektoniczne pozwalają tak zorganizować kod aplikacji, aby był czytelny, testowalny oraz łatwy do utrzymania.

Wzorce architektoniczne

- Wyobraźmy sobie aplikację sklepu, bez korzystania z żadnego wzorca, code-behind będzie wyglądało mniej więcej tak:

```
private async void OnPageLoaded(object sender, EventArgs e)
{
    // Pobieranie danych
    var client = new HttpClient();
    var json = await client.GetStringAsync("https://api.sklep.pl/products");
    _products = JsonSerializer.Deserialize<List<Product>>(json);

    // Logika biznesowa
    _products = _products.Where(p => p.Price > 0 && p.InStock).ToList();
    _products = _products.OrderBy(p => p.Name).ToList();

    // Aktualizacja UI
    ProductsListView.ItemsSource = _products;
    CountLabel.Text = $"Znaleziono: {_products.Count}";
}
```

```
private void OnSearchTextChanged(object sender, TextChangedEventArgs e)
{
    // Filtrowanie + aktualizacja UI w jednym miejscu
    var filtered = _products
        .Where(p => p.Name.Contains(e.NewTextValue))
        .ToList();
    ProductsListView.ItemsSource = filtered;
    CountLabel.Text = $"Znaleziono: {filtered.Count}";
}
```

```
private async void OnDeleteClicked(object sender, EventArgs e)
{
    // Usuwanie + API + UI razem
    var btn = (Button)sender;
    var product = (Product)btn.BindingContext;
    var client = new HttpClient();
    await client.DeleteAsync($"https://api.sklep.pl/products/{product.Id}");
    _products.Remove(product);
    ProductsListView.ItemsSource = _products;
}
```

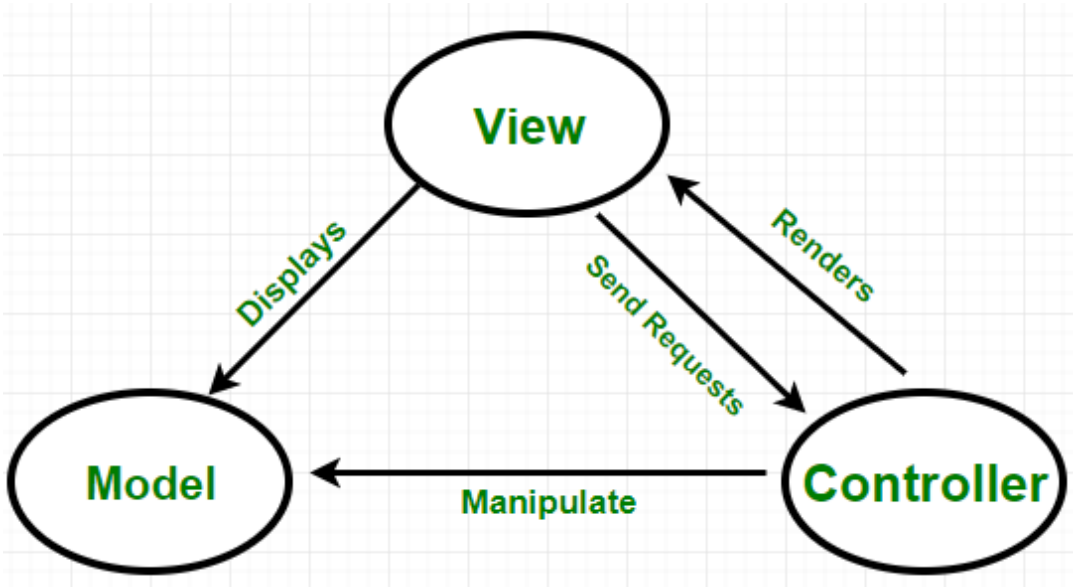
Kod na tych przykładach działa, jednakże jest on słabo testowalny, a tworzenie w ten sposób kilkunastu stron w aplikacji, będzie bardzo męczące...

Problemy tłustego code-behind

- Kod, z poprzedniego slajdu jest zły z wielu powodów:
  - **Brak separacji odpowiedzialności** – zmiana w API wymaga grzebania w kodzie UI.
  - **Nie da się testować jednostkowo** – żeby przetestować filtrowanie, musisz uruchomić całą aplikację.
  - **Duplikacja kodu** – druga strona też potrzebuje produktów? Kopiuj-wklej.
  - **Trudna praca zespołowa** – designer nie może ruszać XAML-a, bo jest spleciony z logiką.
  - **Skalowanie** – 3 strony? OK. 30 stron? No nie…
- Rozwiązaniem będzie podzielenie odpowiedzialności tak, aby każda warstwa kodu miała jedno, jasno określone zadanie. Tutaj z pomocą przychodzą wzorce architektoniczne.

MVC

- **Model** – dane i logika biznesowa (klasa Product, repozytorium, reguły walidacji).
- **View** – warstwa prezentacji – to co użytkownik widzi (HTML, XAML).
- **Controller** – "dyrektor" – odbiera akcje od użytkownika, prosi Model o dane, przekazuje je do View



Kluczowa zasada – View nie rozmawia z Modelem bezpośrednio, Controller jest pośrednikiem.

MVC

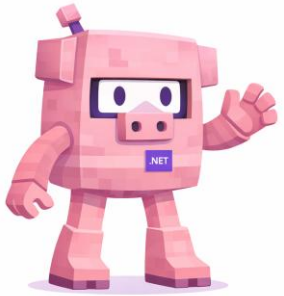
```cs
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}

public class ProductRepository
{
    public List<Product> GetAll() => /* pobierz z bazy */;
    public void Delete(int id) => /* usuń z bazy */;
}
```

Model

```cs
public class ProductController
{
    private ProductRepository _repo = new();

    [HttpGet]
    public IActionResult Products()
    {
        var products = _repo.GetAll();  // Pyta Model o dane
        return Ok(products);            // Każe View je wyświetlić
    }

    [HttpDelete]
    public IActionResult Product(int id)
    {
        _repo.Delete(id);               // Mówi Modelowi: usuń
        return Products();              // Odświeża widok
    }
}
```

Controller

MVC – gdzie go spotkamy?



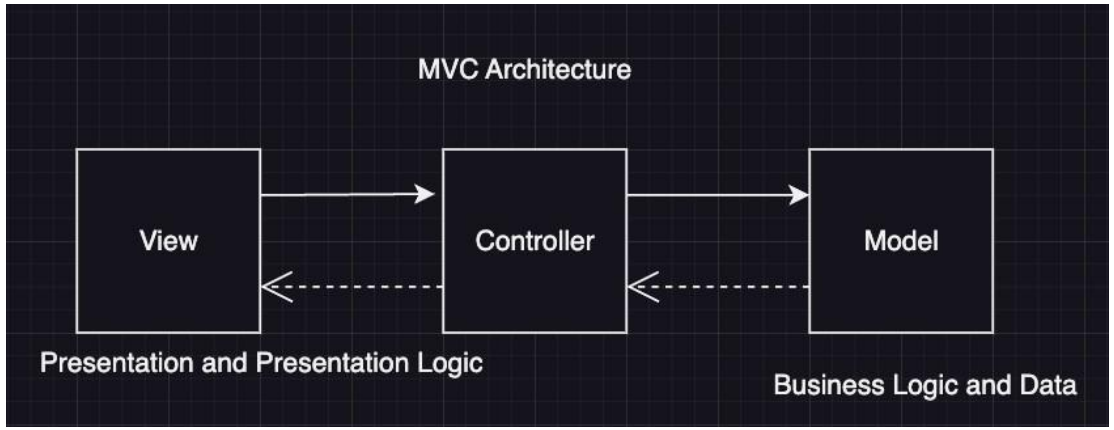
- Wzorzec MVC jest bardzo popularny, ale głównie w aplikacjach webowych:
  - ASP.NET Core (C#)
  - Ruby on Rails (Ruby)
  - Django (Python)
  - Spring (Java)
  - Laravel (PHP)

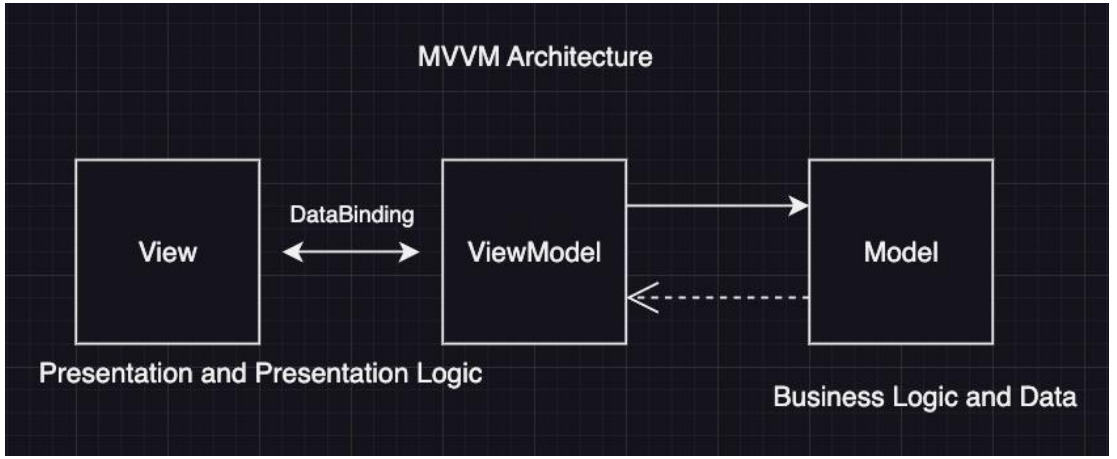
- W aplikacjach mobilnych i desktopowych MVC sprawdza się gorzej, ponieważ:
  - View w takiej aplikacji jest bardziej żywe – reaguje na gesty, animacje, zmiany orientacji.
  - Controller staje się zbyt duży.
  - Brakuje mechanizmu automatycznego powiadamiania View o zmianach w danych – Controller musiałby ręcznie aktualizować UI.

