Listy danych w .NET MAUI

Bartosz Miąskowski

CollectionView

`CollectionView` to komponent w .NET MAUI służący do prezentowania danych w postaci listy lub siatki.

- Przykładowe użycia:
  - lista produktów w sklepie,
  - lista wiadomości w czacie,
  - galeria zdjęć,
  - lista zadań,
  - lista użytkowników.
- Możliwości CollectionView:
  - pobieranie danych z kolekcji (*np. z* `ObservableCollection`),
  - wyświetlanie elementów kolekcji,
  - określanie wyglądu elementu – `ItemTemplate`,
  - obsługa zaznaczania elementów listy,
  - obsługuje wirtualizację elementów,
  - obsługa przewijania elementów.

`CollectionView` – widok siatki

CollectionView

```xaml
<CollectionView ItemsSource="{Binding Products}" />
```

Wiązanie kontrolki z kolekcją w XAML

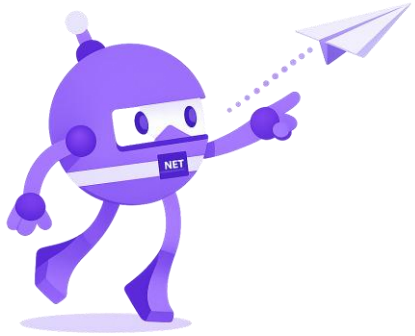
```csharp
using System.Collections.ObjectModel;

public partial class ProductsViewModel : ObservableObject
{
    public ObservableCollection<string> Products { get; }
        = new ObservableCollection<string>
    {
        "Jabłko",
        "Banan",
        "Pomarańcza"
    };
}
```

Przykładowa kolekcja w VM

Używamy `ObservableCollection<T>` aby kontrolka była **aktualizowana automatycznie**.

Wygląd pojedynczego elementu

- Domyślnie `CollectionView` wyświetla listę zwykłych elementów tekstowych – często jest to stanowczo za mało. Jeżeli chcemy zdefiniować jak ma wyglądać pojedynczy element listy, używamy `ItemTemplate`, który pozwala zbudować szablon dla pojedynczego elementu w `CollectionView`.
- `DataTemplate`:
  - opisuje wygląd jednego elementu,
  - każdy element kolekcji otrzymuje swoją kopię szablonu,
  - wiązanie (*binding*) wewnątrz szablonu odnosi się do danego elementu kolekcji.



```
<CollectionView ItemsSource="{Binding Products}">
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <Label Text="{Binding}"
                   FontSize="18"
                   TextColor="DarkBlue"/>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

Przykładowy szablon w `CollectionView`

DataTemplate z obiektem

```xaml
<CollectionView ItemsSource="{Binding Products}">
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <Grid Padding="10" ColumnDefinitions="80,*,Auto">
                <Image Source="{Binding ImageUrl}"
                       WidthRequest="70"
                       HeightRequest="70" />
                <VerticalStackLayout Grid.Column="1">
                    <Label Text="{Binding Name}"
                           FontAttributes="Bold"
                           FontSize="18"/>
                    <Label Text="{Binding Price, StringFormat='Cena: {0} zł'}"
                           TextColor="Gray"/>
                </VerticalStackLayout>
            </Grid>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

Kompleksowy szablon w XAML do wyświetlania listy produktów

- Co tutaj się dzieje?
  - `{Binding Name}` – odnosi się do właściwości obiektu Product.
  - `{Binding Price}` – pobiera cenę produktu.
  - Każdy element listy ma własny `Grid`.
  - UI ładuje się dynamicznie na podstawie danych.

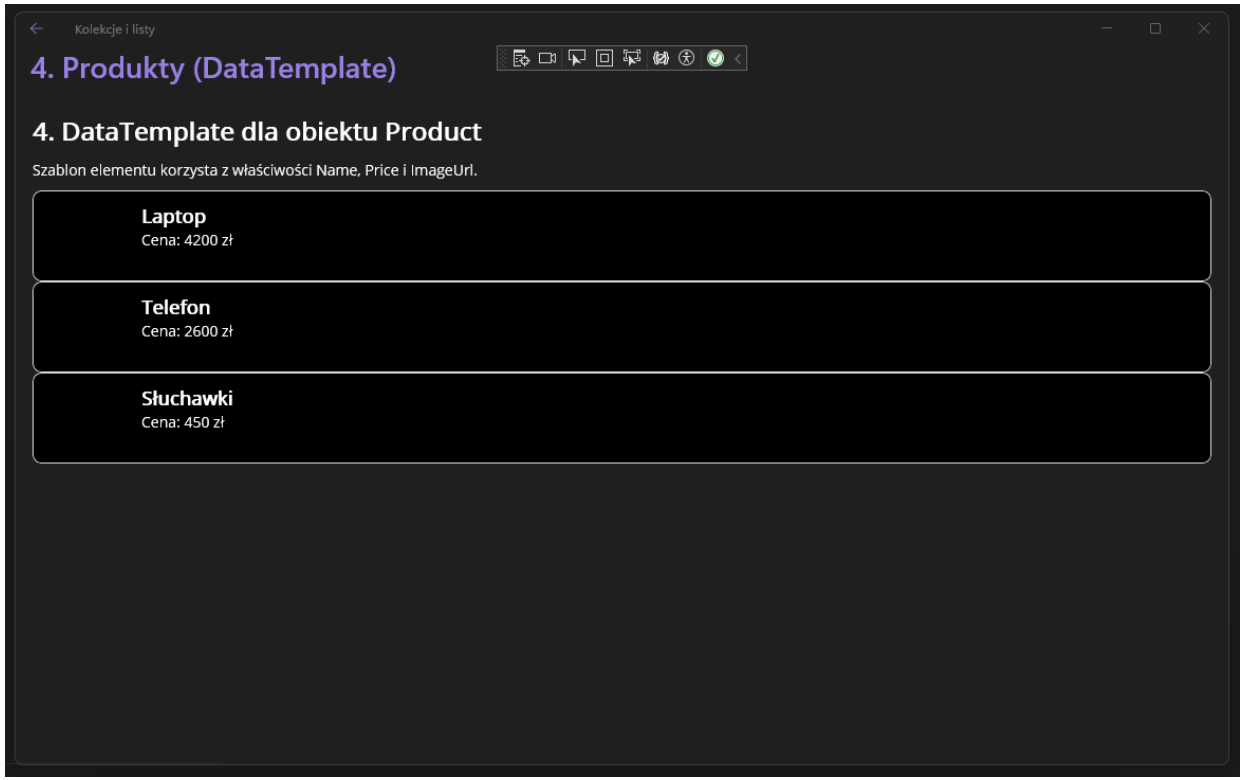

```csharp
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string ImageUrl { get; set; }
}
```

