Helion

Flutter i Dart 2 dla początkujących

Przewodnik dla twórców aplikacji mobilnych

Packt›

Alessandro Biessek

Tytuł oryginału: Flutter for Beginners: An introductory guide to building cross-platform mobile applications with Flutter and Dart 2

Tłumaczenie: Łukasz Wójcicki

ISBN: 978-83-283-7825-4

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/flutte.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/flutte
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Część II. Interfejs użytkownika Fluttera — wszystko jest widżetem 117

ROZDZIAŁ 4

Widżety: tworzenie layoutów Fluttera

W tym rozdziale poznasz główne koncepcje widżetów, różnice między widżetami bezstanowymi i stanowymi, najpopularniejsze widżety we Flutterze oraz dowiesz się, jak dodać je do swojej aplikacji i jak tworzyć pełne interfejsy za pomocą wbudowanych lub niestandardowych widżetów opracowanych przez Ciebie.

W tym rozdziale zostaną omówione następujące tematy:

- Widżety stanowe / bezstanowe.
- Wbudowane widżety.
- Wbudowane widżety layoutów.
- Tworzenie niestandardowych widżetów.

Widżety stanowe i bezstanowe

Z rozdziału 3. dowidzieliśmy się, że widżety odgrywają ważną rolę w funkcjonowaniu aplikacji Flutter. Są to elementy, które tworzą interfejs użytkownika; są reprezentacją kodu tego, co jest widoczne dla użytkownika.

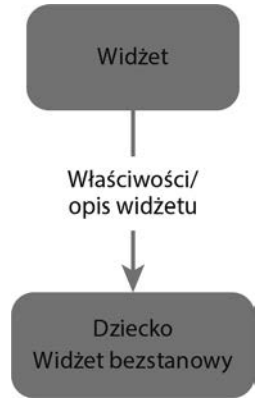
Interfejsy użytkownika prawie nigdy nie są statyczne; jak wiesz, często się zmieniają. Chociaż z definicji niezmienne, widżety nie mają być ostateczne — w końcu mamy do czynienia z UI, a UI z pewnością ulegnie zmianie w trakcie cyklu życia każdej aplikacji. Dlatego Flutter zapewnia nam dwa rodzaje widżetów: bezstanowe i stanowe.

Duża różnica między nimi polega na sposobie budowania widżetu. Do obowiązków programisty należy wybór rodzaju widżetu, który ma być używany w każdej sytuacji podczas tworzenia interfejsu użytkownika, aby maksymalnie wykorzystać możliwości warstwy renderującej widżety Fluttera.

> Flutter posiada także koncepcję **widżetów dziedziczonych** (typ `InheritedWidget`), który jest również rodzajem widżetu, ale różni się nieco od pozostałych dwóch typów, o których wspomnieliśmy. Sprawdzimy to po szczegółowym zapoznaniu się z przykładem `hello_flutter` z rozdziału 3.

Widżety bezstanowe

Typowy interfejs użytkownika będzie składał się z wielu widżetów, a niektóre z nich nigdy nie zmienią swoich właściwości po utworzeniu. Nie mają **stanu**; to znaczy, że nie zmieniają się same przez jakieś wewnętrzne działanie lub zachowanie. Zamiast tego są zmieniane przez zdarzenia zewnętrzne w widżetach nadrzędnych w drzewie widżetów. Można więc śmiało powiedzieć, że widżety bezstanowe zapewniają kontrolę nad tym, jak są powiązane z jakimś widżetem nadrzędnym w drzewie. Poniżej przedstawiono reprezentację widżetu bezstanowego:

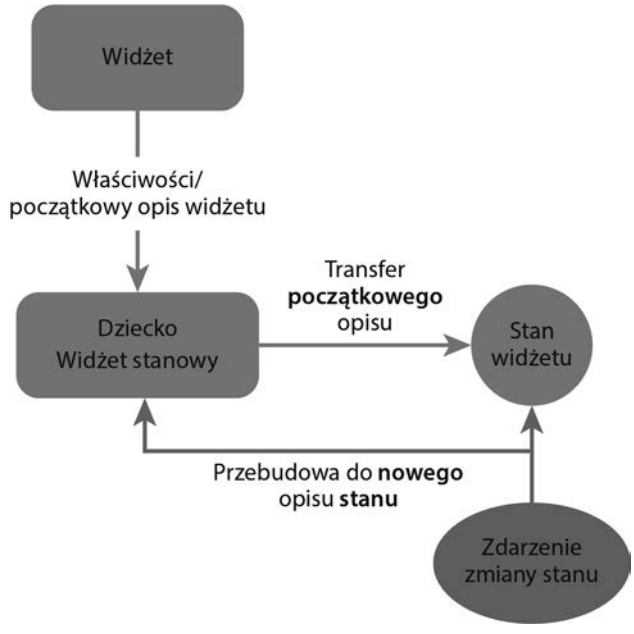


Tak więc widżet podrzędny otrzyma swój opis od widżetu nadrzędnego i sam go nie zmieni. Jeśli chodzi o kod, oznacza to, że widżety bezstanowe mają tylko właściwości `final` zdefiniowane podczas konstrukcji, i to jedyna rzecz, którą należy zbudować na ekranie urządzenia.

> Kod źródłowy szczegółowo zbadamy za chwilę, kiedy prześledzimy projekt domyślnie wygenerowany za pomocą narzędzia Fluttera, używany w poprzednim rozdziale.

Widżety stanowe

W przeciwieństwie do widżetów bezstanowych, które otrzymują opis od rodziców, utrzymujący się przez cały okres ich istnienia, widżety stanowe mają dynamicznie zmieniać opisy w trakcie swojego życia. Z definicji widżety stanowe są również niezmienne, ale mają firmową klasę `State`, która reprezentuje ich bieżący stan. Przedstawia to poniższy schemat:



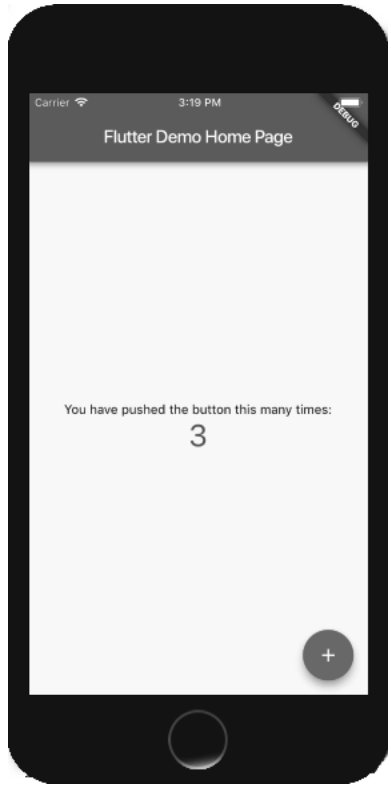
Trzymając stan widżetu w osobnym obiekcie `State`, framework może go w razie potrzeby odbudować bez utraty bieżącego skojarzonego stanu. Element w drzewie elementów zawiera odniesienie do odpowiedniego widżetu, a także skojarzony z nim obiekt `State`, który powiadomi o konieczności przebudowania widżetu, a następnie spowoduje również aktualizację w drzewie elementów.

Reprezentowanie widżetów stanowych i bezstanowych za pomocą kodu

W poprzednim rozdziale wygenerowaliśmy projekt Fluttera za pomocą następującego polecenia:

```
flutter create
```

Ten projekt został utworzony z domyślnymi argumentami z domyślnego szablonu Fluttera i reprezentuje małą aplikację z licznikiem, który pokazuje, ile razy został naciśnięty przycisk plus (+):



Aplikacja demonstracyjna Flutter z poprzedniego zrzutu ekranu jest przydatna do pokazania obu typów widżetów w praktyce.