Dla aplikacji z bogatym UI powstał wzorzec MVVM, który rozwiązuje te problemy dzięki wiązaniu danych.

MVVM

- W MVC Controller ręcznie aktualizował View – mówił mu "wyświetl te dane". W MVVM tego nie ma. Zamiast Controllera pojawia się ViewModel, a zamiast ręcznej aktualizacji – data binding.





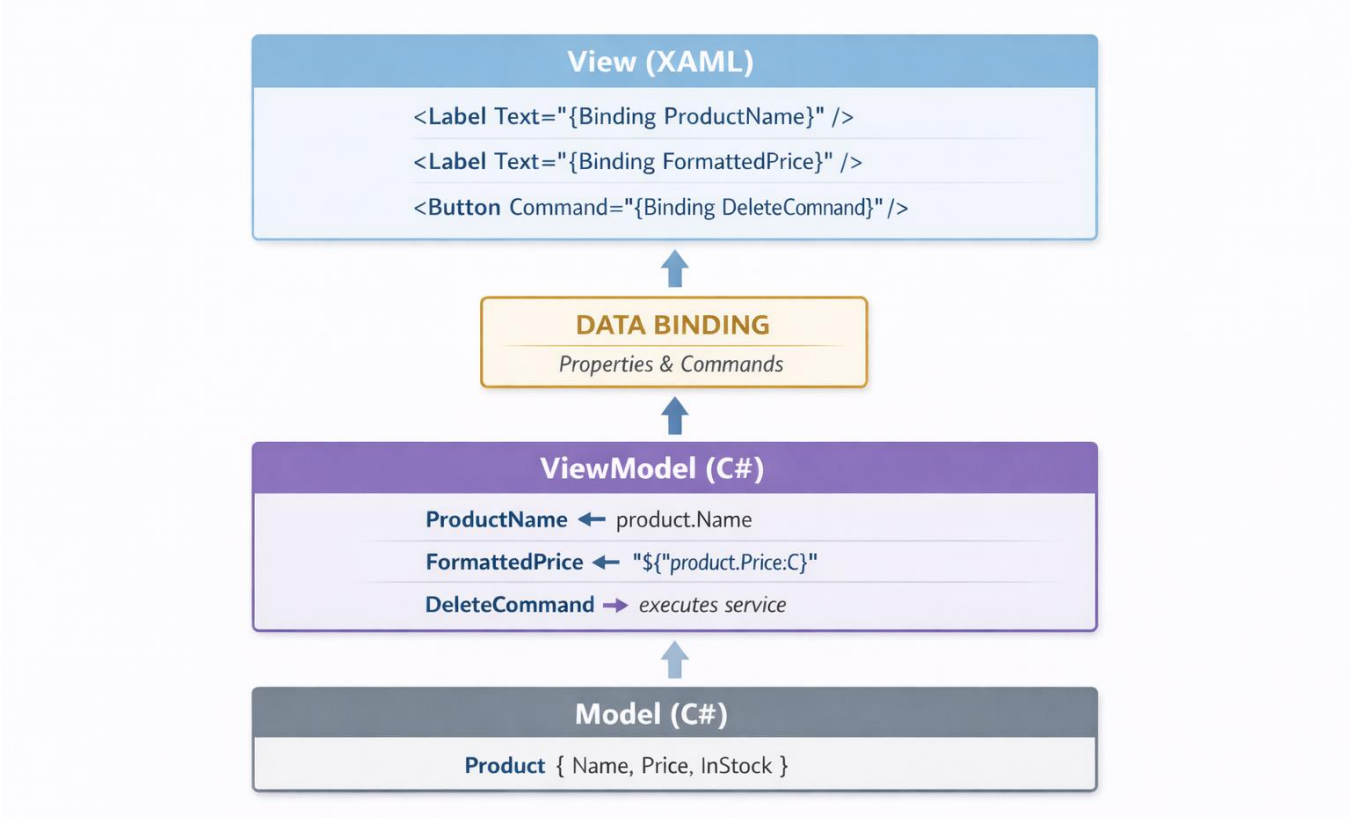
Kluczowa różnica – w MVVM View i ViewModel są połączone bindingiem – View automatycznie wie o zmianach w ViewModelu i odwrotnie. Nikt nie musi ręcznie przepisywać danych.

MVVM – trzy warstwy

- **Model:**
  - Dane i logika biznesowa.
  - Nie wie nic o UI.
  - Klasy typu `Product`, `User`, `Order`.
  - Może zawierać walidację, obliczenia, reguły biznesowe.
  - Nie zna View ani ViewModelu.
- **View:**
  - To co użytkownik widzi – strona XAML.
  - Wyświetla dane i zbiera akcje użytkownika.
  - Nie zawiera logiki biznesowej.
  - Łączy się z ViewModelem przez binding.
- **ViewModel:**
  - Pośrednik między View a Modelem.
  - Przygotowuje dane z Modelu w formie, którą View może wyświetlić.
  - Udostępnia właściwości (dane) i komendy (akcje).
  - Powiadamia View o zmianach przez `INotifyPropertyChanged`.

MVVM – przepływ danych

- View nie wie skąd dane pochodzą. ViewModel nie wie jak dane są wyświetlane. Model nie wie, że UI w ogóle istnieje. Każda warstwa robi swoją robotę.



Binding Context (bez DI)

- Żeby binding działał, View musi wiedzieć z jakim obiektem jest powiązany. To właśnie robi `BindingContext` – wskazuje ViewModel, z którego View czerpie dane.

```
public partial class ProductsPage : ContentPage
{
    public ProductsPage()
    {
        InitializeComponent();
        BindingContext = new ProductsViewModel();
    }
}
```

Ustawienie `BindingContext` w code-behind

```
<ContentPage xmlns:vm="clr-namespace:MojaApka.ViewModels">
    <ContentPage.BindingContext>
        <vm:ProductsViewModel />
    </ContentPage.BindingContext>

    <Label Text="{Binding ProductName}" />
</ContentPage>
```

Ustawianie `BindingContext` w XAML
(tylko jeżeli nie korzystamy z DI)

Zasada I: Typowo jeden ViewModel na jedną stronę. `BindingContext` ustawiamy raz, a wszystkie bindingi na stronie korzystają z tego samego ViewModelu.

Zasada II: Ustawiamy `BindingContext` tylko w jednym miejscu – albo w XAML, albo w C#.

Binding Context (Dependency Injection)

- Jeżeli korzystamy z zalecanego wzorca, czyli wstrzykiwania zależności, nieco inaczej ustawia się `BindingContext`.

```
// W code-behind przypisujemy obiekt z DI
public ProductsPage(ProductsViewModel vm)
{
    InitializeComponent();
    BindingContext = vm;
}
```

Ustawienie `BindingContext` w code-behind
(ViewModel jest wstrzykiwany)

```
<ContentPage
    xmlns:vm="clr-namespace:MojaApka.ViewModels"
    x:DataType="vm:ProductsViewModel">

  <!-- x:DataType sprawia, że XAML będzie miał silne typowanie -->

    <Label Text="{Binding ProductName}" />
</ContentPage>
```

Ustawienie typu obiektu `BindingContext` w XAML
(W XAML będzie obecne silne typowanie)

W tym przypadku musimy **ustawić `BindingContext` w C# oraz zadeklarować jego typ w XAML**.