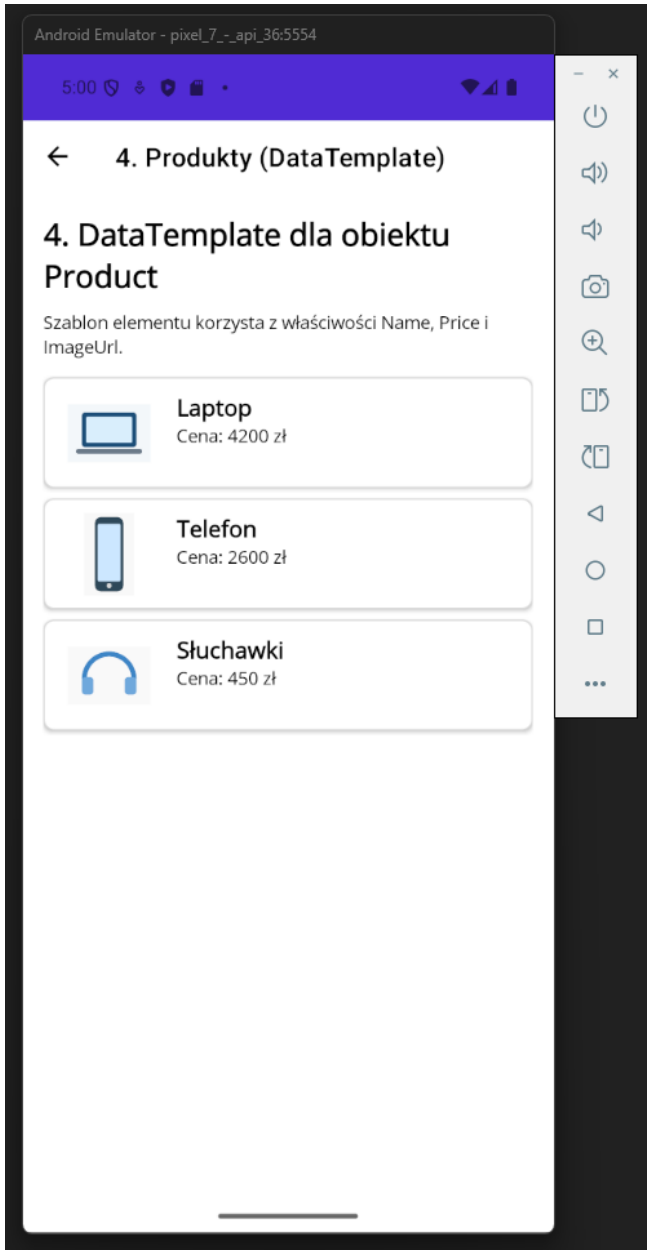
Klasa produktu

```csharp
public partial class ProductsViewModel : ObservableObject
{
    public ObservableCollection<Product> Products { get; }
        = new()
    {
        new Product
        {
            Name = "Laptop",
            Price = 4999,
            ImageUrl = "laptop.png"
        },
        new Product
        {
            Name = "Telefon",
            Price = 2999,
            ImageUrl = "phone.png"
        }
    };
}
```

Źródło danych `CollectionView`

DataTemplate z obiektem





DataTemplate a BindingContext

Skąd DataTemplate wie, że {Binding Name} dotyczy instancji klasy Product?

- CollectionView dla każdego elementu w kolekcji:
  - Tworzy kopię DataTemplate.
  - Ustawia BindingContext na pojedynczy element kolekcji.

- **Działa to tak, że**:
  - Pierwszy element jest powiązany przez BindingContext z pierwszym elementem kolekcji.
  - Drugi element jest powiązany przez BindingContext z drugim elementem kolekcji.

```
<Label Text="{Binding Name}" />
```

Name oznacza Product.Name, **kontekstem jest klasa Product**.

Układanie elementów w `CollectionView`

`LinearItemsLayout` – lista pionowa (*domyślna*)

```
<CollectionView ItemsSource="{Binding Products}">
    <CollectionView.ItemsLayout>
        <LinearItemsLayout Orientation="Vertical" />
    </CollectionView.ItemsLayout>
</CollectionView>
```

Układanie elementów w `CollectionView`

`LinearItemsLayout` – lista pozioma

```
<CollectionView ItemsSource="{Binding Products}">
    <CollectionView.ItemsLayout>
        <LinearItemsLayout Orientation="Horizontal" />
    </CollectionView.ItemsLayout>
</CollectionView>
```