Reprezentacja widżetu bezstanowego za pomocą kodu

Zacznijmy od zapoznania się z reprezentacją widżetu bezstanowego w postaci kodu. Pierwszym widżetem bezstanowym w aplikacji jest sama klasa aplikacji:

```
class MyApp extends statelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

Jak widać, klasa `MyApp` rozszerza `statelessWidget` i zastępuje metodę `build(BuildContext)`. Ta metoda opisuje część interfejsu użytkownika; to znaczy, że tworzy pod nim poddrzewo widżetów.

W opisywanym przykładzie `MyApp` jest elementem głównym (*root*) drzewa widżetów i dlatego tworzy wszystkie widżety w drzewie. W tym przypadku jego bezpośrednim dzieckiem jest `MaterialApp`. Zgodnie z dokumentacją jest to określone w następujący sposób:

> *Widżet, który opakowuje wiele widżetów, powszechnie wymaganych w aplikacjach Material Design.*

`BuildContext` to argument dostarczany do metody `build` jako przydatny sposób interakcji z drzewem widżetów. Umożliwia dostęp do ważnych informacji o przodkach, które pomagają opisać budowany widżet. Pamiętaj, opis zależy tylko od tych informacji kontekstowych i właściwości widżetu, które są zdefiniowane w konstruktorze.

> Widżetom Material Design przyjrzymy się szczegółowo, gdy będziemy badać dostępne wbudowane widżety, a także w rozdziale 6.

Oprócz innych właściwości `MaterialApp` zawiera właściwość `home`, która określa pierwszy widżet wyświetlany jako strona główna aplikacji. Tutaj `home` reprezentuje widżet `MyHomePage`, który jest w tym przykładzie widżetem stanowym.

> Korzystając z klasy `Navigator`, `MaterialApp` umożliwia definiowanie widżetów, które mają być wyświetlane dla określonych ścieżek (*routes*) z logiczną historią nawigacji stron, poprzez zarządzanie stosem wstecznym (ścieżki i nawigację stron będziemy sprawdzać w rozdziale 7.).

Reprezentacja widżetu stanowego za pomocą kodu

`MyHomePage` jest widżetem stanowym, dlatego jest zdefiniowany za pomocą obiektu `State`, `_MyHome ↪PageState`, który zawiera właściwości wpływające na jego wygląd:

```
class MyHomePage extends statefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

  @override
  _MyHomePageState createState() => _MyHomePageState();
}
```

Rozszerzając `statefulWidget`, `MyHomePage` musi zwrócić prawidłowy obiekt `State` w swojej metodzie `createState()`. W naszym przykładzie zwraca instancję `_MyHomePageState`.

> Zwykle widżety stanowe definiują odpowiadające im klasy `State` w tym samym pliku. Ponadto stan jest zazwyczaj prywatny dla biblioteki widżetów, ponieważ klienci zewnętrzni nie muszą bezpośrednio z nią współpracować.

Poniższa klasa _MyHomePageState reprezentuje obiekt State widżetu MyHomePage:

```
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.display1,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ), // Ten końcowy przecinek sprawia, że automatyczne formatowanie jest
      ), // przyjemniejsze.
    );
  }
}
```

Prawidłowy stan widżetu to klasa, która rozszerza klasę frameworku State zdefiniowaną w dokumentacji w następujący sposób:

Logika i stan wewnętrzny dla StatefulWidget.

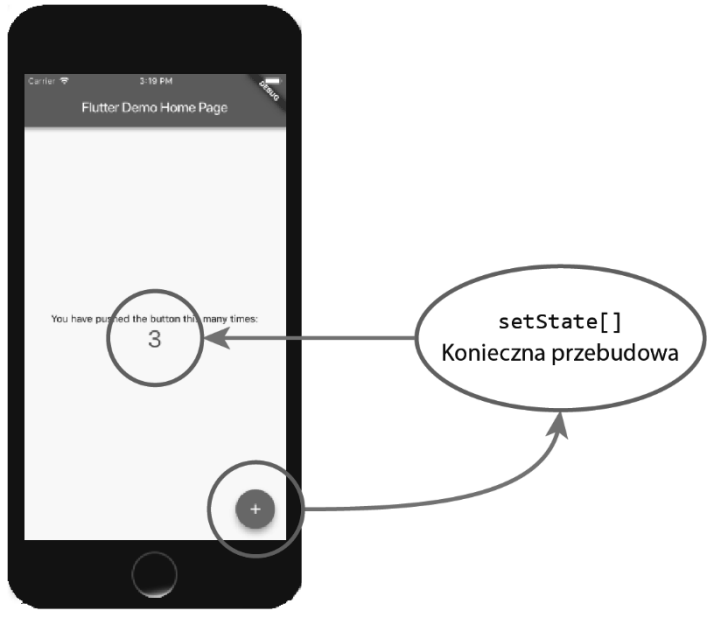
Stan widżetu MyHomePage jest definiowany przez pojedynczą właściwość _counter. Właściwość _counter zachowuje liczbę naciśnięć przycisku w prawym dolnym rogu ekranu. Tym razem za zbudowanie widżetu odpowiada klasa potomna widżetu State. Składa się z widżetu Text, który wyświetla wartość _counter.

`Text` to wbudowany widżet używany do wyświetlania tekstu na ekranie. Więcej informacji o wbudowanych widżetach pojawi się w następnej sekcji.

Widżet stanowy ma zmieniać wygląd w trakcie swojego życia — to znaczy, że to, co go definiuje, zmieni się — dlatego należy go przebudować, aby odzwierciedlał takie zmiany. Tutaj zmiana następuje w metodzie `_incrementCounter()`, która jest wywoływana za każdym razem, gdy naciśnięty zostanie przycisk.

Zwróć uwagę na użycie właściwości `onPressed` widżetu `FloatingActionButton`.

`FloatingActionButton` to pływający przycisk akcji ze stylu Material Design, a ta właściwość odbiera wywołanie zwrotne funkcji, które zostanie wykonane po naciśnięciu:

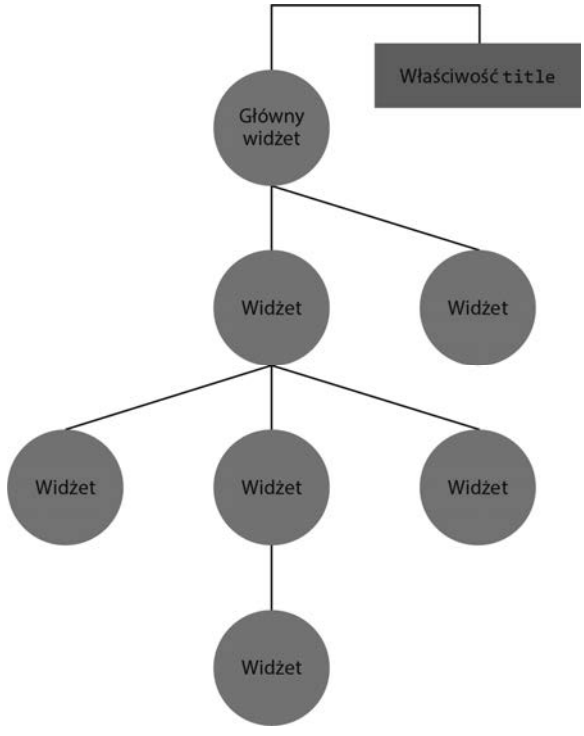


Jest to obraz strony głównej Flutter Demo.
Inne (nakładające się) informacje nie są tutaj ważne

Skąd framework wie, kiedy coś w widżecie się zmienia i musi go przebudować? Odpowiedzią jest `setState`. Ta metoda otrzymuje funkcję jako parametr, w którym należy zaktualizować odpowiedni widżet nawiązujący do `State` (czyli metodę `_incrementCounter`). Wywołując `setState`, framework jest powiadamiany, że musi przebudować widżet. W naszym przykładzie jest wywoływana w celu odzwierciedlenia nowej wartości właściwości `_counter`.

Widżety dziedziczone

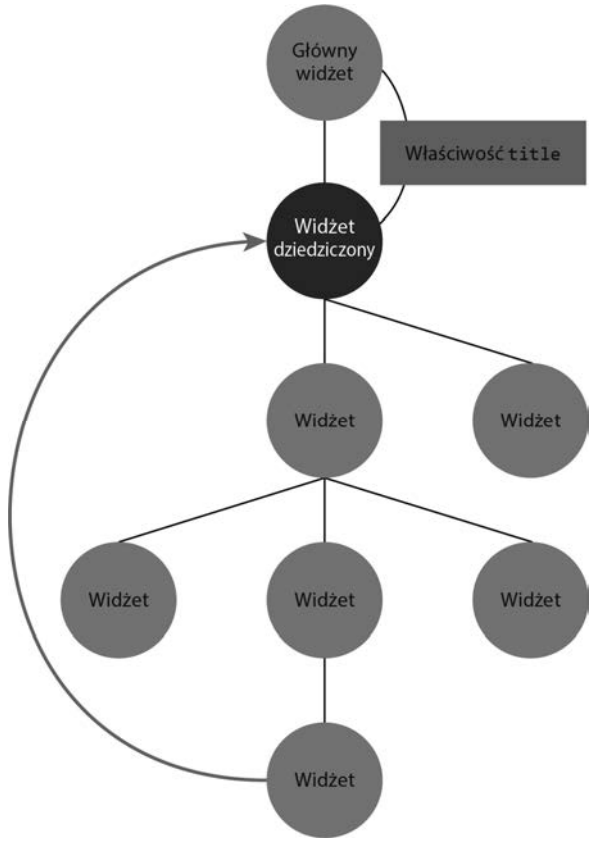
Oprócz `statelessWidget` i `statefulWidget` we frameworku Fluttera istnieje jeszcze jeden typ widżetu, `InheritedWidget`. Czasami jeden widżet może potrzebować dostępu do danych z góry drzewa i w takim przypadku musielibyśmy replikować informacje w dół do interesującego widżetu. Ten proces przedstawiono na poniższym schemacie:



Załóżmy, że niektóre widżety w drzewie muszą uzyskać dostęp do właściwości `title` z poziomu widżetu głównego. Aby to zrobić, za pomocą `statelessWidget` lub `statefulWidget` musielibyśmy zreplikować właściwość w odpowiednich widżetach i przekazać ją przez konstruktor. Replikowanie właściwości we wszystkich widżetach podrzędnych może być denerwujące.

Aby rozwiązać ten problem, Flutter udostępnia klasę `InheritedWidget`, pomocniczy rodzaj widżetu, który pomaga propagować informacje w dół drzewa, jak pokazano na diagramie na następnej stronie.

Jeśli dodamy `InheritedWidget` do drzewa, każdy widżet znajdujący się poniżej może uzyskać dostęp do danych, udostępnionych za pomocą metody `inheritFromWidgetOfExactType(InheritedWidget)` klasy `BuildContext`, która otrzymuje typ `InheritedWidget` jako parametr i używa drzewa do znalezienia pierwszego przodka widżetu żądanego typu.



Istnieje kilka bardzo częstych przypadków użycia `InheritedWidget` we Flutterze. Jednym z najczęstszych zastosowań jest klasa `Theme`, która pomaga opisać kolory dla całej aplikacji. Przyjrzymy się temu w rozdziale 5.

Właściwość key widżetu

Jeśli spojrzysz na oba konstruktory klas `statelessWidget` i `statefulWidget`, zauważysz parametr o nazwie `key`. Jest to ważna właściwość dla widżetów we Flutterze. Pomaga w renderowaniu z drzewa widżetów do drzewa elementów. Oprócz typu i odniesienia do odpowiedniego widżetu element ten zawiera również klucz identyfikujący widżet w drzewie. Właściwość `key` pomaga zachować stan widżetu między przebudowami. Najczęstszym zastosowaniem `key` jest sytuacja, w której mamy do czynienia z kolekcjami widżetów tego samego typu; bez kluczy drzewo elementów nie wiedziałoby, który stan odpowiada danemu widżetowi, ponieważ wszystkie miałyby ten sam typ. Na przykład za każdym razem, gdy widżet zmienia swoją pozycję lub poziom w drzewie widżetów, w drzewie elementów następuje dopasowanie, pozwalające zobaczyć, co należy zaktualizować na ekranie, aby odzwierciedlić nową strukturę widżetu. Gdy widżet

ma stan, wymaga, aby odpowiedni stan został z nim przeniesiony. Krótko mówiąc, to właśnie klucz pomaga frameworkowi to robić. Trzymając wartość klucza, dany element będzie znał odpowiedni stan widżetu.

> W dalszej części książki będziemy używać w naszej aplikacji kluczy. Jeśli chcesz teraz znaleźć więcej szczegółów na temat wpływu `key` na widżet i dostępnych typów kluczy, zapoznaj się z oficjalnym wprowadzeniem do kluczy w dokumentacji: *https://flutter.io/docs/development/ui/widgets-intro#keys*.

Widżety wbudowane

Flutter kładzie duży nacisk na interfejs użytkownika, dlatego zawiera duży katalog widżetów, które umożliwiają tworzenie niestandardowych interfejsów zgodnie z Twoimi potrzebami. Dostępne widżety Fluttera obejmują zarówno proste elementy, widżet `Text` (przykład aplikacji licznika), jak i złożone widżety, które pomagają projektować dynamiczny interfejs użytkownika z animacjami i obsługą wielu gestów.

Widżety podstawowe

Widżety podstawowe we Flutterze są dobrym punktem wyjścia, nie tylko ze względu na łatwość użycia, ale także dlatego, że demonstrują moc i elastyczność frameworka, nawet w prostych przypadkach.

Nie będziemy studiować wszystkich dostępnych widżetów, ponieważ zniweczyłoby to cel tej książki, dlatego wymienimy tylko niektóre z nich dla Twojej wiedzy, a część z nich będziemy wykorzystywać w praktyce, abyś mógł nauczyć się podstaw w celu dalszego poszerzania wiedzy.

Widżet Text

`Text` wyświetla ciąg tekstu w dowolnym stylu:

```
Text(
  "To jest tekst",
)
```

Najczęściej używane właściwości widżetu Text są następujące:

- `style` — klasa określająca style tekstu. Udostępnia właściwości, które pozwalają na zmianę koloru tekstu, tła, rodziny czcionek (umożliwiając użycie niestandardowej czcionki z zasobów; zobacz rozdział 3.), wysokość linii, rozmiar czcionki i tak dalej.
- `textAlign` — kontroluje wyrównanie tekstu w poziomie. Dostępne są opcje takie jak wyśrodkowanie lub wyjustowanie.
- `maxLines` — umożliwia określenie maksymalnej liczby wierszy tekstu, które zostaną obcięte w przypadku przekroczenia limitu.

- `overflow` — określa, w jaki sposób tekst zostanie obcięty w przypadku przepełnienia. Dostępne są opcje takie jak limit maksymalnej liczby wierszy. Tę właściwość można wykorzystać, aby np. na końcu zdania dodać wielokropek.

> Aby zobaczyć wszystkie dostępne właściwości widżetu `Text`, przejdź na oficjalną stronę z dokumentacją widżetu: *https://docs.flutter.io/flutter/widgets/Text-class.html*.

Widżet Image

`Image` wyświetla obraz z różnych źródeł i formatów. Obsługiwane w dokumentach formaty obrazu to JPEG, PNG, GIF, animowany GIF, WebP, animowany WebP, BMP i WBMP:

```
Image(
  image: AssetImage(
    "assets/dart_logo.jpg"
  ),
)
```

Właściwość `Image` widżetu określa `ImageProvider`. Wyświetlany obraz może pochodzić z różnych źródeł. Klasa `Image` zawiera różne konstruktory dla różnych sposobów ładowania obrazów:

- `Image` (*https://api.flutter.dev/flutter/widgets/Image/Image.html*), do uzyskania obrazu z `ImageProvider` (*https://api.flutter.dev/flutter/painting/ImageProvider-class.html*), jak w poprzednim przykładzie.
- `Image.asset` (*https://api.flutter.dev/flutter/widgets/Image/Image.asset.html*) tworzy `AssetImage`, który służy do uzyskania obrazu z `AssetBundle` (*https://api.flutter.dev/flutter/ services /AssetBundle-class.html*) przy użyciu klucza zasobu. Przykład jest następujący.

  ```
  Image.asset(
    'assets/dart_logo.jpg',
  )
  ```

- `Image.network` (*https://api.flutter.dev/flutter/widgets/Image/Image.network.html*) tworzy `NetworkImage` w celu uzyskania obrazu z adresu URL.

  ```
  Image.network(
    'https://picsum.photos/250?image=9',
  )
  ```

- `Image.file` (`https://api.flutter.dev/flutter/widgets/Image/Image.file.html`) tworzy `FileImage` w celu uzyskania obrazu z pliku (*https://api.flutter.dev/flutter/dart- io/File-class.html*).

  ```
  Image.file(
    File(file_path)
  )
  ```

- `Image.memory` (*https://api.flutter.dev/flutter/widgets/Image/Image.memory.html*) tworzy `MemoryImage` w celu uzyskania obrazu z `Uint8List` (*https://api.flutter.dev/flutter/dart-typed_data/Uint8List-class.html*).