INotifyPropertyChanged

- Aby UI automatycznie aktualizowało się o zmienione dane, ViewModel powinien o tym powiadamiać View. W tym celu stosuje się interfejs `INotifyPropertyChanged` – wzorzec Observer.
- Kiedy właściwość się zmienia, ViewModel "krzyczy" przez event `PropertyChanged`:

  "*Hej, właściwość X ma nową wartość!*" – a View nasłuchuje i automatycznie się aktualizuje.

```
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler? PropertyChanged;
}
```

INotifyPropertyChanged – implementacja

Przepływ:

- Coś zmienia `ProductName` w ViewModelu
- Setter wywołuje `OnPropertyChanged()`
- Event `PropertyChanged` leci do View
- View odczytuje nową wartość i aktualizuje `Label`

```
public class ProductsViewModel : INotifyPropertyChanged
{
    // Event wymagany przez interfejs
    public event PropertyChangedEventHandler? PropertyChanged;

    // Metoda pomocnicza – wywołuje event
    protected void OnPropertyChanged([CallerMemberName] string? name = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
    }

    // Pole prywatne
    private string _productName = string.Empty;

    // Właściwość publiczna z powiadomieniem
    public string ProductName
    {
        get => _productName;
        set
        {
            if (_productName != value)
            {
                _productName = value;
                OnPropertyChanged();  // "Hej View, ProductName się zmienił!"
            }
        }
    }
}
```

BaseViewModel

- Każdy ViewModel w aplikacji będzie potrzebował `INotifyPropertyChanged` i metody `OnPropertyChanged`. Kopiowanie tego do każdej klasy jest bez sensu.
- Znacznie lepszym pomysłem będzie stworzenie klasy bazowej, z której będą dziedziczyły wszystkie ViewModele.

```
public class BaseViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

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

    // Bonus: metoda SetProperty – jeszcze mniej powtórzeń
    protected bool SetProperty<T>(ref T field, T value,
                                  [CallerMemberName] string? name = null)
    {
        if (EqualityComparer<T>.Default.Equals(field, value))
            return false;

        field = value;
        OnPropertyChanged(name);
        return true;
    }
}
```

Klasa bazowa

```
public class ProductsViewModel : BaseViewModel
{
    private string _productName = string.Empty;

    public string ProductName
    {
        get => _productName;
        set => SetProperty(ref _productName, value);
    }
}
```

ViewModel

Adnotacja
Atrybut **[CallerMemberName]** pozwala **metodzie poznać nazwę metody lub właściwości**, z której została wywołana. Atrybut ten, nie jest refleksją – działa już na etapie kompilacji.

ObservableCollection<T>

- Kolekcja `ObservableCollection<T>` to specjalna kolekcja, która:
  - przechowuje listę elementów (*jak normalna kolekcja*),
  - automatycznie informuje UI, gdy:
    - element zostanie dodany,
    - element zostanie usunięty,
    - lista zostanie wyczyszczona,
    - kolejność elementów się zmieni.

Dzięki tej kolekcji interfejs użytkownika potrafi aktualizować się samodzielnie, bez konieczności odświeżania.

```xaml
<CollectionView ItemsSource="{Binding Products}">
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <Label Text="{Binding Name}" />
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

```csharp
public class ProductsViewModel
{
    public ObservableCollection<Product> Products { get; }
        = new ObservableCollection<Product>();

    public ProductsViewModel()
    {
        Products.Add(new Product { Name = "Laptop" });
        Products.Add(new Product { Name = "Phone" });
    }
}
```

`ObservableCollection<T>`

- `ObservableCollection<T>` powiadamia o zmianach w kolekcji (*dodanie, usunięcie elementu*).
- Nie powiadamia o zmianach wewnątrz elementu (*np. zmiana `product.Name`*).
  - Jeżeli UI ma reagować na zmiany wewnątrz elementu, klasa `Product` też musi implementować `INotifyPropertyChanged`.

```
Products[0].Name = "New name"; // UI się NIE zmieni
```

ICommand

- MVVM:
  - View nie może implementować żadnej logiki.
  - ViewModel nie ma dostępu do żadnych kontrolek.

- W podejściu MVVM stosuje się wzorzec Command – akcja jako obiekt. Przycisk nie wie co robi, wie tylko, że ma wykonać komendę. ViewModel decyduje co się stanie po jej wykonaniu.

```
// Interfejs ICommand (wbudowany w .NET)
public interface ICommand
{
    void Execute(object? parameter);
    bool CanExecute(object? parameter);
    event EventHandler? CanExecuteChanged;
}
```

```
<Button Text="Usuń" Command="{Binding DeleteCommand}" />
```

RelayCommand

- Implementacja `ICommand` od zera za każdym razem, byłaby dość uciążliwa. Dlatego często tworzy się uniwersalną klasę `RelayCommand`.

```
public class RelayCommand : ICommand
{
    private readonly Action _execute;
    private readonly Func<bool>? _canExecute;

    public RelayCommand(Action execute, Func<bool>? canExecute = null)
    {
        _execute = execute;
        _canExecute = canExecute;
    }

    public event EventHandler? CanExecuteChanged;

    public bool CanExecute(object? parameter)
        => _canExecute?.Invoke() ?? true;

    public void Execute(object? parameter)
        => _execute();

    // Wywołaj, gdy zmienił się warunek CanExecute
    public void RaiseCanExecuteChanged()
        => CanExecuteChanged?.Invoke(this, EventArgs.Empty);
}
```

ICommand w ViewModelu

- Przy odpowiedniej implementacji interfejsu `ICommand` w MVVM, możemy bardzo prosto zaprogramować akcje UI.
- Kod na poniższym przykładzie sprawia, że przyicsk „Dodaj” staje się automatycznie nieaktywny, gdy pole „Nazwa produktu” jest puste. Gdy użytkownik wpisze coś w tym polu, przycisk się automatycznie aktywuje.

```
public class ProductsViewModel : BaseViewModel
{
    public ObservableCollection<Product> Products { get; } = new();

    private string _newProductName = string.Empty;
    public string NewProductName
    {
        get => _newProductName;
        set
        {
            SetProperty(ref _newProductName, value);
            // Gdy nazwa się zmienia, sprawdź czy przycisk powinien być aktywny
            AddCommand.RaiseCanExecuteChanged();
        }
    }

    public RelayCommand AddCommand { get; }

    public ProductsViewModel()
    {
        AddCommand = new RelayCommand(
            execute: () =>
            {
                Products.Add(new Product { Name = NewProductName });
                NewProductName = string.Empty;
            },
            canExecute: () => !string.IsNullOrWhiteSpace(NewProductName)
        );
    }
}
```


```
<Entry Text="{Binding NewProductName}" Placeholder="Nazwa produktu" />
<Button Text="Dodaj" Command="{Binding AddCommand}" />
```

CommandParameter

- Chcemy usunąć wybrany element z listy. W tym celu, musimy przekazać komendzie informacje, które pozwolą na zidentyfikowanie tego elementu.

```xaml
<CollectionView ItemsSource="{Binding Products}">
    <CollectionView.ItemTemplate>
        <DataTemplate x:DataType="models:Product">
            <HorizontalStackLayout Spacing="10" Padding="5">
                <Label Text="{Binding Name}" VerticalOptions="Center" />

                <Button Text="Usuń"
                        Command="{Binding Source={RelativeSource
                            AncestorType={x:Type vm:ProductsViewModel}},
                            Path=DeleteCommand}"
                        CommandParameter="{Binding .}" />
            </HorizontalStackLayout>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

```csharp
// RelayCommand z parametrem
public RelayCommand<Product> DeleteCommand { get; }

public ProductsViewModel()
{
    DeleteCommand = new RelayCommand<Product>(
        execute: product => Products.Remove(product)
    );
}
```

- **Co tu się dzieje?**
  - **`Command`** – wskazuje komendę z ViewModelu.
  - **`RelativeSource`** + **`AncestorType`** – pozwala dotrzeć do `ProductsView`.
  - **`ModelCommandParameter="{Binding .}"`** – przekazuje aktualny obiekt `Product`.

CommandParameter

- Jak to dokładnie działa?
  1. Użytkownik klika „Usuń” przy konkretnym produkcie.
  2. MAUI wywołuje `DeleteCommand` z `ProductsViewModel`.
  3. Do komendy trafia kliknięty `Product` jako parametr.
  4. Produkt zostaje usunięty z `ObservableCollection`.
  5. UI aktualizuje się automatycznie.

Wstrzykiwanie zależności (DI)

- Dependency Injection (DI) to mechanizm, w którym obiekt nie tworzy sam swoich zależności, ale dostaje je z zewnątrz – najczęściej przez konstruktor. Centralny kontener DI zarządza tworzeniem obiektów i ich czasem życia.

```
// Wariant I - bez DI
public class ProductsViewModel
{
    // ViewModel SAM tworzy serwis – ścisłe powiązanie
    private readonly ProductService _service = new ProductService();
}
```

```
// Wariant II - dependency injection
public class ProductsViewModel
{
    private readonly IProductService _service;

    // Serwis przychodzi z zewnątrz – luźne powiązanie
    public ProductsViewModel(IProductService service)
    {
        _service = service;
    }
}
```

ViewModel nie wie jaką implementację dostaje – wie tylko, że dostaje coś, co implementuje dany interfejs. Można łatwo podmienić implementację (*np. na mock w testach*) bez zmiany ViewModelu.