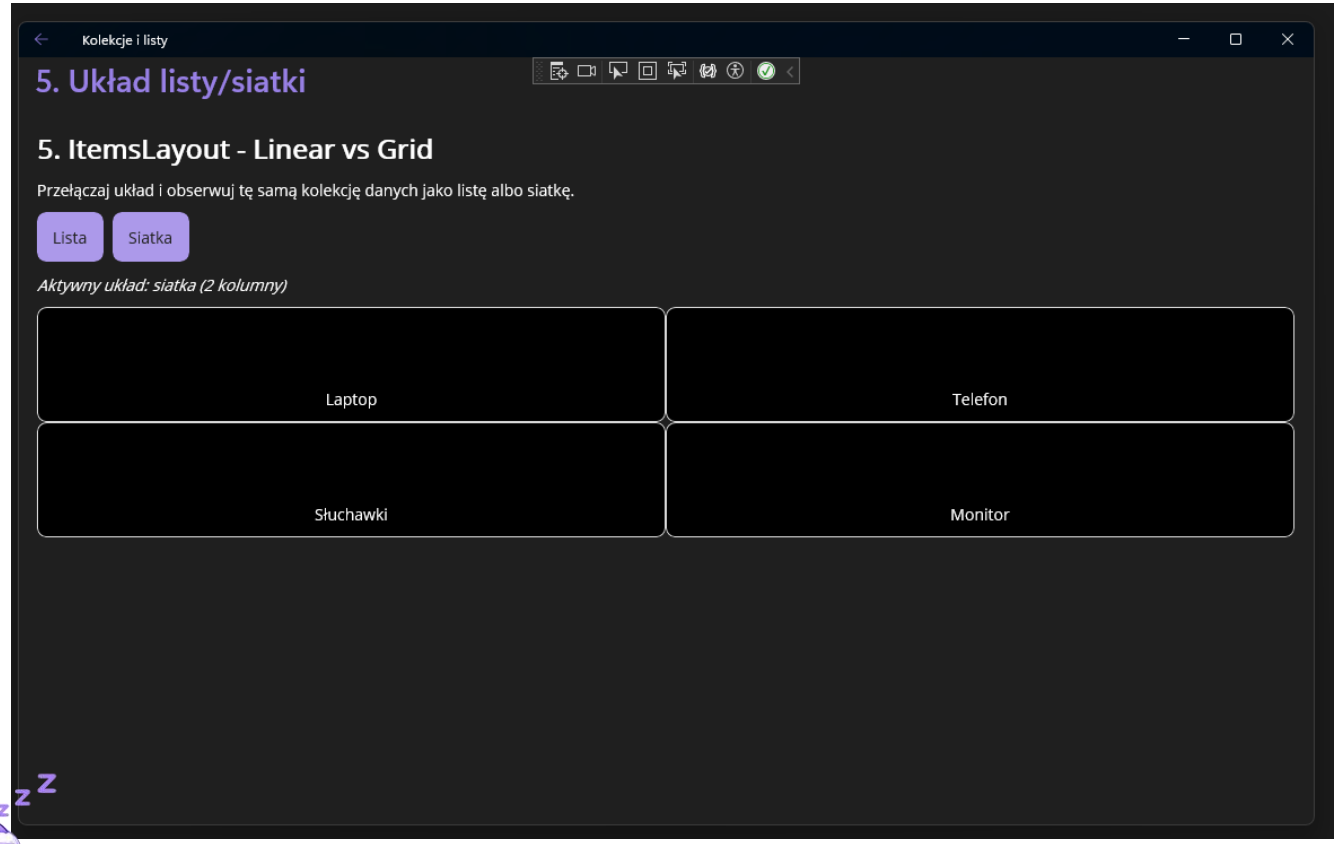
Układanie elementów w `CollectionView`

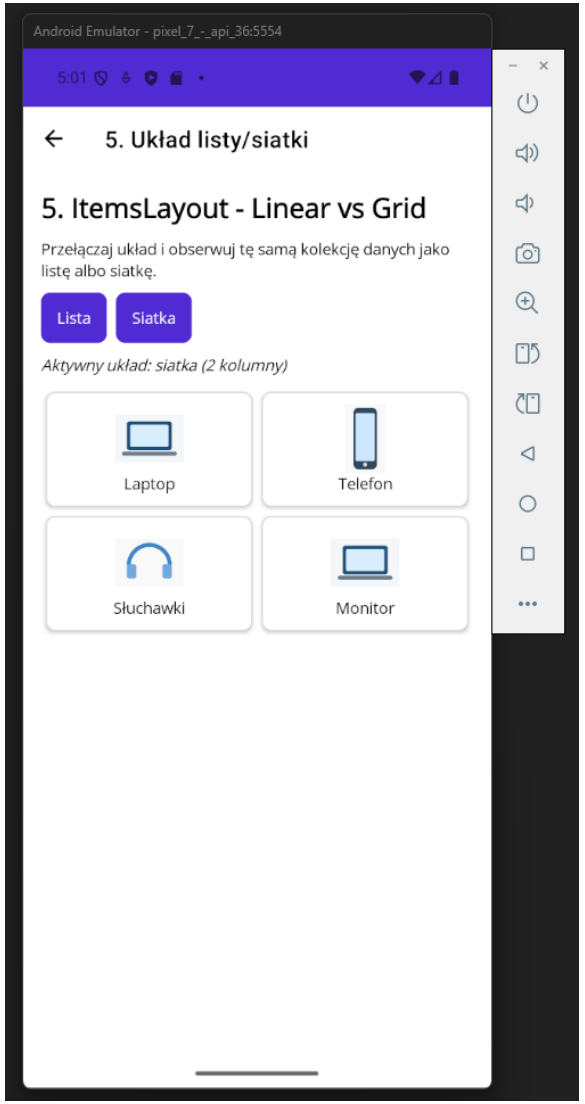
`GridItemsLayout` – siatka

```
<CollectionView ItemsSource="{Binding Products}">
    <CollectionView.ItemsLayout>
        <GridItemsLayout Orientation="Vertical"
                         Span="2" />
    </CollectionView.ItemsLayout>
</CollectionView>
```

Widok siatki w `CollectionView` również może być pionowy lub poziomy.

Układanie elementów w `CollectionView`





DataTemplateSelector – co to?

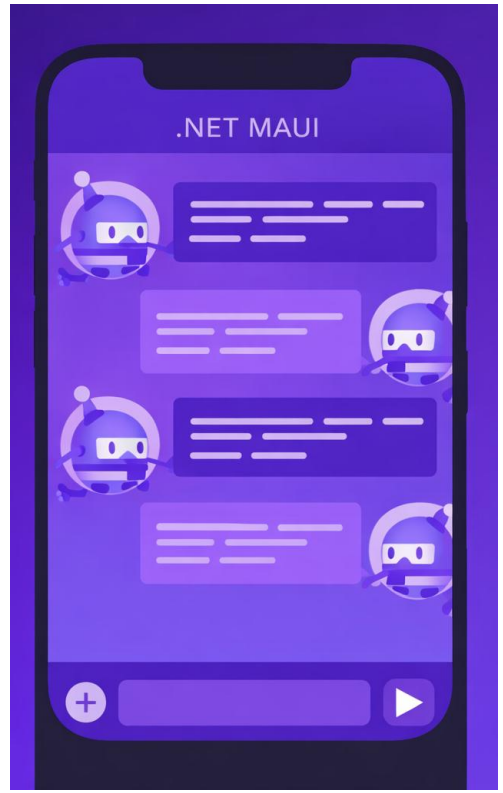
- Czasami chcemy, aby elementy w `CollectionView` wyglądały inaczej, w zależności od swoich właściwości. Załóżmy, że tworzymy listę wiadomości – wiadomości odebrane i wysłane. Chcemy:
  - aby wiadomości miały inne kolory,
  - inne wyrównanie,
  - inny wygląd.

Ale są one umieszczone w jednej kolekcji.

```
public class Message
{
    public string Text { get; set; }
    public bool IsSentByMe { get; set; }
}
```

Chcemy, aby `DataTemplate` był uzależniony od `IsSentByMe`.

```
<DataTemplate x:Key="SentTemplate">
    <Frame BackgroundColor="LightBlue"
           HorizontalOptions="End">
        <Label Text="{Binding Text}" />
    </Frame>
</DataTemplate>

<DataTemplate x:Key="ReceivedTemplate">
    <Frame BackgroundColor="LightGray"
           HorizontalOptions="Start">
        <Label Text="{Binding Text}" />
    </Frame>
</DataTemplate>
```

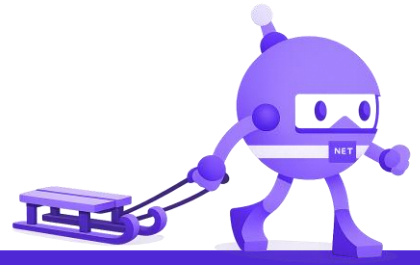


Tworzenie `DataTemplateSelector` w C#

- Co tu się dzieje?
  1. Obiekt `item` to pojedynczy element z kolekcji.
  2. Rzutujemy go na `Message`.
  3. Sprawdzamy właściwość `IsSentByMe`.
  4. Zwracamy odpowiedni `DataTemplate`

- Logika wyboru szablonu jest w C#, dzięki temu:
  - możemy używać warunków,
  - możemy sprawdzać typy,
  - możemy mieć dowolnie złożoną logikę.

```
using Microsoft.Maui.Controls;

public class MessageTemplateSelector : DataTemplateSelector
{
    public DataTemplate SentTemplate { get; set; }
    public DataTemplate ReceivedTemplate { get; set; }

    protected override DataTemplate OnSelectTemplate(
        object item,
        BindableObject container)
    {
        var message = (Message)item;

        if (message.IsSentByMe)
            return SentTemplate;

        return ReceivedTemplate;
    }
}
```



`CollectionView` **dla każdego elementu** odpytuje selektor, **który szablon powinien zostać użyty**.