```
Image.memory(
  Uint8List(image_bytes)
)
```

Oprócz `Image` istnieją inne powszechnie używane właściwości:

- `height/width` — aby określić ograniczenia rozmiaru obrazu;
- `repeat` — aby kopiować obraz, w celu wypełnienia dostępnego miejsca;
- `aligment` — aby wyrównać obraz w określonej pozycji w jego granicach;
- `fit` — aby określić, w jaki sposób obraz powinien zostać umieszczony w dostępnej przestrzeni.

> Aby zobaczyć wszystkie dostępne właściwości widżetu `Image`, przejdź do oficjalnej strony dokumentacji widżetu obrazka: *https://docs.flutter.io/flutter/widgets/Image-class.html*.

Material Design i widżety iOS Cupertino

Wiele widżetów we Flutterze wywodzi się w pewien sposób z wytycznych specyficznych dla platformy: **Material Design** lub **iOS Cupertino**. Pomaga to deweloperowi w możliwie najłatwiejszym przestrzeganiu wytycznych specyficznych dla platformy.

> Jeśli nie znasz wytycznych Material Design lub iOS Cupertino, to dobry czas, aby je poznać:
> Material Design: *https://material.io/guidelines/material-design/introduction.html*;
> iOS Cupertino: *https://developer.apple.com/design/human-interface-guidelines/*.

Na przykład Flutter nie ma widżetu `Button`; zamiast tego zapewnia alternatywne implementacje przycisków za pomocą wytycznych Google Material Design i iOS Cupertino.

> Nie zamierzamy zagłębiać się w każdą właściwość lub zachowanie widżetu, ponieważ można je łatwo przestudiować, uruchamiając przykłady lub zaglądając do dokumentacji. Możesz również sprawdzić aplikację Flutter Gallery w Google Play (*https://play.google.com/store/apps/details?id=io.flutter.demo.gallery*), aby znaleźć krótką i fajną demonstrację dostępnych widżetów.

Buttony

Po stronie Material Design Flutter implementuje następujące komponenty przycisków:

- `RaisedButton` — wypukły przycisk Material Design. Wypukły przycisk składa się z prostokątnego kawałka materiału, który unosi się nad interfejsem.
- `FloatingActionButton` — pływający przycisk akcji to okrągły przycisk z ikoną, który znajduje się nad zawartością w celu promowania podstawowej akcji w aplikacji.

- `FlatButton` — płaski przycisk to sekcja wydrukowana na widżecie Material, która reaguje na dotknięcia przez rozpryskiwanie / falowanie kolorem.
- `IconButton` — przycisk z ikoną to obrazek wydrukowany na widżecie Material, który reaguje na dotknięcie przez rozpryskiwanie / falowanie.

Na podstawie wytycznych dotyczących Material Design działanie komponentu Ink można wyjaśnić w następujący sposób:

Element zapewniający promieniujący efekt w postaci wizualnego tętnienia rozszerzającego się na zewnątrz pod wpływem dotyku użytkownika.

- `DropDownButton` — pokazuje aktualnie wybrany element i strzałkę, która otwiera menu do wyboru innego elementu.
- `PopUpMenuButton` — wyświetla menu po naciśnięciu.

W przypadku stylu Cupertino systemu iOS Flutter udostępnia klasę `CupertinoButton`.

> Ze względu na wytyczne Material Design, elewację, efekty ink i efekty świetlne, widżety Material Design są nieco droższe niż widżety Cupertino. Nie jest to duży problem, ale warto mieć tego świadomość.

Scaffold

`Scaffold` implementuje podstawową strukturę układu wizualnego Material Design lub iOS Cupertino. W przypadku zastosowania Material Design widżet `Scaffold` może zawierać wiele komponentów Material Design:

- `body` — podstawowa zawartość scaffold. Jest wyświetlany poniżej paska `AppBar`, jeśli istnieje.
- `AppBar` — pasek aplikacji składa się z paska narzędzi i potencjalnie innych widżetów.
- `TabBar` — widżet Material Design, który wyświetla poziomy rząd zakładek. Jest to zwykle używane jako część `AppBar`.
- `TabBarView` — widok strony, który wyświetla widżet odpowiadający aktualnie wybranej karcie. Zwykle używany w połączeniu z `TabBar` i używany jako widżet `body`.
- `BottomNavigationBar` — dolne paski nawigacyjne ułatwiają przeglądanie i przełączanie się między widokami najwyższego poziomu za pomocą jednego dotknięcia.
- `Drawer` — panel Material Design, który przesuwa się poziomo od krawędzi scaffold, aby wyświetlić łącza nawigacyjne w aplikacji.

W iOS Cupertino struktura jest inna z określonymi przejściami i zachowaniami.

Dostępne klasy iOS Cupertino to `CupertinoPageScaffold` i `CupertinoTabScaffold`, które składają się zazwyczaj z następujących elementów:

- `CupertinoNavigationBar` — górny pasek nawigacyjny. Zwykle używany z `CupertinoPageScaffold`.
- `CupertinoTabBar` — dolny pasek zakładek, który zwykle używany z `CupertinoTabScaffold`.

Dialogi

Flutter implementuje zarówno okna dialogowe Material Design, jak i Cupertino. Po stronie Material Design są to `SimpleDialog` i `AlertDialog`; po stronie Cupertino są to `CupertinoDialog` i `CupertinoAlertDialog`.

Pola tekstowe

Pola tekstowe są również zaimplementowane w obu wytycznych, przez widżet `TextField` w Material Design oraz przez widżet `CupertinoTextField` w iOS Cupertino. Oba wyświetlają klawiaturę do wprowadzania danych przez użytkownika. Niektóre z ich wspólnych właściwości są następujące:

- `autofocus` — określa, czy pole `TextField` powinno być ustawiane automatycznie (jeśli nic innego nie jest już ustawione);
- `enabled` — pozwala ustawić pole jako edytowalne lub nie;
- `keyboardType` — pozwala zmienić typ klawiatury wyświetlanej użytkownikowi podczas edycji.

> Aby zobaczyć wszystkie dostępne właściwości widżetów `TextField` i `CupertinoTextField`, przejdź do oficjalnej strony dokumentacji widżetów: *https://api.flutter.dev/flutter/material/TextField-class.html* i *https://api.flutter.dev/flutter/cupertino/CupertinoTextField-class.html*.

Widżety wyboru

W Material Design dostępne są następujące widżety wyboru:

- `Checkbox` umożliwia wybór wielu opcji na liście.
- `Radio` umożliwia pojedynczy wybór z listy opcji.
- `Switch` umożliwia przełączanie (włączanie / wyłączanie) pojedynczej opcji.
- `Slider` umożliwia wybór wartości w zakresie poprzez przesuwanie suwaka.

W przypadku iOS Cupertino niektóre z tych funkcji widżetów nie istnieją; są jednak dostępne alternatywy:

- `CupertinoActionSheet` — modalny arkusz akcji w stylu iOS umożliwiający wybór opcji spośród wielu.
- `CupertinoPicker` — również kontrolka selektora. Służy do wybierania pozycji z krótkiej listy.
- `CupertinoSegmentedControl` — zachowuje się jak przycisk opcji, w którym wybór jest pojedynczą pozycją z listy opcji.

- `CupertinoSlider` — podobny do `Slider` w Material Design.
- `CupertinoSwitch` — działanie podobne do `Switch` z Material Design.

Selektory daty i godziny

W przypadku Material Design Flutter udostępnia selektory daty i godziny za pośrednictwem funkcji `showDatePicker` i `showTimePicker`, które budują i wyświetlają okno dialogowe Material Design dla odpowiednich akcji. Po stronie Cupertino iOS dostępne są widżety `CupertinoDatePicker` i `CupertinoTimerPicker`, zgodnie z poprzednim stylem `CupertinoPicker`.

Inne składniki

Istnieją również komponenty specyficzne dla projektu, które są unikalne dla każdej platformy. Na przykład Material Design obejmuje koncepcję **Kart**, która w dokumentacji jest zdefiniowana w następujący sposób:

Arkusz używany do przedstawienia pewnych powiązanych informacji.

Z drugiej strony widżety specyficzne dla Cupertino mogą mieć unikalne przejścia obecne w świecie iOS.

> Więcej informacji można znaleźć w katalogu widżetów Fluttera na stronie *flutter.io*: *https://flutter.io/docs/development/ui/widgets*.

Wprowadzenie do wbudowanych widżetów layoutu

Wydaje się, że niektóre widżety nie pojawiają się na ekranie dla użytkownika, ale jeśli znajdują się w drzewie widżetów, w jakiś sposób tam będą, wpływając na wygląd widżetu podrzędnego (na przykład na jego położenie lub styl).

Aby na przykład umieścić przycisk w dolnym rogu ekranu, moglibyśmy określić pozycję związaną z ekranem, ale jak być może zauważyłeś, przyciski i inne widżety nie mają właściwości `Position`. Możesz więc zadawać sobie pytanie: „Jak są zorganizowane widżety na ekranie?”. Odpowiedzią są znowu widżety. Zgadza się! Flutter dostarcza widżety do komponowania samego layoutu, z pozycjonowaniem, skalowaniem, stylizacją i tak dalej.

Kontenery

Wyświetlanie pojedynczego widżetu na ekranie nie jest dobrym pomysłem na organizację interfejsu użytkownika. Zwykle tworzymy listę widżetów, które są zorganizowane w określony sposób; w tym celu używamy kontenerów widżetów.

Najpopularniejszymi kontenerami we Flutterze są widżety `Row` i `Column`. Mają właściwość `children`, która wymaga, aby lista widżetów była wyświetlana w określonym kierunku (to jest pozioma lista w przypadku `Row` lub pionowa lista w przypadku `Column`).

Innym szeroko stosowanym widżetem jest widżet `Stack`, organizujący dzieci w warstwy, w których jedno może częściowo lub całkowicie nakładać się na drugie.

Jeśli wcześniej tworzyłeś jakąś aplikację mobilną, być może korzystałeś już z list i siatek. Flutter zapewnia klasy dla obu z nich: mianowicie widżety `ListView` i `GridView`. Dostępne są również inne, mniej typowe, ale mimo to ważne widżety kontenerów, takie jak `Table`, który organizuje elementy podrzędne w układzie tabelarycznym.

Stylizacja i pozycjonowanie

Zadanie pozycjonowania widżetu podrzędnego w kontenerze, na przykład widżetu `Stack`, jest wykonywane przy użyciu innych widżetów. Flutter zapewnia widżety do bardzo konkretnych zadań. Wyśrodkowanie widżetu w kontenerze odbywa się poprzez umieszczenie go w widżecie `Center`. Wyrównanie widżetu podrzędnego względem elementu nadrzędnego można wykonać za pomocą widżetu `Align`, w którym żądane położenie można określić za pomocą jego właściwości `aligment`. Kolejnym przydatnym widżetem jest `Padding`, dzięki któremu możemy określić przestrzeń wokół danego dziecka. Funkcjonalności tych widżetów są zagregowane w widżecie `Container`, który łączy te wspólne widżety pozycjonowania i stylizacji, aby zastosować je bezpośrednio do dziecka, dzięki czemu kod jest znacznie czystszy i krótszy.

Inne widżety (gesty, animacje i transformacje)

Flutter zapewnia widżety do wszystkiego, co jest związane z interfejsem użytkownika. Na przykład gesty, takie jak przewijanie lub dotykanie, będą powiązane z widżetem, który zarządza gestami. Animacjami i transformacjami, takimi jak skalowanie i obrót, również zarządzają określone widżety. Niektóre z nich szczegółowo omówimy w kolejnych rozdziałach, kiedy będziemy opracowywać części małej aplikacji.

Nie jesteśmy w stanie zbadać wszystkich dostępnych widżetów i wszystkich ich możliwych kombinacji. Swoją podróż zaczniemy od opracowania małej aplikacji w następnej sekcji, w której zbadamy niektóre z dostępnych widżetów we wszystkich kategoriach, abyś mógł sobie zwizualizować, jak korzystać z niektórych z nich. Co najważniejsze, nauczysz się podstaw tworzenia layoutów we Flutterze. Gdy to zrobisz, poznanie nowych i konkretnych widżetów będzie łatwym zadaniem.

> Podczas pisania tej książki Flutter rozwija kolejną wspaniałą funkcję, **widok platformy** (*platform view*), która pozwala nam wykorzystywać wszelkie natywne interfejsy, dostępne już w iOS i Androidzie. Przeczytasz o tym więcej w rozdziale 11., Widoki platformy oraz integracja mapy, w sekcji Wyświetlanie mapy.

Tworzenie interfejsu użytkownika za pomocą widżetów (aplikacja do zarządzania przysługami)

Teraz, gdy znamy już niektóre z dostępnych widżetów Fluttera, czas zacząć tworzyć małą aplikację, którą będziemy rozbudowywać w trakcie dalszej lektury.

Aplikacja, którą zamierzamy opracować, będzie służyła do zarządzania przysługami. Będzie to mała sieć, w której przyjaciel może prosić innego przyjaciela o przysługę, a ten może ją przyjąć lub odmówić jej wykonania. Zaakceptowanie przysługi powoduje jej wpisanie na listę rzeczy do zrobienia. To jak aplikacja notująca rzeczy do zrobienia, w której zadania są proponowane przez znajomych użytkownika i tylko użytkownik może je zaakceptować lub odrzucić. W tej aplikacji poznamy wiele koncepcji, które mogą pomóc w tworzeniu aplikacji.

W kolejnych rozdziałach będziemy dodawać funkcje do aplikacji, stopniowo poznając różne elementy składające się na aplikację Fluttera.

Ekrany aplikacji

Aplikacja *Friend Favors* będzie się składać z dwóch ekranów. W obu z nich będziemy korzystać z komponentów Material Design dostarczonych przez Fluttera. Na pierwszym ekranie pojawi się lista przysług, a na drugim formularz proszenia znajomego o przysługę. Na razie będziemy używać list w pamięci; oznacza to, że informacje nie będą przechowywane w żadnym innym miejscu niż aplikacja.

Kod aplikacji

Kod aplikacji nie jest jeszcze w pełni funkcjonalny. Jest wystarczająco mały, aby stworzyć layout. Tworzy instancję widżetu `MaterialApp`, która ustawia ekran główny na stronę z listą przysług o nazwie `FavorsPage`:

```
class MyApp extends statelessWidget {
  // na razie używając pozorowanych wartości z pliku mock_favors
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      home: FavorsPage(
        pendingAnswerFavors: mockPendingFavors,
        completedFavors: mockCompletedFavors,
        refusedFavors: mockRefusedFavors,
        acceptedFavors: mockDoingFavors,
      ),
    );
  }
}
```

`MaterialApp` to widżet, który udostępnia przydatne narzędzia dla całej aplikacji. Jednym z nich jest widżet `Theme`, który pozwala nam zmieniać style i kolory naszych aplikacji zgodnie z wytycznymi Material Design. Innym przydatnym narzędziem jest widżet `Navigator`, który zarządza zestawem widżetów aplikacji w sposób podobny do stosu nawigacyjnego, na którym możemy nawigować do ekranu lub wstecz. W aplikacji będziemy używać obu widżetów. `Navigator` zastosowaliśmy już, gdy ustawialiśmymy właściwość `home` widżetu `MaterialApp`. `Navigator` działa na zasadzie ścieżek (*route*) do widżetu. Oznacza to, że istnieje kilka sposobów definiowania określonych ścieżek wskazujących na określone widżety, a kiedy nawigujemy do określonej ścieżki, `Navigator` będzie mógł nawigować do odpowiedniego widżetu. Ustawiając właściwość `home` w jakimś widżecie, mówimy, że `Navigator` używa tego widżetu za pomocą ścieżki '/'.

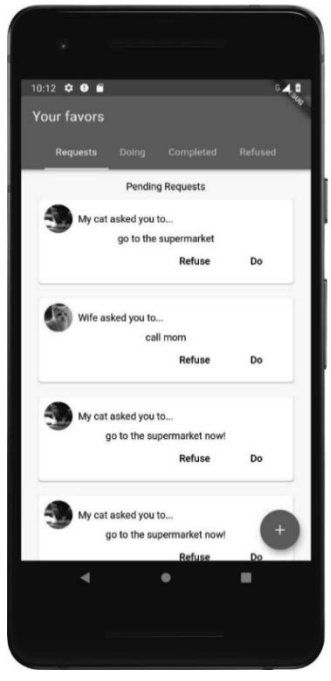
> Jak widać, widżet `FavorsPage` ma wypełnione niektóre parametry konstruktora. Aby zobaczyć, czym one są, czytaj dalej.

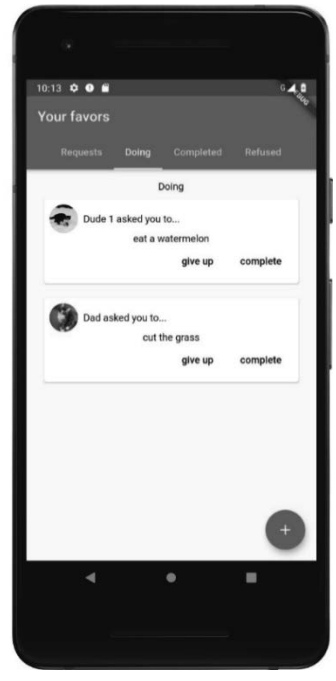
Na pierwszym etapie przyjrzymy się początkowej strukturze układu aplikacji, która będzie ewoluować do końca książki wraz z nowymi stylami i widżetami. W następnym rozdziale dowiesz się, jak dodać niektóre metody wprowadzania danych przez użytkownika za pomocą dotknięć i pól formularzy. Później, w rozdziale 6., zobaczymy, jak dostosować wygląd aplikacji za pomocą motywu. Zacznijmy więc od przyjrzenia się layoutom ekranu.

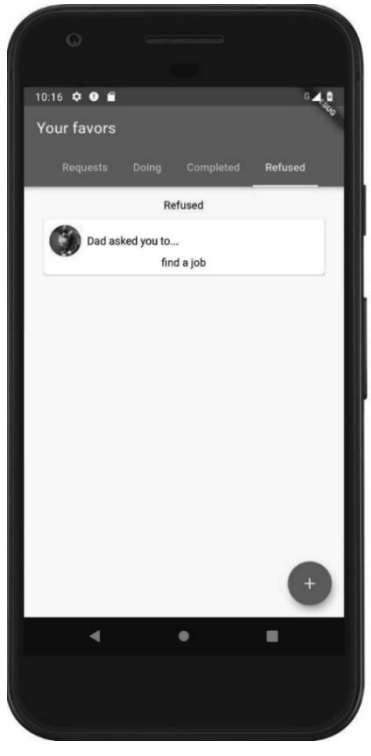
Ekran główny aplikacji

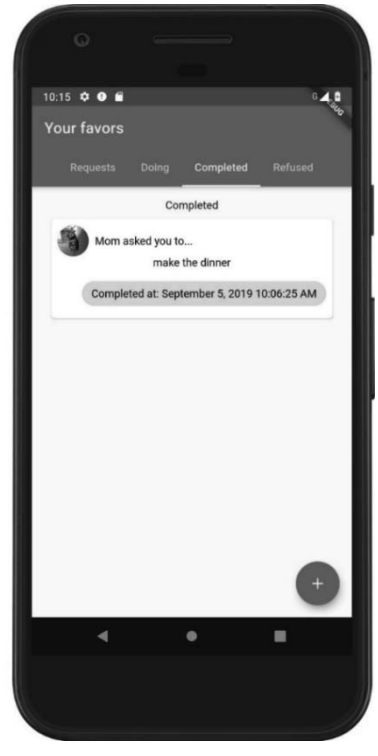
Pierwszy ekran aplikacji to ekran główny, który będzie składał się z czterech zakładek z listą przysług i ich statusami:

- **Oczekujące przysługi** — przysługi, o które prosili niektórzy przyjaciele, a na które jeszcze nie odpowiedzieliśmy.
- **W toku / spełnianie przysług** — przyjęte przysługi czyli takie, którymi zajmujemy się teraz — zobacz pierwszy rysunek na następnej stronie.
- **Ukończone przysługi** — już ukończone przysługi.
- **Odmowa przysług** — lista przysług, których zrobienia odmówiliśmy (nie przyjęliśmy) — zobacz drugi rysunek na następnej stronie.









Lista będzie zawierać wszystkie przysługi, rozdzielone według kategorii. U góry layoutu mamy instancję `TabBar`, która posłuży do zmiany zakładki na żądaną listę. Następnie na każdej zakładce mamy listę elementów `Card`, które zawierają akcje odpowiadające jej kategorii.

> Stworzyliśmy klasy `Friend` i `Favor` reprezentujące dane aplikacji. Możesz przyjrzeć się temu bliżej w kodzie źródłowym rozdziału (katalog `hands_on_layouts`) tej książki. Tutaj są to proste klasy danych, które nie zawierają żadnej zaawansowanej logiki biznesowej.

Ponadto pływający przycisk akcji na dole ekranu powinien przekierowywać do ekranu **Request a favor**, gdzie użytkownik będzie mógł poprosić znajomych o przysługę.

Kod layoutu

Przede wszystkim zdefiniujemy naszą stronę główną jako instancję `statelessWidget`, ponieważ teraz zależy nam tylko na layoucie i nie mamy do wykonania żadnych działań, które spowodowałyby zmianę stanu. Dlatego widżet nadrzędny `MyApp` przekazuje wartości do zdefiniowanych pól listy.

Pamiętaj, że gdy widżet jest bezstanowy, jego opis jest definiowany przez widżet nadrzędny podczas jego tworzenia. Pokazuje to poniższy kod:

```
class FavorsPage extends statelessWidget {
  // na razie używając pozorowanych wartości z pliku mock_favors
  final List<Favor> pendingAnswerFavors;
  final List<Favor> acceptedFavors;
  final List<Favor> completedFavors;
  final List<Favor> refusedFavors;

  FavorsPage({
    Key key,
    this.pendingAnswerFavors,
    this.acceptedFavors,
    this.completedFfavors,
    this.refusedFavors,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {...} // dla zwięzłości
}
```

Jak pokazano w poprzednim kodzie, widżet jest definiowany przez listy specyficzne dla przysług. Zwróć także uwagę na parametr `key`. Chociaż nie jest to tutaj naprawdę potrzebne, dobrą praktyką jest zdefiniowanie parametru.

Rzućmy okiem na metodę `build()`, aby zobaczyć, jak zbudowany jest widżet:

```
@override
Widget build(BuildContext context) {
  return DefaultTabController(
    length: 4,
```

```
      child: Scaffold(
        appBar: AppBar(
          title: Text("Your favors"),
          bottom: TabBar(
            isScrollable: true,
            tabs: [
              _buildCategoryTab("Requests"),
              _buildCategoryTab("Doing"),
              _buildCategoryTab("Completed"),
              _buildCategoryTab("Refused"),
            ],
          ),
        ),
        body: TabBarView(
          children: [
            _favorsList("Pending Requests", pendingAnswerFavors),
            _favorsList("Doing", acceptedFavors),
            _favorsList("Completed", completedFavors),
            _favorsList("Refused", refusedFavors),
          ],
        ),
        floatingActionButton: FloatingActionButton(
          onPressed: () {},
          tooltip: 'Ask a favor',
          child: Icon(Icons.add),
        ),
      ),
    );
  }
```

Pierwszym widżetem obecnym w poddrzewie widżetów `FavorsPage` jest widżet `Default` ➥`TabController`, który obsługuje za nas zmianę zakładki. Następnie mamy widżet `Scaffold`, który implementuje podstawową strukturę Material Design. Tutaj używamy już niektórych z tych elementów, w tym paska aplikacji i pływającego przycisku akcji. Ten widżet jest bardzo przydatny do projektowania aplikacji zgodnych z Material Design, ponieważ zapewnia przydatne właściwości w oparciu o wytyczne:

- W `AppBar` dodaliśmy tytuł za pomocą widżetu `Text`. W niektórych przypadkach możemy również dodać do niego akcje lub niestandardowy layout. Tutaj dodaliśmy instancję `TabBar` u dołu (`bottom`) paska aplikacji, która pokaże dostępne karty.
- We `FloatingActionButton` również nic zbytnio nie zmieniliśmy; dodaliśmy ikonę tylko za pomocą widżetu `Icon`, który zawiera ikonę Material Design dostarczoną przez platformę.
- Właściwość `body` widżetu `Scaffold` to miejsce, w którym projektujemy sam layout. Jest zdefiniowany w następujący sposób: widżet `TabBarView` wyświetla odpowiedni widżet dla wybranej karty w zdefiniowanej wcześniej instancji `DefaultTabController`. Tym, co wymaga uwagi, jest jego własność `children`; dopasowuje zakładki i zwraca odpowiedni widżet skojarzony z daną zakładką.

Elementy paska zakładek `Tab` są tworzone przez metodę `_buildCategoryChip()` w następujący sposób:

```
class FavorsPage extends statelessWidget {
  // … pola metody budowania i inne
  Widget _buildCategoryTab(String title) {
    return Tab(
      child: Text(title),
    );
  }
}
```

Jak widać, funkcja tworzy element zakładek dla kategorii, po prostu budując poddrzewo `Tab > Text`, gdzie `title` jest identyfikatorem elementu.

W ten sam sposób każda sekcja listy przysług jest definiowana w swojej metodzie `_favorsList()`:

```
class FavorsPage extends statelessWidget {
  // ... pola, metody budowania i inne

  Widget _favorsList(String title, List<Favor> favors) {
    return Column(
      mainAxisSize: MainAxisSize.max,
      children: <Widget>[
        Padding(
          child: Text(title),
          padding: EdgeInsets.only(top: 16.0),
        ),
        Expanded(
          child: ListView.builder(
            physics: BouncingScrollPhysics(),
            itemCount: favors.length,
            itemBuilder: (BuildContext context, int index) {
              final favor = favors[index];
              return Card(
                key: ValueKey(favor.uuid),
                margin: EdgeInsets.symmetric(vertical: 10.0,
                horizontal: 25.0),
                child: Padding(
                  child: Column(
                    children: <Widget>[
                        _itemHeader(favor),
                        Text(favor.description),
                        _itemFooter(favor)
                    ],
                  ),
                  padding: EdgeInsets.all(8.0),
                ),
              );
            },
          ),
        ),
      ],
    );
  }
}
```

Widżet sekcji przysług jest reprezentowany przez widżet `Column`, który ma dwa widżety podrzędne:

- widżet `Text` (z elementem nadrzędnym `Padding`) zawierający tytuł sekcji, jak poprzednio;
- instancję `ListView`, które będzie zawierać każdy z elementów przysług.

Ta lista jest zbudowana w inny sposób niż poprzednie. Tutaj użyliśmy konstruktora nazwanego `ListView.builder()`. Oczekuje on instancji `itemCount` i `itemBuilder`, które definiujemy za pomocą listy przekazanej jako argument w wywołaniu funkcji `_favorsList()`:

- `itemCount` to po prostu rozmiar listy;
- `itemBuilder` musi być funkcją, która zwraca widżet odpowiadający elementowi w określonej pozycji. Ta funkcja otrzymuje `BuildContext`, podobnie jak metoda `build()` widżetu, a także pozycję indeksu (tutaj użyliśmy argumentu `index`, aby uzyskać odpowiednią przysługę z listy).

Ta forma tworzenia elementów jest optymalna w przypadku list dużych, takich, które rosną w trakcie cyklu życia, a nawet tych przewijanych w nieskończoność (które być może już widziałeś w niektórych aplikacjach), ponieważ buduje przedmioty tylko wtedy, gdy są potrzebne, zapobiegając marnowaniu zasobów. obliczeniowych.

> Zmiana fizyki listy za pomocą `BouncingScrollPhysic()` powoduje, że lista ma efekt odbijania przewijania widoczny na listach systemu iOS.

Wartość funkcji `itemBuilder` tworzy widżet `Card` dla każdej przysługi na liście argumentów, pobierając odpowiednią pozycję za pomocą `final favor = favors [index];`.

Pozostała część kreatora listy wygląda następująco:

```
return Card(
    key: ValueKey(favor.uuid),
    margin: EdgeInsets.symmetric(vertical: 10.0, horizontal: 25.0),
    child: Padding(
      child: Column(
        children: <Widget>[
          _itemHeader(favor),
          Text(favor.description),
          _itemFooter(favor)
        ],
      ),
    padding: EdgeInsets.all(8.0),
  ),
);
```

Kiedy mówimy o elementach listy, zawsze będziemy potrzebować wartości `key` widżetu, przynajmniej jeśli dodamy do niego obsługę zdarzenia wyboru. Dzieje się tak, ponieważ listy we Flutterze mogą obiegać wiele elementów podczas zdarzeń przewijania, a dodając klucz, będziemy twierdzić, że określony widżet ma powiązany z nim określony stan.

Nowy element, który wystąpił, to właściwość `margin` widżetu `Card`, która dodaje margines do widżetu. W tym przypadku dodajemy `10,0` dip (*Density-independent Pixels*) dla góry i dołu oraz `25,0` dla lewej i prawej strony. Jego dziecko `body` jest podzielone na trzy części:

- Najpierw jest nagłówek, pokazujący znajomego, który wysłał prośbę o przysługę, zdefiniowany w funkcji `_itemHeader()`.