Rejestracja serwisów

- W .NET MAUI kontener DI konfigurujemy w pliku `MauiProgram.cs`. Tu rejestrujemy wszystko – serwisy, ViewModele, strony. Kontener sam tworzy obiekty i wstrzykuje zależności.


```
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder.UseMauiApp<App>();

        // Rejestracja serwisów
        builder.Services.AddSingleton<IProductService, ProductService>();

        // Rejestracja ViewModeli
        builder.Services.AddTransient<ProductsViewModel>();
        builder.Services.AddTransient<ProductDetailViewModel>();

        // Rejestracja stron
        builder.Services.AddTransient<ProductsPage>();
        builder.Services.AddTransient<ProductDetailPage>();

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

Kiedy MAUI tworzy `ProductsPage`, widzi że jej konstruktor wymaga `ProductsViewModel`. Szuka go w kontenerze, widzi że `ProductsViewModel` wymaga `IProductService` – tworzy `ProductService` i przekazuje dalej. Cały łańcuch zależności rozwiązuje się automatycznie.

Cykl życia obiektu

- Przy rejestracji obiektu w kontenerze DI, decydujemy jaki będzie jego cykl życia:
  - Singleton – jeden `ProductService` na całą aplikację. Pobiera dane raz, trzyma w pamięci, wszystkie strony widzą te same produkty.
  - Transient – za każdym razem gdy otwierasz stronę, dostajesz nowy, czysty ViewModel. Żadnych pozostałości z poprzedniego otwarcia.
  - Scoped – jeden obiekt żyje w ramach zdefiniowanego zakresu. W aplikacjach webowych (*ASP.NET*) scope to jedno żądanie HTTP – naturalny i często używany. W MAUI nie ma takiego naturalnego scope'a, więc używa się go rzadko.
    - **Przykład** – proces składania zamówienia rozłożony na kilka stron (koszyk, adres dostawy, płatność) – jeden `OrderService` współdzielony przez te strony, ale nowy dla każdego kolejnego zamówienia. Wymaga ręcznego tworzenia scope'a przez `IServiceScopeFactory`.

- Praktycznie w MAUI:
  - Serwisy jako Singleton.
  - ViewModele i strony jako Transient.

Metoda	Cykl życia	Kiedy stosować
AddSingleton<T>	Jeden obiekt na całą aplikację.	Serwisy współdzielące stan – ustawienia, cache, `HttpClient`.
AddTransient<T>	Nowy obiekt za każdym razem.	ViewModele, strony – świeży stan, przy każdym otwarciu.
AddScoped<T>	Jeden obiekt w ramach zakresu.	Operacja wymagające wspólnego kontekstu w ograniczonym zakresie.

Wstrzykiwanie przez konstruktor

- Kiedy zarejestrujemy serwisy, ViewModele i strony w kontenerze DI, wstrzykiwanie odbywa się automatycznie przez konstruktory. Każda klasa deklaruje czego potrzebuje – kontener dostarcza.

Serwis

```
public interface IProductService
{
    List<Product> GetAll();
    Product GetById(int id);
}

public class ProductService : IProductService
{
    public List<Product> GetAll() => /* pobierz dane */;
    public Product GetById(int id) => /* pobierz produkt */;
}
```

Strona (*przyjmuje ViewModel*)

```
public partial class ProductsPage : ContentPage
{
    public ProductsPage(ProductsViewModel viewModel)
    {
        InitializeComponent();
        BindingContext = viewModel;
    }
}
```

2

1

ViewModel (*przyjmuje serwis*)

```
public partial class ProductsViewModel : ObservableObject
{
    private readonly IProductService _service;

    public ProductsViewModel(IProductService service)
    {
        _service = service;
    }

    [RelayCommand]
    private void LoadProducts()
    {
        var products = _service.GetAll();
        // ...
    }
}
```

Aplikacja MVVM – ToDo List

- Spróbujmy zaplanować projekt aplikacji „ToDo List” pod kątem tego, gdzie umieścimy poszczególne elementy logiki – będziemy operować na wzorcu MVVM.

- Wymagania:
  - Proste modele zadań – tytuł oraz informacja o tym, czy już zostało wykonane.
  - Możliwość tworzenia i usuwania zadań.
  - Wyświetlanie wszystkich zadań.

Teraz musimy ustalić, w których miejscach MVVM umieścimy poszczególne elementy logiki.

Aplikacja MVVM – ToDo List

Model

```
// Models/TodoItem.cs
public class TodoItem
{
    public string Title { get; set; } = string.Empty;
    public bool IsDone { get; set; }
}
```

Aplikacja MVVM – ToDo List

ViewModel

```
// ViewModels/TodoViewModel.cs
public class TodoViewModel : BaseViewModel
{
    public ObservableCollection<TodoItem> Tasks { get; } = new();

    private string _newTaskTitle = string.Empty;
    public string NewTaskTitle
    {
        get => _newTaskTitle;
        set
        {
            SetProperty(ref _newTaskTitle, value);
            AddTaskCommand.RaiseCanExecuteChanged();
        }
    }

    public RelayCommand AddTaskCommand { get; }
    public RelayCommand<TodoItem> DeleteTaskCommand { get; }

    public TodoViewModel()
    {
        AddTaskCommand = new RelayCommand(
            execute: () =>
            {
                Tasks.Add(new TodoItem { Title = NewTaskTitle });
                NewTaskTitle = string.Empty;
            },
            canExecute: () => !string.IsNullOrWhiteSpace(NewTaskTitle)
        );

        DeleteTaskCommand = new RelayCommand<TodoItem>(
            execute: task => Tasks.Remove(task)
        );
    }
}
```

Aplikacja MVVM – ToDo List

View

```xaml
<!-- Views/TodoPage.xaml -->
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:vm="clr-namespace:MojaApka.ViewModels"
             xmlns:models="clr-namespace:MojaApka.Models"
             x:Class="MojaApka.Views.TodoPage">

    <ContentPage.BindingContext>
        <vm:TodoViewModel />
    </ContentPage.BindingContext>

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

        <Label Text="Lista zadań" FontSize="24" FontAttributes="Bold" />

        <HorizontalStackLayout Spacing="10">
            <Entry Text="{Binding NewTaskTitle}"
                   Placeholder="Nowe zadanie..."
                   HorizontalOptions="FillAndExpand" />
            <Button Text="Dodaj" Command="{Binding AddTaskCommand}" />
        </HorizontalStackLayout>

        <CollectionView ItemsSource="{Binding Tasks}">
            <CollectionView.ItemTemplate>
                <DataTemplate x:DataType="models:TodoItem">
                    <HorizontalStackLayout Spacing="10" Padding="5">
                        <CheckBox IsChecked="{Binding IsDone}" />
                        <Label Text="{Binding Title}"
                               VerticalOptions="Center" />
                        <Button Text="✕"
                                Command="{Binding Source={RelativeSource
                                    AncestorType={x:Type vm:TodoViewModel}},
                                    Path=DeleteTaskCommand}"
                                CommandParameter="{Binding .}" />
                    </HorizontalStackLayout>
                </DataTemplate>
            </CollectionView.ItemTemplate>
        </CollectionView>

    </VerticalStackLayout>
</ContentPage>
```

Aplikacja MVVM – ToDo List

Opis przepływu

- Dodawanie zadania:
  1. Użytkownik wpisuje tekst w `Entry` – binding aktualizuje `NewTaskTitle` w VM.
  2. Setter `NewTaskTitle` wywołuje `RaiseCanExecuteChanged()` – przycisk "Dodaj" staje się aktywny.
  3. Użytkownik klika "Dodaj" – binding wywołuje `AddTaskCommand.Execute()`.
  4. Komenda tworzy nowy `TodoItem` i dodaje do Tasks (`ObservableCollection`).
  5. `ObservableCollection` powiadamia `CollectionView` – nowy element pojawia się na liście.
  6. Komenda czyści `NewTaskTitle` – `Entry` się opróżnia przez binding.

- Usuwanie zadania:
  1. Użytkownik klika „X" przy zadaniu – `DeleteTaskCommand.Execute(todoItem)`.
  2. `CommandParameter` przekazuje konkretny obiekt `TodoItem`.
  3. Komenda wywołuje `Tasks.Remove(task)` – element znika z listy.

W code-behind nie ma żadnego kodu. Plik `TodoPage.xaml.cs` zawiera tylko konstruktor z metodą `InitializeComponent()` – nic więcej.

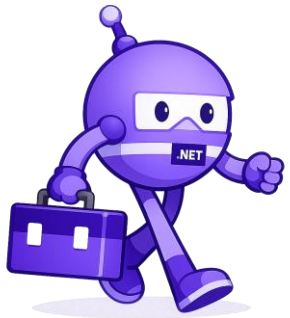
Community Toolkit MVVM

- Tworząc od zera kod w MVVM bez narzędzi pomocniczych, musimy:
  - zawsze ręcznie implementować `INotifyPropertyChanged`,
  - ręcznie wywoływać `OnPropertyChanged`,
  - pisać bardzo dużo powtarzalnego kodu,
  - tworzyć własne klasy `Command`.

Przy takim podejściu kod będzie długi, mało czytelny i podatny na błędy.

- CommunityToolkit.Mvvm to oficjalna biblioteka Microsoftu, która:
  - upraszcza tworzenie ViewModeli,
  - eliminuje powtarzalny kod,
  - zapewnia gotowe mechanizmy MVVM,
  - jest w pełni kompatybilna z .NET MAUI.

NuGet Gallery | CommunityToolkit.Mvvm 8.4.0



Community Toolkit MVVM

`ObservableProperty`

Pure