Podpięcie `DataTemplateSelector` w XAML

```xaml
<ContentPage.Resources>
    <DataTemplate x:Key="SentTemplate">
        <Frame BackgroundColor="LightBlue"
               Padding="10"
               CornerRadius="12"
               HorizontalOptions="End">
            <Label Text="{Binding Text}" />
        </Frame>
    </DataTemplate>

    <DataTemplate x:Key="ReceivedTemplate">
        <Frame BackgroundColor="LightGray"
               Padding="10"
               CornerRadius="12"
               HorizontalOptions="Start">
            <Label Text="{Binding Text}" />
        </Frame>
    </DataTemplate>
</ContentPage.Resources>
```

1) Definiujemy szablon w `Resources`

```xaml
<local:MessageTemplateSelector x:Key="MessageSelector"
    SentTemplate="{StaticResource SentTemplate}"
    ReceivedTemplate="{StaticResource ReceivedTemplate}" />
```

2) Tworzymy selektor w `Resources`

```xaml
<CollectionView ItemsSource="{Binding Messages}"
                ItemTemplate="{StaticResource MessageSelector}" />
```

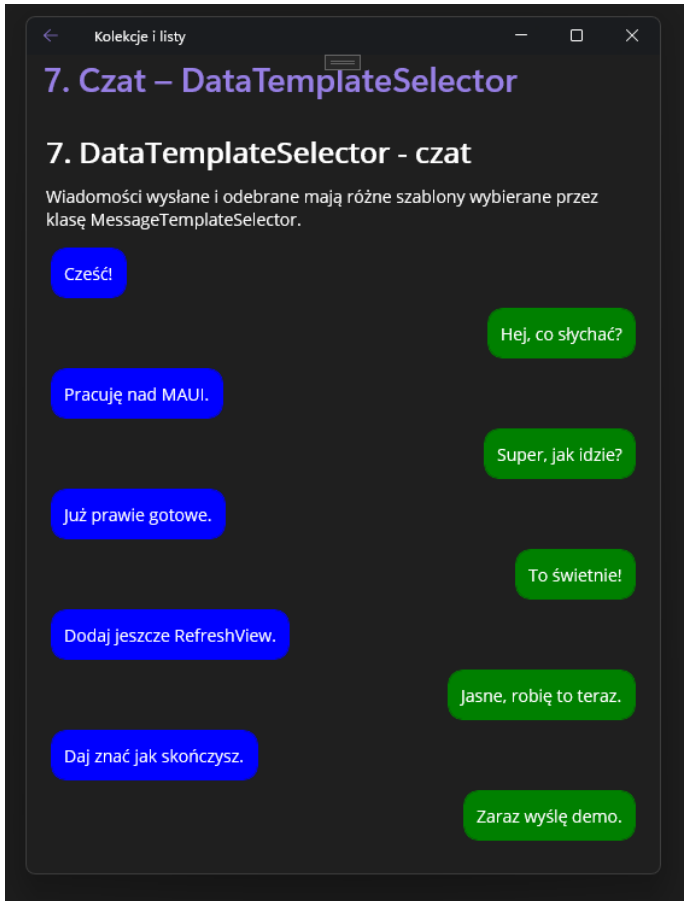
3) Używamy selektora w `CollectionView`

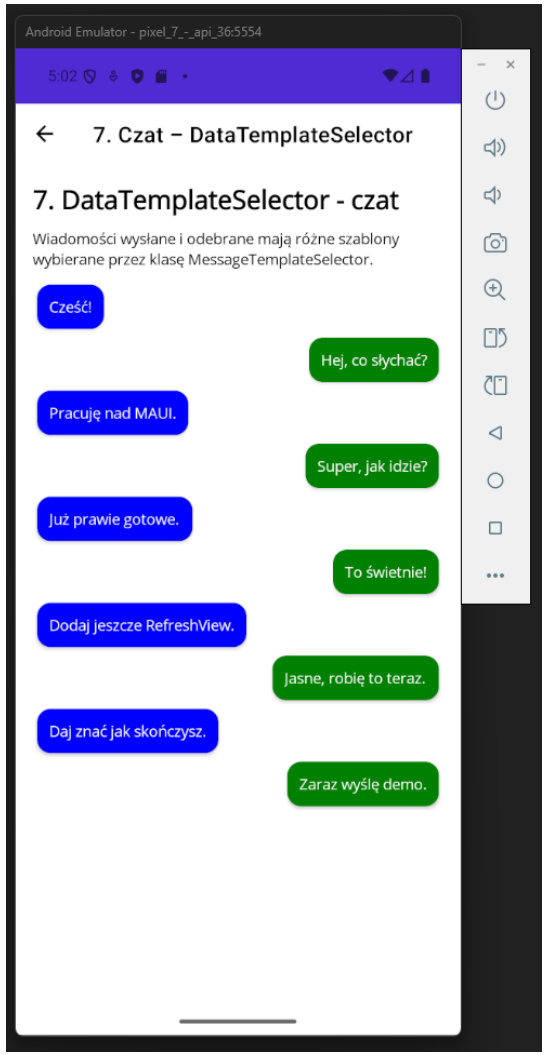
Ważne!
Elementy `DataTemplate` możemy zapisać globalnie dla całej strony (*jak na przykładzie*), lub lokalnie dla `CollectionView` – wtedy umieszczamy je w `<CollectionView.Resources>`.

Co tutaj się dzieje?

1. Kontrolka `CollectionView` pobiera wiadomość.
2. Wywołuje `OnSelectTemplate`.
3. Dodaje odpowiedni `DataTemplate`.
4. Renderuje element.

Podpięcie `DataTemplateSelector` w XAML





Grupowanie elementów w `CollectionView`

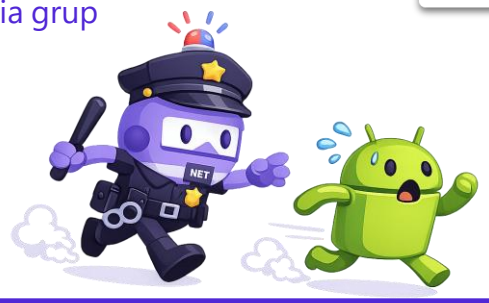
`CollectionView` pozwala nam na grupowanie wyświetlanych elementów.

```
public class ProductGroup : List<Product>
{
    public string CategoryName { get; set; }

    public ProductGroup(string categoryName, List<Product> products)
        : base(products)
    {
        CategoryName = categoryName;
    }
}
```

Struktura danych – potrzebujemy struktury, do przechowywania grup

```
public ObservableCollection<ProductGroup> Groups { get; } = new()
{
    new ProductGroup("Elektronika", new List<Product>
    {
        new Product { Name = "Laptop", Price = 5000 },
        new Product { Name = "Telefon", Price = 3000 }
    }),

    new ProductGroup("Spożywcze", new List<Product>
    {
        new Product { Name = "Mleko", Price = 4 },
        new Product { Name = "Chleb", Price = 5 }
    })
};
```

Obsługa w ViewModelu



W tym przykładzie użyliśmy `List<T>` **dla uproszczenia**, zakładamy, że elementy nie będą się aktualizowały po grupowaniu.