```
Row _itemHeader(Favor favor) {
  return Row(
    children: <Widget>[
      CircleAvatar(
        backgroundImage: NetworkImage(
          favor.friend.photoURL,
        ),
      ),
      Expanded(
        child: Padding(
            padding: EdgeInsets.only(left: 8.0),
            child: Text("${favor.friend.name} asked you to...
            ")),
      )
    ],
  );
}
```

Nagłówek jest zdefiniowany jako poddrzewo `Row> [CircleAvatar, Expanded]`. Zaczyna się od definicji `Row` (działa jak widżet `Column`, ale na osi poziomej), która ma instancję `CircleAvatar`, czyli okrągły obraz reprezentujący użytkownika. Tutaj użyliśmy dostawcy `NetworkImage`; po prostu przekazujemy do niego adres URL obrazu i pozwalamy mu się załadować. Pozostała przestrzeń widżetu `Row` jest używana przez `Text` z pewną wartością `Padding`, która pokazuje imię znajomego.

- Po drugie, istnieje treść, która jest po prostu widżetem Text z opisem przysługi.
- Na końcu jest stopka, która zawiera dostępne akcje dla prośby o przysługę w zależności od kategorii przysługi, zdefiniowanej w funkcji `_itemFooter()`.

```
Widget _itemFooter(Favor favor) {
  if (favor.isCompleted) {
    final format = DateFormat();
    return Container(
      margin: EdgeInsets.only(top: 8.0),
      alignment: Alignment.centerRight,
      child: Chip(
        label: Text("Completed at:
  ${format.format(favor.completed)}"),
        ),
     );
  }
  if (favor.isRequested) {
    return Row(
      mainAxisAlignment: MainAxisAlignment.end,
      children: <Widget>[
        FlatButton(
          child: Text("Refuse"),
          onPressed: () {},
```

```
            ),
            FlatButton(
              child: Text("Do"),
              onPressed: () {},
            )
          ],
        );
      }
      if (favor.isDoing) {
        return Row(
          mainAxisAlignment: MainAxisAlignment.end,
          children: <Widget>[
            FlatButton(
              child: Text("give up"),
              onPressed: () {},
            ),
            FlatButton(
              child: Text("complete"),
              onPressed: () {},
            )
          ],
        );
      }
      return Container();
    }
```

Funkcja `_itemFooter()` zwraca widżet w zależności od statusu przysługi.

Statusy przysług są definiowane przez `getters` w klasie `Favor`:

- W fazie **żądania** (przysługa nie została jeszcze zaakceptowana lub odrzucona) zwracamy widżet `Row` z dwoma instancjami `FlatButton` z odpowiednimi dostępnymi akcjami: odmów lub zrób. `FlatButton` to przycisk Material Design, który nie ma ani elewacji, ani koloru tła.
- W fazie **wykonywania** zwracamy widżet Row z odrzuconymi lub zakończonymi akcjami jako `FlatButtons`.
- Aby uzyskać stan **ukończenia**, wyświetlamy datę i godzinę zakończenia sformatowaną za pomocą klasy `DateFormat` z Darta wewnątrz widżetu `Chip`, aby odróżnić ją od reszty tekstu.
- W statusie **odmowy** zwracamy widżet `Container` bez ograniczeń rozmiaru; to jest pusty kontener (nie zajmuje miejsca na layoucie).

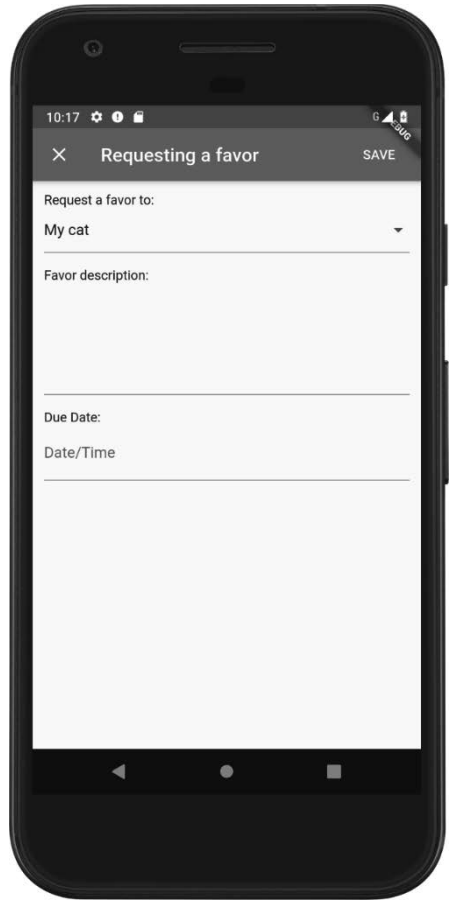
> Zawsze możesz skorzystać z metod klasy pomocnika `EdgeInsets`, gdy definiujesz padding lub marginesy. Ma on przydatne do tego metody. Sprawdź oficjalną stronę dokumentacji: *https://api.flutter.dev/flutter/painting/EdgeInsets-class.html*.

Jak widzieliśmy w implementacji list przysług, istnieją różne widżety tworzące layout. Zwróć jednak uwagę, że nie obsługujemy tutaj żadnych działań użytkownika; tym wszystkim zajmiemy się w następnym rozdziale. Rzućmy okiem na ekran prośby o przysługę.

Zwróć uwagę na właściwość `onPressed` dla `FlatButton`; definiuje akcję, gdy użytkownik ją wybierze. Przyjrzymy się temu w rozdziale 5., więc kontynuuj lekturę!

Ekran prośby o przysługę

Ekran prośby o przysługę będzie miejscem, w którym nastąpi interakcja użytkownika z aplikacją. Na razie przyjrzymy się tylko układowi tego ekranu. W miarę upływu czasu będziemy łączyć elementy, aby wybrać znajomego, który poprosi o przysługę, a także zapisać przysługę w zewnętrznej bazie danych Firebase:



Widżet ekranu prośby o przysługę również ma widżet Material Design `Scaffold` z paskiem aplikacji, który tym razem zawiera akcje. Treść widżetu `Scaffold` zawiera pola, które przyjmą informacje wejściowe od użytkownika w celu utworzenia prośby o przysługę.

Kod layoutu

Widżet RequestFavorPage również jest teraz bezstanowy, ponieważ obecnie zależy nam tylko na jego layoucie:

```
class RequestFavorPage extends statelessWidget {
  final List<Friend> friends;

  RequestFavorPage({Key key, this.friends}) : super(key: key);

  @override
  Widget build(BuildContext context) {...} // dla zwięzłości
}
```

Jak widać, jedyną rzeczą w opisie widżetu jest lista znajomych, która musi być dostarczona przez widżet nadrzędny, ponieważ jest to obecnie bezstanowa (statelessWidget) instancja widżetu.

> Aby dowiedzieć się, jak poruszać się między ekranami (czyli od listy przysług do ekranu **Poproś o przysługę**), przejdź do rozdziału 7., w którym mówimy o wyznaczaniu ścieżek i nawigacji.

Metoda build() widżetu zaczyna się w następujący sposób:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("Requesting a favor"),
      leading: CloseButton(),
      actions: <Widget>[
        FlatButton(
            child: Text("SAVE"), textColor: Colors.white, onPressed: ()
            {}),
      ],
    ),
  body: ... // kontynuacja poniżej
...
```

appBar zawiera tutaj dwie nowe właściwości:

- leading, czyli widżet wyświetlany przed tytułem. W tym przypadku używamy widżetu CloseButton, który jest przyciskiem zintegrowanym z widżetem Navigator (więcej o tym w rozdziale 7.).
- actions, która otrzymuje listę widżetów do wyświetlenia po tytule; w tym przypadku wyświetlamy instancję FlatButton, za pomocą której zapiszemy prośbę o przysługę.

body Scaffolda definiuje układ w widżecie Column. Zawiera dwie nowe właściwości: pierwsza to mainAxisSize, która definiuje rozmiar na osi pionowej; tutaj używamy MainAxisSize.min, więc zajmuje ona tylko tyle miejsca, ile jest konieczne. Druga to crossAxisAlignment, które definiuje wyrównanie elementów podrzędnych na osi poziomej. Domyślnie Column wyrównuje

swoje elementy podrzędne poziomo do środka. Korzystając z tej właściwości, możemy zmienić to zachowanie. W `Column` znajdują się trzy widżety podrzędne, które przyjmą dane wejściowe użytkownika:

- Widżet `DropdownButtonFormField`, który po wybraniu wyświetla elementy widżetu `DropdownMenuItem` w wyskakującym okienku:

```
...
DropdownButtonFormField(
  items: friends
      .map(
        (f) => DropdownMenuItem(
              child: Text(f.name),
            ),
      )
      .toList(),
),
...
```

Tutaj używamy metody `map()` z typu Darta `Iterable`, gdzie każdy element z listy (w tym przypadku przyjaciele) jest mapowany na nowy widżet `DropdownMenuItem`. Tak więc każdy element z listy znajomych zostanie wyświetlony jako element widżetu na liście rozwijanej.