```
public class TodoViewModel : BaseViewModel
{
    private string _newTaskTitle = string.Empty;
    public string NewTaskTitle
    {
        get => _newTaskTitle;
        set => SetProperty(ref _newTaskTitle, value);
    }

    private bool _isBusy;
    public bool IsBusy
    {
        get => _isBusy;
        set => SetProperty(ref _isBusy, value);
    }
}
```

CommunityToolkit.Mvvm

```
public partial class TodoViewModel : ObservableObject
{
    [ObservableProperty]
    private string _newTaskTitle = string.Empty;

    [ObservableProperty]
    private bool _isBusy;
}
```

Ważne!
Klasa musi być partial i dziedziczyć po `ObservableObject` (odpowiednik naszego `BaseViewModel`).

Co tutaj się dzieje?
Source generator **w czasie kompilacji** generuje pełną właściwość `NewTaskTitle` z getterem, setterem, porównaniem i wywołaniem `OnPropertyChanged`. Ty piszesz jedno pole + atrybut.

Community Toolkit MVVM

RelayCommand

Pure

```
public RelayCommand AddTaskCommand { get; }

public TodoViewModel()
{
    AddTaskCommand = new RelayCommand(
        execute: () =>
        {
            Tasks.Add(new TodoItem { Title = NewTaskTitle });
            NewTaskTitle = string.Empty;
        },
        canExecute: () => !string.IsNullOrWhiteSpace(NewTaskTitle)
    );
}
```

CommunityToolkit.Mvvm

```
// Komenda bez parametru
[RelayCommand(CanExecute = nameof(CanAddTask))]
private void AddTask()
{
    Tasks.Add(new TodoItem { Title = NewTaskTitle });
    NewTaskTitle = string.Empty;
}

// Komenda z parametrem
[RelayCommand]
private void DeleteTask(TodoItem task) => Tasks.Remove(task);
// Generuje: DeleteTaskCommand typu IRelayCommand<TodoItem>

private bool CanAddTask()
 =>!string.IsNullOrWhiteSpace(NewTaskTitle);
```

Co tutaj się dzieje?
Source generator tworzy właściwość `AddTaskCommand` typu `IRelayCommand`, opakowuje metodę `AddTask` w obiekt komendy, a `CanAddTask` podpina pod `CanExecute`.

Community Toolkit MVVM

`ObservableObject`

Pure

```
public class BaseViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

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

    // Bonus: metoda SetProperty – jeszcze mniej powtórzeń
    protected bool SetProperty<T>(ref T field, T value,
                                  [CallerMemberName] string? name = null)
    {
        if (EqualityComparer<T>.Default.Equals(field, value))
            return false;

        field = value;
        OnPropertyChanged(name);
        return true;
    }
}
```

CommunityToolkit.Mvvm

```
public partial class ProductViewModel : ObservableObject
{
    // 1. [ObservableProperty] – już znamy
    [ObservableProperty]
    private string _name = string.Empty;

    // 2. SetProperty – działa jak w naszym BaseViewModel
    private decimal _price;
    public decimal Price
    {
        get => _price;
        set => SetProperty(ref _price, value);
    }

    // 3. OnPropertyChanged – można wywoływać ręcznie
    public string FormattedPrice => $"{Price:C}";

    // Gdy Price się zmieni, powiadom też o FormattedPrice
    partial void OnPriceChanged(decimal value)
    {
        OnPropertyChanged(nameof(FormattedPrice));
    }
}
```

Co tutaj się dzieje?

Dla każdego [`ObservableProperty`] o nazwie `_name` **source generator tworzy**:

- Właściwość publiczną `Name` (z getterem, setterem, powiadomieniem).
- Metodę `partial void OnNameChanging(string value)` – przed zmianą.
- Metodę `partial void OnNameChanged(string value)` – po zmianie.

Community Toolkit MVVM

AsyncRelayCommand

Pure

```
// ❌ Tak NIE robimy – async void to zło (wyjątki giną w tle)
[RelayCommand]
private async void LoadProducts() { ... }
```

Co daje AsyncRelayCommand?

Funkcja	Opis
IsRunning	Automatyczna flaga true podczas wykonywania.
Blokada pojedynczego kliknięcia	Komenda jest nieaktywna dopóki Task się nie zakończy.
Obsługa wyjątków	Wyjątki nie giną – można je przechwycić.
CancellationToken	Można anulować operację.

Ważne!
Toolkit automatycznie też wygeneruje LoadProducts**AsyncCancelCommand**, który można podpiąć pod przycisk „Anuluj”.

CommunityToolkit.Mvvm

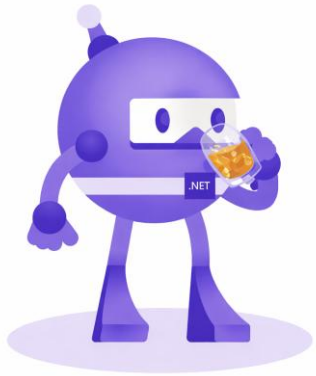
```
public partial class ProductsViewModel : ObservableObject
{
    public ObservableCollection<Product> Products { get; } = new();

    [ObservableProperty]
    private bool _isBusy;

    [RelayCommand]
    private async Task LoadProductsAsync()
    {
        IsBusy = true;
        try
        {
            var client = new HttpClient();
            var json = await client.GetStringAsync("https://api.sklep.pl/products");
            var products = JsonSerializer.Deserialize<List<Product>>(json);

            Products.Clear();
            foreach (var p in products)
                Products.Add(p);
        }
        finally
        {
            IsBusy = false;
        }
    }
}
```

```
<Button Text="Odśwież" Command="{Binding LoadProductsAsyncCommand}" />
<ActivityIndicator IsRunning="{Binding IsBusy}" IsVisible="{Binding IsBusy}" />
```

Nawigacja w MVVM

- MVVM charakteryzuje się tym, że logika jest w ViewModelu, a ViewModel nie powinien znać stron. Mimo to nawigacja to decyzja logiczna – *użytkownik kliknął na produkt -> przejdź do szczegółów*, więc powinna być w ViewModelu.
- W .NET MAUI klasa Shell w sposób bardzo elegancki, rozwiązuje ten problem – nawigacja odbywa się przez URI, nie przez tworzenie obiektów stron. ViewModel nie tworzy strony, on tylko mówi „*idź pod ten adres*".
- ViewModel wie dokąd nawigować, ale nie wie jak wygląda strona, ani jak jest zbudowana.

```
// W ViewModelu – nie tworzymy strony, podajemy ścieżkę
await Shell.Current.GoToAsync("ProductDetailPage");
```



Nawigacja w MVVM – parametry

- Nawigacja w MVVM wygląda podobnie, jak w przypadku braku tego wzorca. Kluczową zmianą jest to, że teraz parametry wysyła jeden ViewModel, a odbiera inny ViewModel.
- Zauważ też, że można przekazywać wiele parametrów (*nie tylko stringi*), jeżeli zamiast query stringa, wyślemy obiekt `Dictionary<string, object>`.

```
await Shell.Current.GoToAsync("ProductDetailPage", new Dictionary<string, object>
{
    { "Product", product },
    { "IsEditing", true }
});
```

Wysyłanie parametrów

```
[QueryProperty(nameof(Product), "Product")]
[QueryProperty(nameof(IsEditing), "IsEditing")]
public partial class ProductDetailViewModel : ObservableObject
{
    [ObservableProperty]
    private Product _product;

    [ObservableProperty]
    private bool _isEditing;
}
```

Odbieranie parametrów

Uwaga!
Aby ViewModele mogły odbierać parametry nawigacji z `Shell`, dany **ViewModel musi być ustawiony jako** `BindingContext` na danej stronie.

Behaviors

`Behavior` to mechanizm pozwalający dodać zachowanie do kontrolki bez modyfikowania jej kodu i bez dziedziczenia. Behavior tworzymy jako osobną klasę, a potem dołączamy do dowolnej kontrolki w XAML.

- Behaviors umożliwiają:
  - Walidację pól formularza (np. zmiana koloru gdy pole puste)
  - Ograniczanie wejścia (np. tylko cyfry w `Entry`)
  - Formatowanie tekstu w czasie rzeczywistym
  - Reagowanie na zdarzenia kontrolki bez code-behind
  - Reużywanie tego samego zachowania na wielu kontrolkach i stronach

```xaml
<!-- Ten sam Behavior na różnych kontrolkach, różnych stronach -->
<Entry Placeholder="Imię...">
    <Entry.Behaviors>
        <local:RequiredFieldBehavior />
    </Entry.Behaviors>
</Entry>

<Entry Placeholder="Email...">
    <Entry.Behaviors>
        <local:RequiredFieldBehavior />
    </Entry.Behaviors>
</Entry>
```

Behavior działa jak naklejka – **przyklejasz go do kontrolki, a on robi swoją robotę**. Kontrolka nie musi o nim wiedzieć.

Behavior<T> – budowa własnego zachowania

- Własny `Behavior` tworzymy dziedzicząc po klasie `Behavior<T>`, gdzie `T` to typ kontrolki, do której `Behavior` będzie dołączany. Klasa udostępnia dwie metody do nadpisania:
  - **`OnAttachedTo`** – wywoływana gdy Behavior zostaje dołączony do kontrolki. Tu podpinamy się do zdarzeń.
  - **`OnDetachingFrom`** – wywoływana gdy Behavior jest odłączany. Tu odpinamy się od zdarzeń (ważne, żeby uniknąć wycieków pamięci).