Grupowanie elementów w `CollectionView`

```
<CollectionView ItemsSource="{Binding Groups}"
                IsGrouped="True">
```

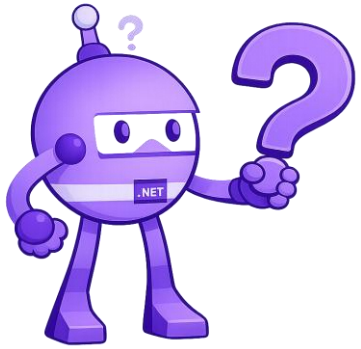
1) Włączamy `IsGrouped` w `CollectionView`

```
<CollectionView.GroupHeaderTemplate>
    <DataTemplate>
        <Label Text="{Binding CategoryName}"
               FontAttributes="Bold"
               FontSize="20"
               BackgroundColor="#EEE"
               Padding="5" />
    </DataTemplate>
</CollectionView.GroupHeaderTemplate>
```

2) Definiujemy nagłówek grupy

```
<CollectionView.ItemTemplate>
    <DataTemplate>
        <Label Text="{Binding Name}"
               Padding="10"/>
    </DataTemplate>
</CollectionView.ItemTemplate>
```

3) Definiujemy wygląd elementu

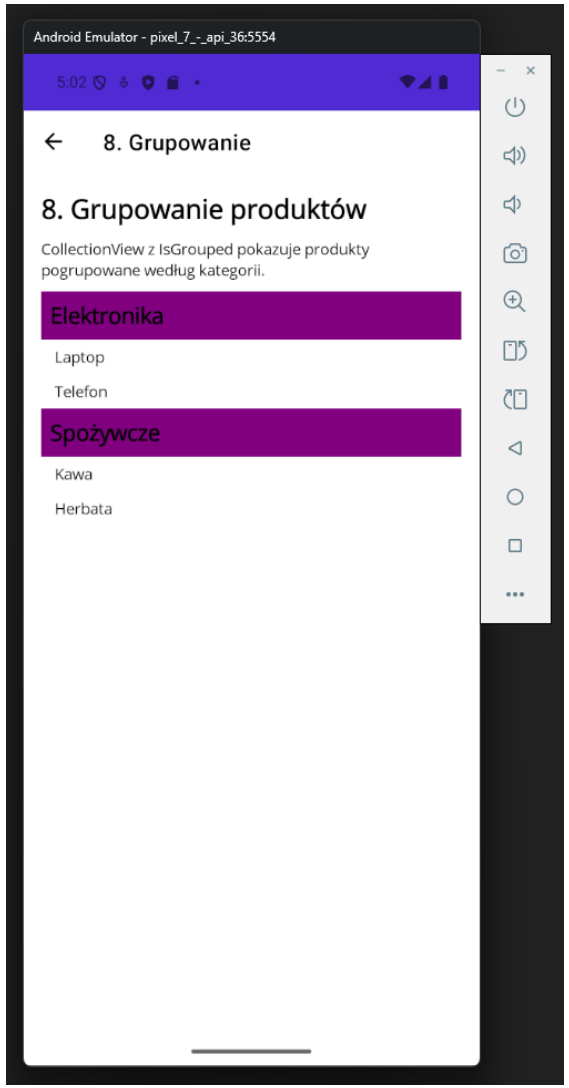


Co tutaj się dzieje?

1. CollectionView iteruje po grupach.
2. Renderuje GroupHeaderTemplate.
3. Iteruje po elementach w grupie.
4. Renderuje ItemTemplate.

Pytanie:
Jaki będzie `BindingContext` wewnątrz `GroupHeaderTemplate`?

Grupowanie elementów w `CollectionView`



Zaznaczanie elementów kolekcji

- `CollectionView` pozwala zaznaczać elementy, robimy to poprzez właściwość `SelectionMode`:
  - `None` – domyślny, brak zaznaczenia.
  - `Single` – można zaznaczyć jeden element (*nowe zaznaczenie usuwa poprzednie*).
  - `Multiple` – można zaznaczyć wiele elementów.

```
<CollectionView
    ItemsSource="{Binding Products}"
    SelectionMode="Single"
    SelectedItem="{Binding SelectedProduct}" />
```

Binding zaznaczonego elementu – tryb `Single`

```
[ObservableProperty]
private Product selectedProduct;
```

Pole w VM – tryb `Single`

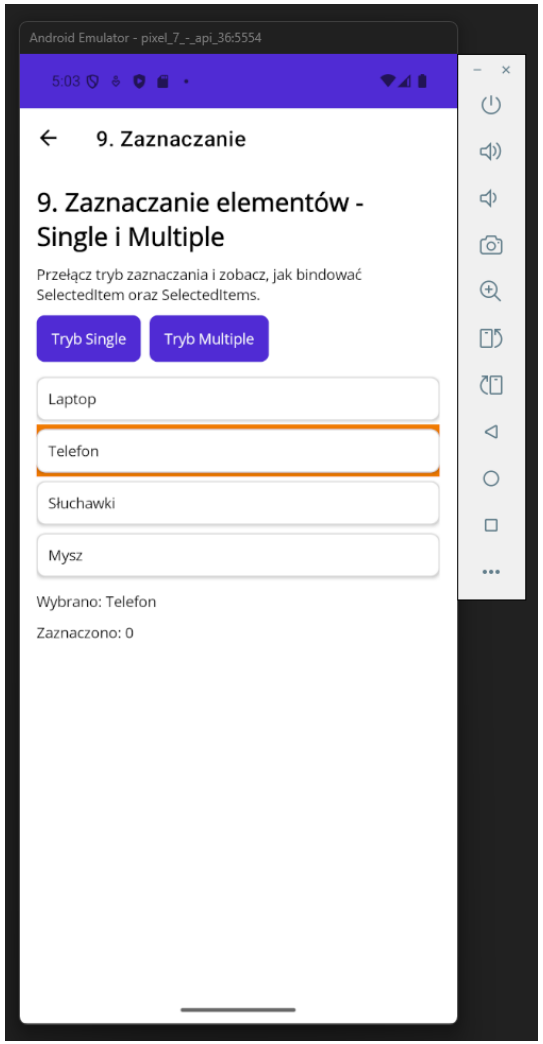
```
<CollectionView
    ItemsSource="{Binding Products}"
    SelectionMode="Multiple"
    SelectedItems="{Binding SelectedProducts}" />
```

Binding zaznaczonego elementu – tryb `Multiple`

```
public ObservableCollection<Product> SelectedProducts { get; } = new();
```

Pole w VM – tryb `Multiple`

Zaznaczanie elementów kolekcji



Reagowanie na zmianę zaznaczenia

- Jeżeli chcemy zareagować na zmianę zaznaczenia w `CollectionView` używamy `SelectionChangedCommand`. Pozwala to na wykonanie akcji, już w momencie zaznaczenia elementu.
  - W trybie `Single` parametrem jest zaznaczony element.
  - W trybie `Multiple` parametrem jest kolekcja zaznaczonych elementów.

```
<CollectionView
    ItemsSource="{Binding Products}"
    SelectionMode="Single"
    SelectedItem="{Binding SelectedProduct}"
    SelectionChangedCommand="{Binding ProductSelectedCommand}" />
```

```
[RelayCommand]
private async Task ProductSelected(Product product)
{
    if (product == null)
        return;

    await Shell.Current.DisplayAlert(
        "Wybrano produkt",
        product.Name,
        "OK");

    SelectedProduct = null; // opcjonalnie – usuwa zaznaczenie
}
```

Uwaga!
Jeżeli nie zresetujemy `SelectedProduct`, element **pozostanie podświetlony** po zareagowaniu na zdarzenie, i kliknięcie na niego **drugi raz nie wywoła tego zdarzenia**.

`CollectionView` – dodatkowe elementy

`EmptyView` mówi aplikacji, co wyświetlić, gdy lista jest pusta.

```
<CollectionView ItemsSource="{Binding Products}">
    <CollectionView.EmptyView>
        <Label Text="Brak produktów"
               HorizontalOptions="Center"
               VerticalOptions="Center" />
    </CollectionView.EmptyView>
</CollectionView>
```

Informacja pokazuje się, gdy kolekcja jest pusta lub `ItemsSource = null`

`CollectionView` – dodatkowe elementy

`Header` to element wyświetlany na początku listy.

```
<CollectionView.Header>
    <Label Text="Lista produktów"
           FontSize="24"
           Padding="10" />
</CollectionView.Header>
```

Header przewija się razem z listą, można tutaj umieścić przykładowo jakąś wyszukiwarkę.

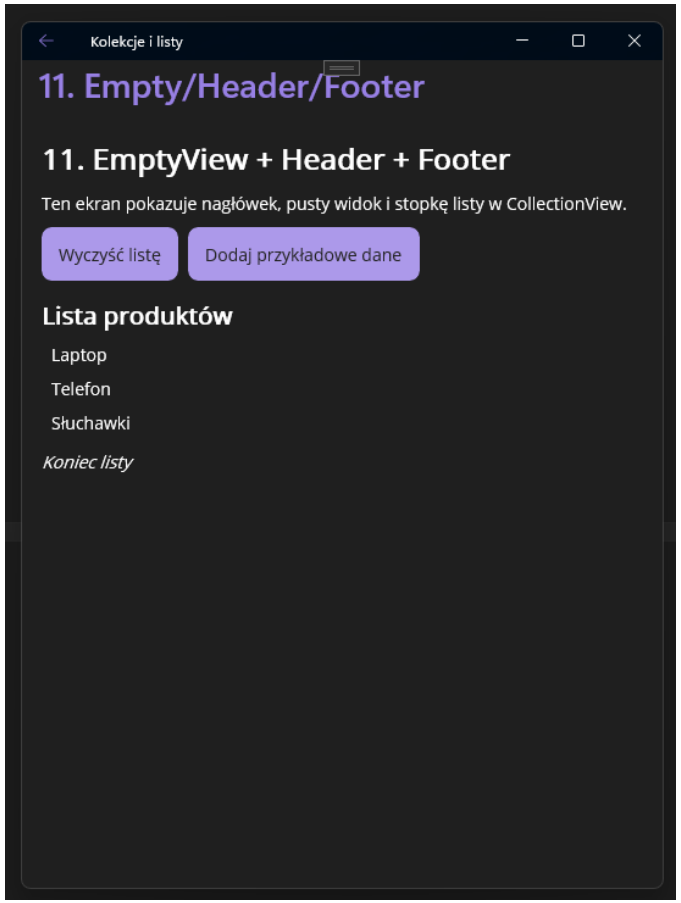
`CollectionView` – dodatkowe elementy

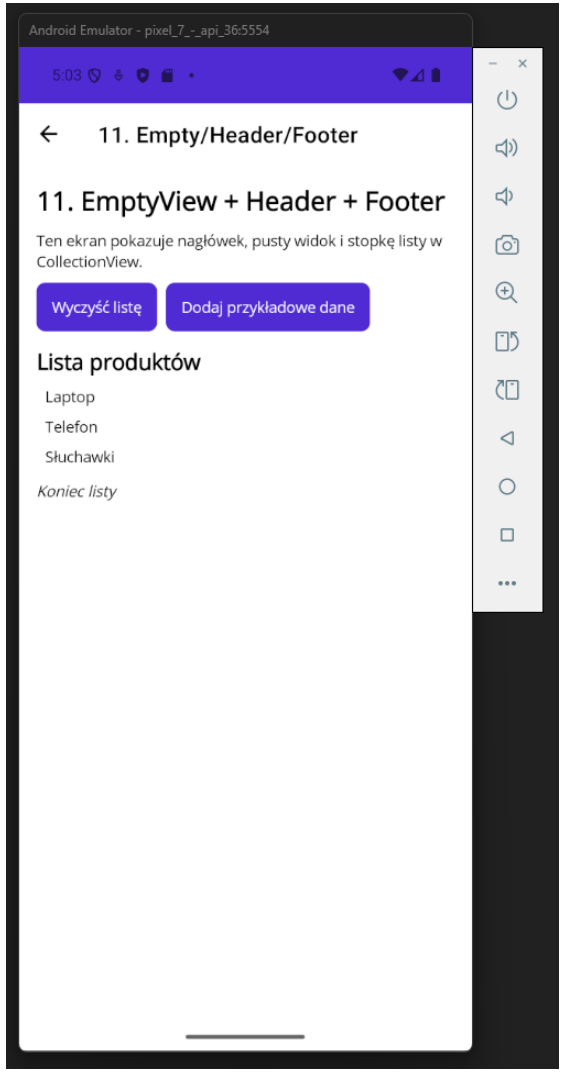
`Footer` to element wyświetlany na końcu listy.

```
<CollectionView.Footer>
    <Label Text="Koniec listy"
           HorizontalOptions="Center"
           Padding="10" />
</CollectionView.Footer>
```

`Footer` przewija się razem z listą, można tutaj umieścić np. przycisk „załaduj więcej”.

CollectionView – dodatkowe elementy





Dynamiczne ładowanie elementów

- `CollectionView` pozwala na dynamiczne ładowanie danych, sprawdza się to wtedy, gdy nasze źródło danych ma bardzo dużo elementów (*np. 10.000*) i jest pobierane przykładowo z API lub bazy danych.
  - Nie chcemy ładować wszystkiego na raz – będzie to trwało bardzo długo.
  - Nie chcemy blokować aplikacji.
  - Nie chcemy zawalić całej pamięci RAM.