```xaml
<Entry Placeholder="Tylko cyfry...">
    <Entry.Behaviors>
        <local:NumericOnlyBehavior />
    </Entry.Behaviors>
</Entry>
```

Teraz każde `Entry` z tym zachowaniem będzie przyjmowało jedynie wartości liczbowej – i nie była do tego potrzebna nawet jedna linijka w code-behind.

```csharp
public class NumericOnlyBehavior : Behavior<Entry>
{
    protected override void OnAttachedTo(Entry entry)
    {
        base.OnAttachedTo(entry);
        entry.TextChanged += OnTextChanged;
    }

    protected override void OnDetachingFrom(Entry entry)
    {
        base.OnDetachingFrom(entry);
        entry.TextChanged -= OnTextChanged;
    }

    private void OnTextChanged(object? sender, TextChangedEventArgs e)
    {
        if (sender is Entry entry && !string.IsNullOrEmpty(e.NewTextValue))
        {
            // Jeśli nowy tekst nie jest liczbą – przywróć stary
            if (!double.TryParse(e.NewTextValue, out _))
                entry.Text = e.OldTextValue;
        }
    }
}
```

Behavior<T> – budowa własnego zachowania

Chcielibyśmy aby użytkownik widział, że pole jest wymagane, poprzez zmianę koloru tła jeżeli użytkownik zostawi je puste.

```
public class RequiredFieldBehavior : Behavior<Entry>
{
    protected override void OnAttachedTo(Entry entry)
    {
        base.OnAttachedTo(entry);
        entry.TextChanged += OnTextChanged;
        entry.Unfocused += OnUnfocused;
    }

    protected override void OnDetachingFrom(Entry entry)
    {
        base.OnDetachingFrom(entry);
        entry.TextChanged -= OnTextChanged;
        entry.Unfocused -= OnUnfocused;
    }

    private void OnTextChanged(object? sender, TextChangedEventArgs e)
    {
        if (sender is Entry entry)
            Validate(entry);
    }

    private void OnUnfocused(object? sender, FocusEventArgs e)
    {
        if (sender is Entry entry)
            Validate(entry);
    }

    private void Validate(Entry entry)
    {
        entry.BackgroundColor = string.IsNullOrWhiteSpace(entry.Text)
            ? Colors.LightPink
            : Colors.Transparent;
    }
}
```

```
<Entry Placeholder="Imię (wymagane)">
    <Entry.Behaviors>
        <local:RequiredFieldBehavior />
    </Entry.Behaviors>
</Entry>

<Entry Placeholder="Nazwisko (wymagane)">
    <Entry.Behaviors>
        <local:RequiredFieldBehavior />
    </Entry.Behaviors>
</Entry>
```

Jedno zachowanie, wiele pól – zachowanie jest w pełni reużywalne. Reaguje zarówno na zmianę tekstu, jak i na opuszczenie pola.

EventToCommandBehavior

- Nie każda kontrolka ma właściwość Command. Na przykład `Entry` ma zdarzenie `TextChanged`, ale nie ma `TextChangedCommand`. W MVVM chcielibyśmy podpiąć to zdarzenie do komendy w ViewModelu – bez pisania code-behind.
- `EventToCommandBehavior` łączy dowolne zdarzenie kontrolki z komendą w ViewModelu.

```xaml
<Entry Placeholder="Szukaj...">
    <Entry.Behaviors>
        <toolkit:EventToCommandBehavior
            EventName="TextChanged"
            Command="{Binding SearchCommand}"
            CommandParameter="{Binding Text, Source={RelativeSource Self}}" />
    </Entry.Behaviors>
</Entry>
```

```csharp
// W ViewModelu:
[RelayCommand]
private void Search(string query)
{
    // Filtruj listę na podstawie wpisanego tekstu
    var filtered = _allProducts
        .Where(p => p.Name.Contains(query, StringComparison.OrdinalIgnoreCase))
        .ToList();

    Products.Clear();
    foreach (var p in filtered)
        Products.Add(p);
}
```

`EventToCommandBehavior` jest dostępny w paczce Community Toolkit MAUI – nie trzeba go pisać samodzielnie. Działa z każdym zdarzeniem dowolnej kontrolki.

Zachowania z CommunityToolkit.Maui

- Paczka `CommunityToolkit.Maui` zawiera zestaw gotowych zachowań, które pokrywają najczęstsze scenariusze. Wystarczy zainstalować pakiet NuGet i zarejestrować w `MauiProgram.cs`.

```
using CommunityToolkit.Maui;

public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>()

            // Inicjalizacja .NET MAUI Community Toolkit
            .UseMauiCommunityToolkit()

            .ConfigureFonts(fonts =>
            {
                fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
                fonts.AddFont("OpenSans-Semibold.ttf", "OpenSansSemibold");
            });

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

```
<!-- Importowanie przestrzeni nazw w XAML -->
xmlns:toolkit="http://schemas.microsoft.com/dotnet/2022/maui/toolkit"
```

Zwróć uwagę, że mówimy teraz o pakiecie **`CommunityToolkit.Maui`**, nie o pakiecie z bibliotekami MVVM.

UserStoppedTypingBehavior

- Zachowanie sprawia, że `Entry` wykonuje komendę dopiero gdy użytkownik przestanie pisać (np. po 500ms). Idealne do wyszukiwania – nie odpytujemy API po każdym znaku.

```
<Entry Placeholder="Szukaj produktu...">
    <Entry.Behaviors>
        <toolkit:UserStoppedTypingBehavior
            StoppedTypingTimeThreshold="500"
            Command="{Binding SearchCommand}"
            ShouldDismissKeyboardAutomatically="True" />
    </Entry.Behaviors>
</Entry>
```

StatusBarBehavior

- Zachowanie, które zmienia kolor paska statusu (góra ekranu) na danej stronie.

```
<ContentPage>
    <ContentPage.Behaviors>
        <toolkit:StatusBarBehavior StatusBarColor="#1E90FF" StatusBarStyle="LightContent" />
    </ContentPage.Behaviors>
</ContentPage>
```

IconTintColorBehavior

- Zachowanie zmieniające kolor ikony (np. przyciemnienie w zależności od stanu).

```xaml
<Image Source="heart.png">
    <Image.Behaviors>
        <toolkit:IconTintColorBehavior TintColor="Red" />
    </Image.Behaviors>
</Image>
```

Zauważ, że do obsługi tych zachowań, **wystarczy jedynie kilka linii XAML** – nie musisz pisać dziesiątek linii C#.

Zasady dotyczące zachowań

- Zawsze odpinaj zdarzenia w `OnDetachingFrom` – inaczej ryzykujesz wycieki pamięci.
- `Behavior` powinien robić jedną rzecz – walidacja to jeden `Behavior`, formatowanie to drugi.
- `Behavior` nie powinien znać ViewModelu – działa na poziomie kontrolki.
- Jeśli potrzebujesz komunikacji z ViewModelem, użyj `EventToCommandBehavior` lub bindowalnych właściwości

Tryb aplikacji w .NET MAUI

- .NET MAUI automatycznie wykrywa systemowy tryb wyświetlania (jasny/ciemny) ustawiony przez użytkownika na urządzeniu. Aplikacja może na to reagować na trzy sposoby:
  - Podążać za systemem – domyślne zachowanie, aplikacja automatycznie przełącza się między trybami.
  - Wymusić konkretny tryb – aplikacja zawsze jasna lub zawsze ciemna, niezależnie od ustawień systemu.
  - Pozwolić użytkownikowi wybrać – przełącznik w ustawieniach aplikacji.

```
// Podążaj za systemem (domyślne)
Application.Current.UserAppTheme = AppTheme.Unspecified;

// Zawsze jasny
Application.Current.UserAppTheme = AppTheme.Light;

// Zawsze ciemny
Application.Current.UserAppTheme = AppTheme.Dark;
```