W takiej sytuacji, najlepszym rozwiązaniem będzie ładowanie danych kawałek po kawałku.

Dynamiczne ładowanie elementów

`RemainingItemsThershold` określa, w którym momencie wywoływana jest komenda ładująca kolejne elementy.

Co się dzieje?

1. Użytkownik przewija listę.
2. Zbliża się do końca.
3. Wywołuje się komenda `LoadMoreCommand`.
4. Dodawane są nowe elementy.
5. Lista się rozszerza.

Dzięki temu, możemy przewijać prawie w nieskończoność.


```xml
<CollectionView
    ItemsSource="{Binding Products}"
    RemainingItemsThreshold="5"
    RemainingItemsThresholdReachedCommand="{Binding LoadMoreCommand}" />
```

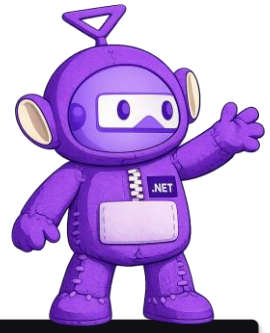
Gdy użytkownik **dojdzie do ostatnich 5 elementów** listy, wywoływana jest komenda.

```
[RelayCommand]
private void LoadMore()
{
    for (int i = 0; i < 10; i++)
    {
        Products.Add(new Product
        {
            Name = $"Nowy produkt {Products.Count + 1}",
            Price = 100
        });
    }
}
```

Na bieżąco dodajemy kolejne elementy do listy.

`SwipeView` – akcja przesunięcia na elemencie

- `SwipeView` pozwala dodać akcje do elementu listy:
  - przesunięcie w prawo lub lewo,
  - przyciski typu „Usuń”, „Edytuj”.



```xaml
<CollectionView ItemsSource="{Binding Products}">
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <SwipeView>
                <SwipeView.RightItems>
                    <SwipeItems Mode="Execute">
                        <SwipeItem Text="Usuń"
                                   Command="{Binding BindingContext.DeleteProductCommand, Source={x:Reference ProductsPage}}"
                                   CommandParameter="{Binding}" />
                    </SwipeItems>
                </SwipeView.RightItems>

                <!-- Zwykły wygląd elementu -->
                <Grid Padding="10" ColumnDefinitions="*,Auto">
                    <Label Text="{Binding Name}" FontSize="18"/>
                    <Label Grid.Column="1"
                           Text="{Binding Price, StringFormat='{0} zł'}"
                           TextColor="Gray"/>
                </Grid>
            </SwipeView>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

Przykład `CollectionView` z akcją „Usuń” na liście

```csharp
[RelayCommand]
private void DeleteProduct(Product product)
{
    if (product is null) return;
    Products.Remove(product);
}
```

Komenda usuwania w VM

Uwaga!
`x:Reference ProductsPage` to odwołanie do strony, żeby dostać się do komendy z VM, bo wewnątrz `DataTemplate` **BindingContextem jest pojedynczy Product**.

SwipeView – akcja przesunięcia na elemencie

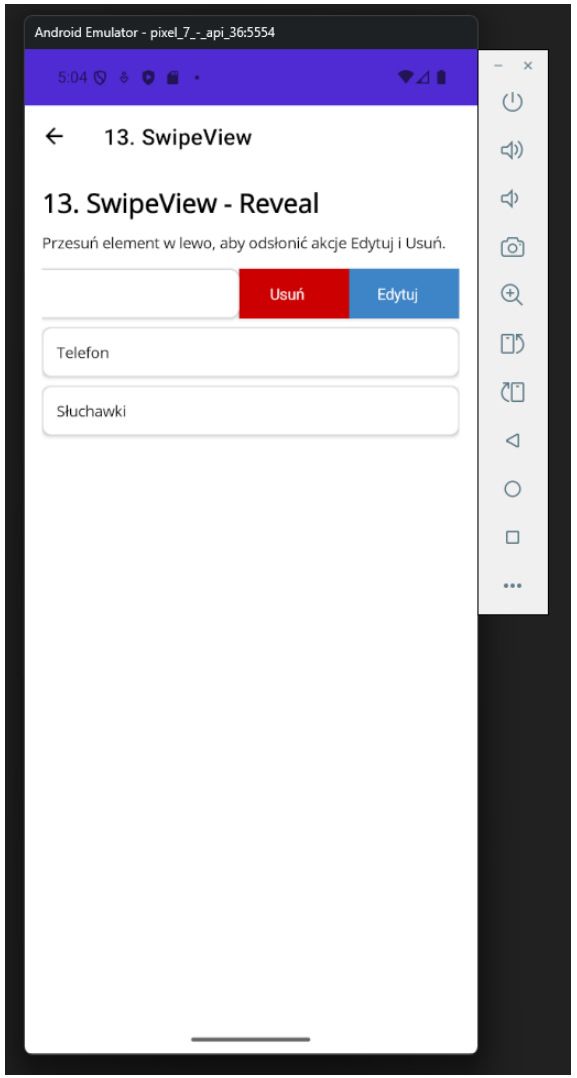
- SwipeView posiada dwa tryby:
  - **Execute** – akcja jest wykonywana natychmiast po przesunięciu, użytkownik nie musi nic więcej klikać.
  - **Reveal** – przesunięcie odsłania przyciski, które użytkownik może kliknąć.

```
[RelayCommand]
private void EditProduct(Product product)
{
    if (product is null) return;

    // np. nawigacja do strony edycji
}

[RelayCommand]
private void DeleteProduct(Product product)
{
    if (product is null) return;

    Products.Remove(product);
}
```

Uwaga!
Ta funkcja działa jedynie na **ekranach dotykowych**.

```
<CollectionView ItemsSource="{Binding Products}">

    <CollectionView.ItemTemplate>
        <DataTemplate>
            <SwipeView>
                <!-- Akcje po przesunięciu w lewo -->
                <SwipeView.RightItems>
                    <SwipeItems Mode="Reveal">

                        <SwipeItem
                            Text="Edytuj"
                            BackgroundColor="LightBlue"
                            Command="{Binding BindingContext.EditProductCommand,
                                      Source={x:Reference ProductsPage}}"
                            CommandParameter="{Binding}" />

                        <SwipeItem
                            Text="Usuń"
                            BackgroundColor="LightCoral"
                            Command="{Binding BindingContext.DeleteProductCommand,
                                      Source={x:Reference ProductsPage}}"
                            CommandParameter="{Binding}" />
                    </SwipeItems>
                </SwipeView.RightItems>
                <!-- Wygląd elementu -->
                <Grid Padding="15">
                    <Label Text="{Binding Name}"
                           FontSize="18" />
                </Grid>
            </SwipeView>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

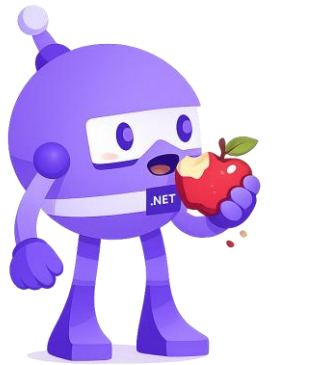
`SwipeView` – akcja przesunięcia na elemencie



SwipeItemView – własny SwipeView

- Jeżeli chcemy bardziej spersonalizować widok `SwipeView`, możemy użyć `SwipeItemView` – działa to jak layout, możliwy do zastosowania jako `SwipeItem`.

```
<SwipeItems Mode="Execute">
    <SwipeItemView
        BackgroundColor="Green"
        Command="{Binding BindingContext.MarkAsDoneCommand, Source={x:Reference ToDoPage}}"
        CommandParameter="{Binding .}"
        WidthRequest="100">
        <Label
            HorizontalOptions="Center"
            Text="Zrobione ✅"
            TextColor="{StaticResource White}"
            VerticalOptions="Center" />
    </SwipeItemView>
</SwipeItems>
```



Uwaga!
Wewnątrz **`SwipeItemView`** może znajdować się **wyłącznie jeden element**.