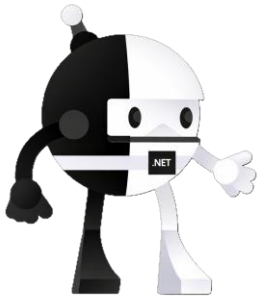
Ustawianie trybu aplikacji

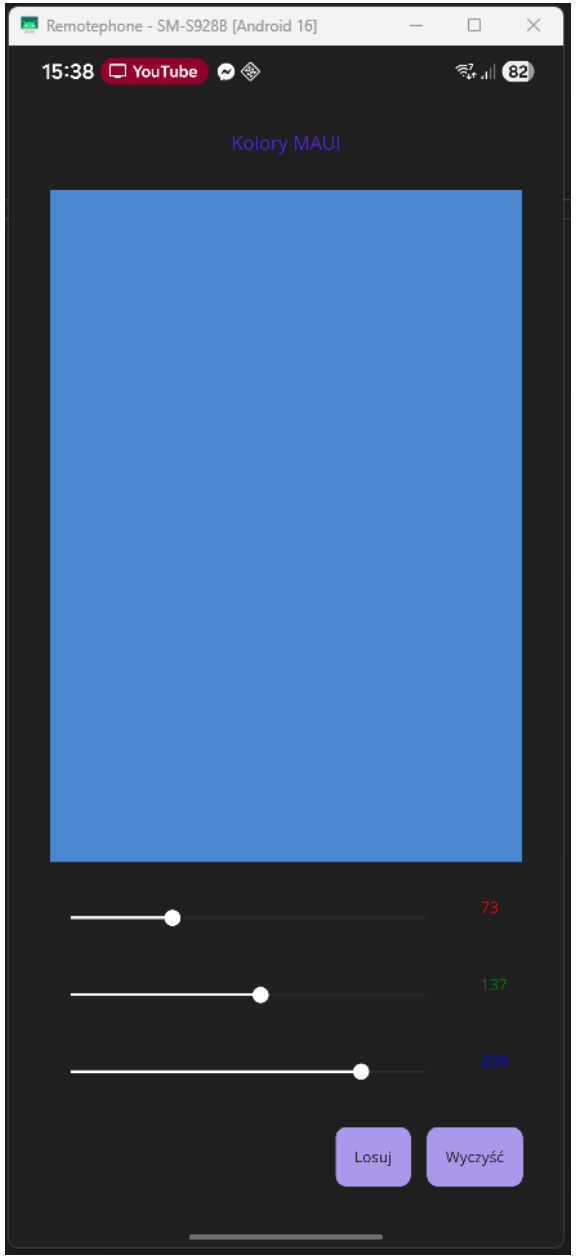
```
// Jaki tryb ma ustawiony użytkownik w systemie?
AppTheme systemTheme = Application.Current.RequestedTheme;

// Jaki tryb wymusza nasza aplikacja?
AppTheme appTheme = Application.Current.UserAppTheme;
```