Odświeżanie listy – pull to refresh

- `RefreshView` pozwala na odświeżanie danych na liście gestem – przeciągnij listę w dół, by odświeżyć.
- `RefreshView` posiada dwie kluczowe rzeczy:
  - Właściwość `IsRefreshing` – informuje, czy trwa odświeżanie (*kręcący się spinner*).
  - Komenda – co ma się wykonać, po przeciągnięciu listy.

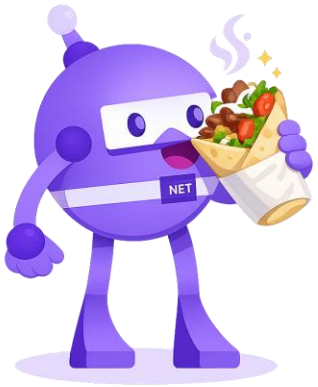
```
[ObservableProperty]
private bool isRefreshing;

[RelayCommand]
private async Task Refresh()
{
    IsRefreshing = true;

    // symulacja pobrania danych (np. z API)
    await Task.Delay(1000);

    Products.Clear();
    Products.Add(new Product { Name = "Nowe dane 1", Price = 10 });
    Products.Add(new Product { Name = "Nowe dane 2", Price = 20 });

    IsRefreshing = false;
}
```

```
<RefreshView
    IsRefreshing="{Binding IsRefreshing}"
    Command="{Binding RefreshCommand}">

    <CollectionView ItemsSource="{Binding Products}" />
</RefreshView>
```

Uwaga!
Ta funkcja działa jedynie na **ekranach dotykowych**.

CollectionView – bardzo długie listy

- Jeżeli istnieje podejrzenie, że wszystkie elementy na liście nie zmieszczą się na wyświetlaczu, będziemy potrzebowali scrollowania, które jest możliwe tylko jeżeli `CollectionView` ma określony rozmiar – w przeciwnym wypadku, może wyjść poza obszar wyświetlania. W tym celu, możemy:
  - Użyć `Grid`, jako kontenera na `CollectionView`.
  - Wymusić stały rozmiar dla `CollectionView`.




```
<Grid Padding="16" RowDefinitions="Auto, Auto, *" RowSpacing="10">
    <Label Grid.Row="0" Text="15. Bardzo długa lista" FontSize="24" FontAttributes="Bold" />
    <Label Grid.Row="1" Text="Ekran zawiera 1000 elementów, aby wygodnie testować przewijanie CollectionView." />

    <CollectionView Grid.Row="2" ItemsSource="{Binding Items}">
        <CollectionView.ItemTemplate>
            <DataTemplate x:DataType="{x:Null}">
                <Border Padding="10" Margin="0,4" >
                    <Label Text="{Binding}" />
                </Border>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</Grid>
```

Przykład z wykorzystaniem Grida

CollectionView – bardzo długie listy

- Jeżeli istnieje podejrzenie, że wszystkie elementy na liście nie zmieszczą się na wyświetlaczu, będziemy potrzebowali scrollowania, które jest możliwe tylko jeżeli `CollectionView` ma określony rozmiar – w przeciwnym wypadku, może wyjść poza obszar wyświetlania. W tym celu, możemy:
  - Użyć `Grid`, jako kontenera na `CollectionView`.
  - Wymusić stały rozmiar dla `CollectionView`.

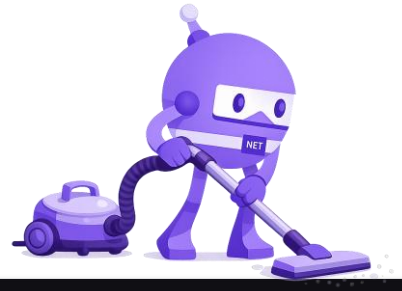
```
<Grid Padding="16" RowDefinitions="Auto, Auto, *" RowSpacing="10">
    <Label Grid.Row="0" Text="15. Bardzo długa lista" FontSize="24" F
    <Label Grid.Row="1" Text="Ekran zawiera 1000 elementów, aby wygoc

    <CollectionView Grid.Row="2" ItemsSource="{Binding Items}">
        <CollectionView.ItemTemplate>
            <DataTemplate x:DataType="{x:Null}">
                <Border Padding="10" Margin="0,4" >
                    <Label Text="{Binding}" />
                </Border>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</Grid>
```

Przykład z wykorzystaniem Grida

```
<VerticalStackLayout Padding="16" Spacing="10">
    <Label Text="15. Bardzo długa lista" FontSize="24" FontAttributes="Bol
    <Label Text="Ekran zawiera 1000 elementów, aby wygodnie testować przew

    <CollectionView HeightRequest="400" ItemsSource="{Binding Items}">
        <CollectionView.ItemTemplate>
            <DataTemplate x:DataType="{x:Null}">
                <Border Padding="10" Margin="0,4" >
                    <Label Text="{Binding}" />
                </Border>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</VerticalStackLayout>
```

Przykład z stałym rozmiarem

`ScrollView` – co to jest?



`ScrollView` umożliwia przewijanie zawartości, gdy nie mieści się ona na ekranie.

- Kiedy używamy?
  - do bardzo długich formularzy,
  - wiele kontrolek – jedna pod drugą,
  - długi tekst,
  - rozbudowane UI.

- Jak działa `ScrollView`?
  - Umożliwia przewijanie, jeżeli zawartość jest większa niż ekran.
  - Obsługuje przewijanie pionowe lub poziome.
  - Może mieć tylko jednego potomka – np. któryś z `StackLayout`.

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

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

        <Entry Placeholder="Imię" />
        <Entry Placeholder="Nazwisko" />
        <Entry Placeholder="Email" />
        <Entry Placeholder="Hasło" />
        <Entry Placeholder="Powtórz hasło" />

        <Button Text="Zarejestruj się" />

    </VerticalStackLayout>
</ScrollView>
```

Uwaga!
`ScrollView` **nie służy do wyświetlania zbiorów danych** – on przewija layout, nie kolekcję.