Odczytywanie trybu aplikacji

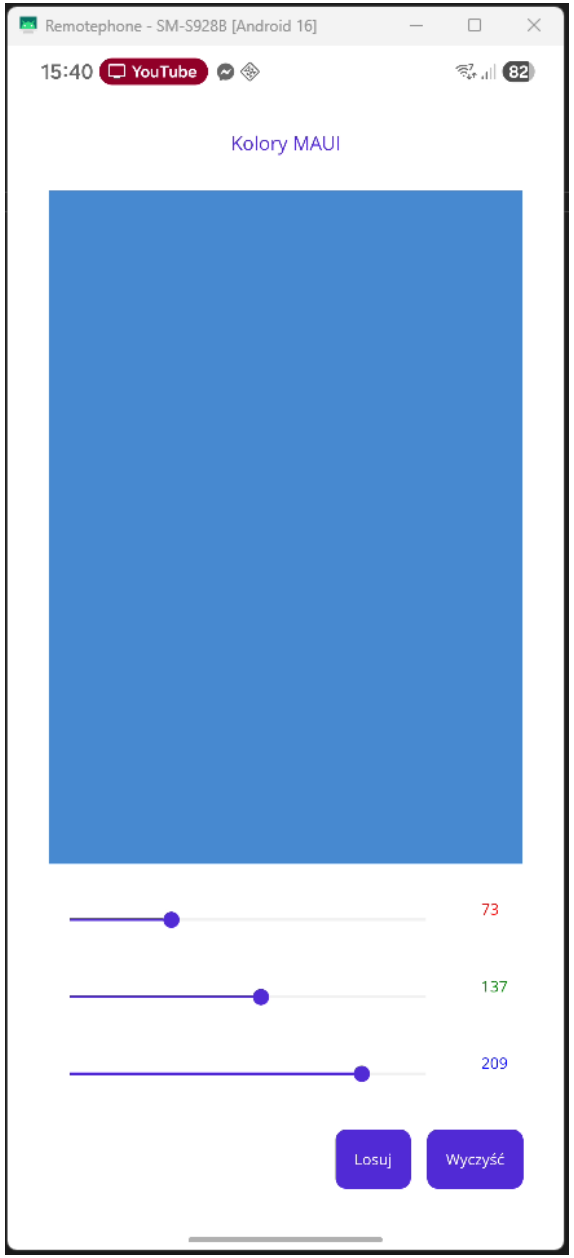
Tryb aplikacji

Android





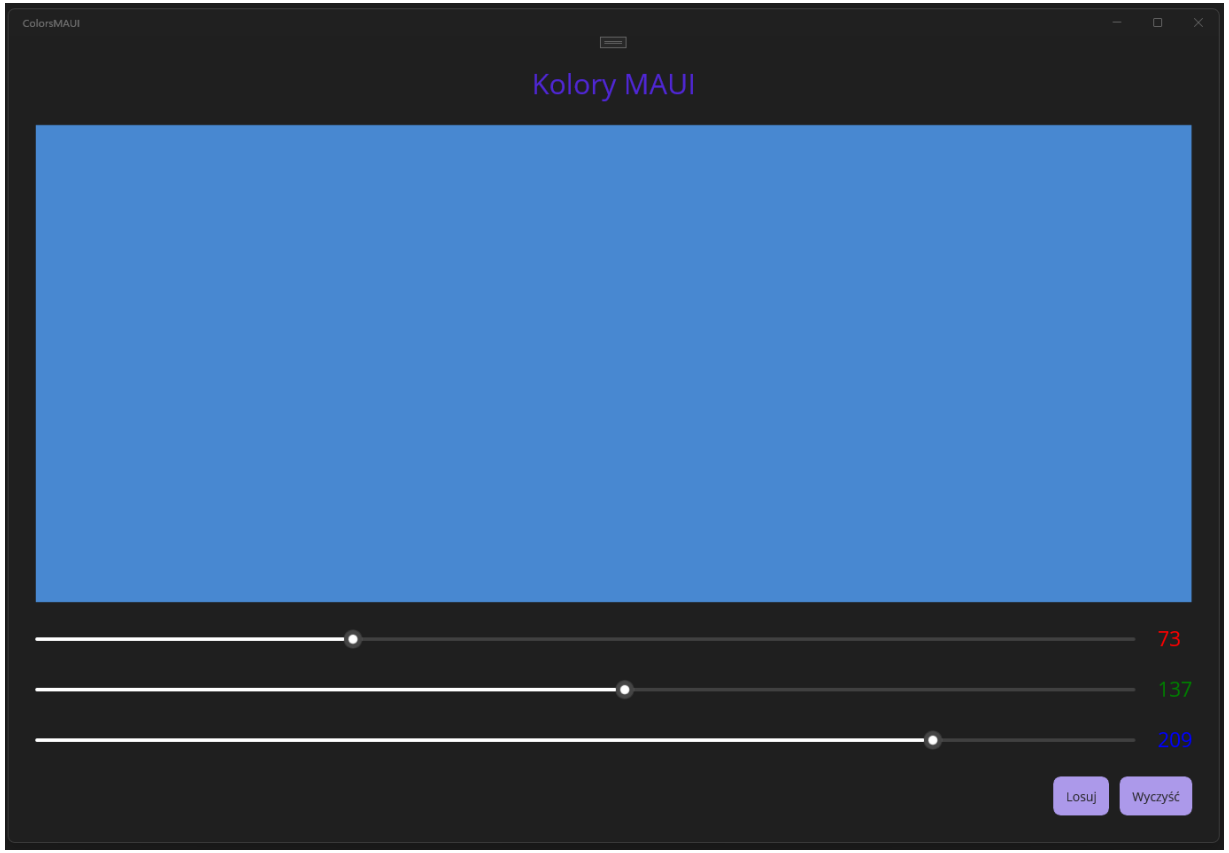
Ciemny



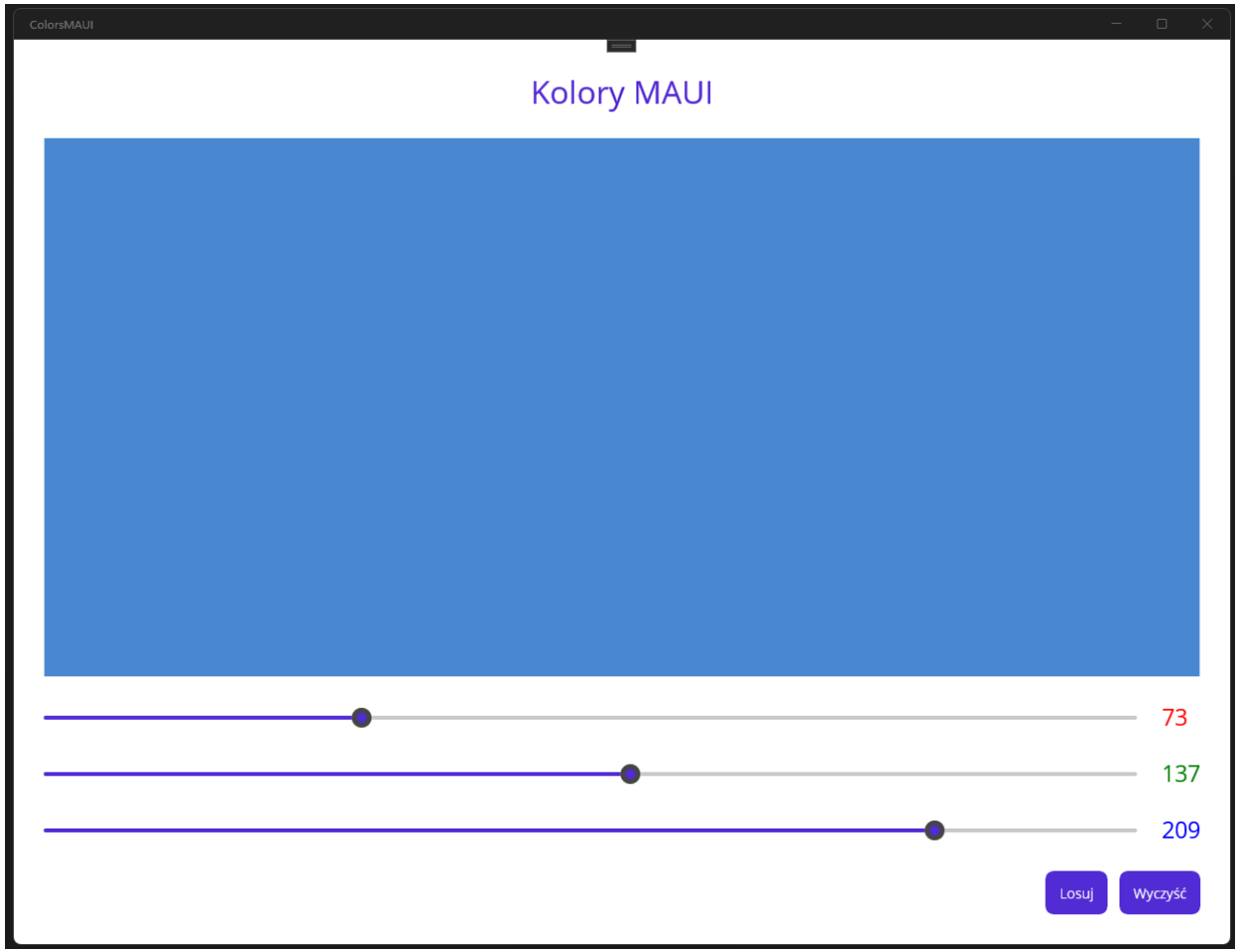
Jasny

Tryb aplikacji

Windows



Ciemny



Jasny

AppThemeBinding

- `AppThemeBinding` pozwala ustawić różne wartości właściwości w zależności od aktywnego trybu – bezpośrednio w XAML.

```xaml
<!-- Kolor tła zmienia się automatycznie z trybem -->
<ContentPage BackgroundColor="{AppThemeBinding Light=White, Dark=#1E1E1E}">

    <VerticalStackLayout Padding="20" Spacing="15">

        <!-- Tekst dopasowuje kolor do trybu -->
        <Label Text="Witaj w aplikacji!"
               FontSize="24"
               TextColor="{AppThemeBinding Light=Black, Dark=White}" />

        <!-- Ramka z różnym kolorem obramowania -->
        <Border StrokeShape="RoundRectangle 10"
                Stroke="{AppThemeBinding Light=#E0E0E0, Dark=#444444}"
                BackgroundColor="{AppThemeBinding Light=#F5F5F5, Dark=#2D2D2D}"
                Padding="15">

            <Label Text="To jest karta produktu"
                   TextColor="{AppThemeBinding Light=#333333, Dark=#CCCCCC}" />

        </Border>

    </VerticalStackLayout>
</ContentPage>
```

AppThemeBinding

- `AppThemeBinding` działa z każdą właściwością kontrolki – kolory, czcionki, obrazki, marginesy, a nawet `Source` dla `Image` (*można podać inną ikonę dla jasnego i ciemnego trybu*).

```
<Image Source="{AppThemeBinding Light=logo_dark.png, Dark=logo_light.png}"
       HeightRequest="50" />
```

Style zależne od trybu

- Zamiast powtarzać `AppThemeBinding` na każdej kontrolce, można zdefiniować kolory i style w `App.xaml` jako zasoby aplikacji.

```xaml
<ContentPage BackgroundColor="{StaticResource PageBackground}">
    <VerticalStackLayout Padding="20" Spacing="15">
        <Label Text="Produkty" Style="{StaticResource HeaderStyle}" />

        <Border Style="{StaticResource ProductCardStyle}">
            <Label Text="Laptop" TextColor="{StaticResource TextPrimary}" />
        </Border>
    </VerticalStackLayout>
</ContentPage>
```

Użycie na stronach – czysto, bez powtarzania `{AppThemeBinding}`

```xaml
<!-- App.xaml -->
<Application.Resources>
    <ResourceDictionary>

        <!-- Kolory reagujące na tryb -->
        <Color x:Key="PageBackground">
            <Color.Setter>
                <AppThemeBinding Light="White" Dark="#1E1E1E" />
            </Color.Setter>
        </Color>

        <!-- Prostsze podejście – osobne kolory + Style -->
        <Color x:Key="TextPrimary">{AppThemeBinding Light=#333333, Dark=#EEEEEE}</Color>
        <Color x:Key="CardBackground">{AppThemeBinding Light=#F5F5F5, Dark=#2D2D2D}</Color>
        <Color x:Key="CardBorder">{AppThemeBinding Light=#E0E0E0, Dark=#444444}</Color>

        <!-- Styl karty produktu -->
        <Style x:Key="ProductCardStyle" TargetType="Border">
            <Setter Property="StrokeShape" Value="RoundRectangle 10" />
            <Setter Property="Padding" Value="15" />
            <Setter Property="Stroke" Value="{StaticResource CardBorder}" />
            <Setter Property="BackgroundColor" Value="{StaticResource CardBackground}" />
        </Style>

        <!-- Styl nagłówka -->
        <Style x:Key="HeaderStyle" TargetType="Label">
            <Setter Property="FontSize" Value="24" />
            <Setter Property="FontAttributes" Value="Bold" />
            <Setter Property="TextColor" Value="{StaticResource TextPrimary}" />
        </Style>

    </ResourceDictionary>
</Application.Resources>
```

Zdefiniowanie zasobów zależnych od trybu aplikacji

Reagowanie na zmianę trybu

- Użytkownik może zmienić tryb systemowy w trakcie korzystania z aplikacji (*np. włączy się tryb nocny o zaplanowanej godzinie*). Aplikacja może na to reagować w kodzie C# przez zdarzenie `RequestedThemeChanged`.

```
// App.xaml.cs
public partial class App : Application
{
    public App()
    {
        InitializeComponent();
        RequestedThemeChanged += OnThemeChanged;
    }

    private void OnThemeChanged(object? sender, AppThemeChangedEventArgs e)
    {
        // e.RequestedTheme – nowy tryb (Light / Dark)
        Debug.WriteLine($"Tryb zmieniony na: {e.RequestedTheme}");
    }
}
```

Przydaje się to w scenariuszach, które **`AppThemeBinding`** nie pokrywa – np. zmiana kolorów na wykresie, przeładowanie mapy, aktualizacja ikon generowanych w kodzie.

Przełącznik trybu w aplikacji

- Istnieje możliwość zaimplementowania w aplikacji funkcji, umożliwiającej przełączanie się między trybami.

```csharp
public partial class SettingsViewModel : ObservableObject
{
    [ObservableProperty]
    private bool _isDarkMode;

    partial void OnIsDarkModeChanged(bool value)
    {
        Application.Current!.UserAppTheme = value
            ? AppTheme.Dark
            : AppTheme.Light;
    }
}
```

Zmiana trybu (C#)

```xaml
<HorizontalStackLayout Spacing="10">
    <Label Text="Tryb ciemny" VerticalOptions="Center" />
    <Switch IsToggled="{Binding IsDarkMode}" />
</HorizontalStackLayout>
```

Zmiana trybu (XAML)