- Widżet `TextFormField`, który umożliwia wprowadzanie tekstu za pomocą klawiatury:

```
TextFormField(
  maxLines: 5,
  inputFormatters: [LengthLimitingTextInputFormatter(200)],
),
```

Widżet `TextFormField` umożliwia wprowadzanie tekstu. Dodając do niego `inputFormatters`, możemy skonfigurować jego wygląd na ekranie. Tutaj po prostu ograniczamy całkowitą długość wpisywanego tekstu do 200 znaków za pomocą klasy `LengthLimitingTextInputFormatter`, która jest udostępniana przez bibliotekę `flutter/services`.

> Sprawdź wszystkie dostarczone narzędzia z pakietu `flutter/services` na stronie pakietu: *https://api.flutter.dev/flutter/services/services-library.html*.

- Widżet `DateTimePickerFormField`, który umożliwia użytkownikowi wybranie instancji `DateTime` i mapowanie go na typ `DateTime` Darta:

```
DateTimePickerFormField(
  inputType: InputType.both,
  format: DateFormat("EEEE, MMMM d, yyyy 'at' h:mma"),
  editable: false,
  decoration: InputDecoration(
    labelText: 'Date/Time', hasFloatingPlaceholder: false),
  onChanged: (dt) {},
),
```

Widżet `DateTimePickerFormField` nie jest wbudowanym widżetem Fluttera. To jest wtyczka innej firmy z biblioteki `datetime_picker_formfield`. Definiujemy niektóre właściwości, aby zmienić jej wygląd:

- `inputType` — czy wybrać datę, godzinę lub obie wartości.
- `format` — typ `DateFormat` Darta definiujący format reprezentacji ciągu znaków.
- `editable` — czy widżet ma być ręcznie edytowany przez użytkownika.
- `decoration` — służy do definiowania dekoracji dla pola wejściowego (za pomocą Material Design). Zauważ, że nie zdefiniowaliśmy go dla innych pól wejściowych.
- `onChanged` — wywołanie zwrotne z nową wartością wybraną przez użytkownika.

> Aby dowiedzieć się o wszystkich dostępnych opcjach i sposobie korzystania z widżetu `DateTimePickerFormField`, odwiedź stronę *https://pub.dartlang.org/packages/datetime_picker_formfield*.

Oprócz pól wejściowych w `Column` znajdują się również widżety `Container` i `Text`, które pomagają w formatowaniu i projektowaniu ekranu. Spójrz na kod źródłowy rozdziału, aby uzyskać pełny kod layoutu.

Tworzenie niestandardowych widżetów

Podczas tworzenia interfejsów użytkownika za pomocą Fluttera zawsze będziemy musieli tworzyć niestandardowe widżety; nie możemy i nie chcemy od tego uciekać. W końcu Flutter tak dobrze umożliwia kompozycję widżetów do tworzenia unikalnych interfejsów.

W aplikacji utworzyliśmy już część layotu, a jedynymi niestandardowymi widżetami, które stworzyliśmy, są widżety `FavorsPage` i `RequestFavorPage`.

Być może zauważyłeś również, że ze względu na sposób tworzenia layoutów we Flutterze kod może stać się ogromny i trudny do utrzymania. Aby rozwiązać ten problem, stworzyliśmy małe metody, które dzielą tworzenie widżetu na części w celu zbudowania pełnego layoutu.

Dzielenie widżetów na małe metody pomaga zmniejszyć rozmiar kodu, ale nie jest tak dobre dla Fluttera. W naszym przypadku nie mamy jeszcze złożonego layoutu, więc jest to w porządku, ale w przypadku złożonego layotu, w którym drzewo widżetów może zmieniać się wiele razy, posiadanie widżetów jako wbudowanych metod nie pomoże frameworkowi w optymalizacji procesu renderowania.

Aby pomóc platformie w optymalizacji procesu renderowania, powinniśmy zamiast tego podzielić nasze metody na małe, celowe widżety. Tak więc operacje na **drzewie widżetów | drzewie elementów** zostaną zoptymalizowane. Pamiętaj, że rodzaj widżetu pomaga platformie wiedzieć, kiedy widżet się zmienia i należy go przebudować, co wpływa na cały proces renderowania.

Wróćmy więc do naszego widżetu `FavorsPage` i przekonwertujmy metody małych widżetów na nowe, niestandardowe, małe widżety.

Metodę `_favorsList()` (zobacz załączony kod źródłowy) można refaktoryzować do nowego widżetu `FavorsList`. Następnie właściwość `itemBuilder` widżetu `FavorsList` może zostać zmieniona w celu zwrócenia widżetu `FavorCardItem`, który zwraca element karty:

```
class FavorCardItem extends statelessWidget {
  final Favor favor;

  const FavorCardItem({Key key, this.favor}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Card(
      key: ValueKey(favor.uuid),
      margin: EdgeInsets.symmetric(vertical: 10.0, horizontal: 25.0),
      child: Padding(
        child: Column(
          children: <Widget>[
            _itemHeader(favor),
            Text(favor.description),
            _itemFooter(favor)
          ],
        ),
        padding: EdgeInsets.all(8.0),
      ),
    );
  }
  Widget _itemHeader(Favor favor) { ... } // dla zwięzłości
  Widget _itemFooter(Favor favor) { ... } // dla zwięzłości
}
```

Jedyną rzeczą, która się zmienia, jest dodanie nowej klasy z odpowiednimi polami typu `final`, które mają znaczenie dla renderowania widżetu; metoda `build()` jest prawie taka sama jak poprzednia metoda `_buildFavorsList()`.

Zwróć uwagę, że element karty przysługi nadal zawiera części nagłówka i stopki jako metody, odpowiednio `_itemHeader()` i `_itemActions()`. Dzięki temu są one wystarczająco małe, aby nie szkodzić procesowi renderowania. Ale pamiętaj, że podzielenie ich na widżety też nie zaszkodzi.

Dzięki technice korzystania z podzielonych widżetów przekażemy platformie wystarczającą ilość informacji o naszych widżetach, które będą zachowywać się jak widżety wbudowane i będą mogły być optymalizowane jak widżety wbudowane.

> Polecam przeczytanie interesującego posta na blogu na temat wydajności widżetów: *https://iiro.dev/splitting-widgets-to-methods-performance-antipattern/*.

Podsumowanie

W tym rozdziale widzieliśmy każdy z dostępnych typów widżetów Fluttera oraz ich różnice. Widżety `stateless` nie są często odbudowywane przez framework; z drugiej strony, widżety `stateful` są odbudowywane za każdym razem, gdy zmienia się skojarzony z nim obiekt `State` (co może mieć miejsce, na przykład, gdy używana jest funkcja `setState()`). Widzieliśmy również, że Flutter zawiera wiele widżetów, które można łączyć w celu tworzenia unikalnych interfejsów użytkownika, i że nie muszą one być elementami wizualnymi na ekranie użytkownika; mogą to być layouty, style, a nawet widżety danych, takie jak `InheritedWidget`. Rozpoczęliśmy tworzenie małej aplikacji, którą będziemy rozwijać w następnych kilku rozdziałach; będziemy dodawać do niej określone funkcje, przedstawiając nowe ważne koncepcje dotyczące Fluttera.

W następnym rozdziale dowiemy się, jak dodać interakcję użytkownika do aplikacji w wyniku reakcji na dotknięcia użytkownika i wprowadzane dane, które później będą przechowywane w bazie Firebase.

Flutter od Google.
Specjalnie dla profesjonalnych twórców najlepszych aplikacji!

Flutter to opracowany przez Google framework do tworzenia aplikacji mobilnych, które świetnie wyglądają i dobrze działają w systemach Android i iOS. Pisanie reaktywnych aplikacji we Flutterze jest przyjemną i efektywną pracą. Język Dart został bowiem zbudowany tak, by ułatwić pisanie poprawnego kodu. Flutter jest intensywnie rozwijany przez Google: framework co kilka tygodni zyskuje nowe funkcjonalności. Mimo że to relatywnie nowe rozwiązanie, cieszy się dużą popularnością. Liczba poważnych aplikacji napisanych we Flutterze stale rośnie.

Dzięki temu przewodnikowi płynnie rozpoczniesz pisanie aplikacji we Flutterze w języku Dart. Dowiesz się, jak skonfigurować środowisko programistyczne i rozpocząć projekt. Książka poprowadzi Cię przez proces projektowania interfejsu użytkownika i funkcji umożliwiających poprawną pracę aplikacji. Nauczysz się pisać własne wtyczki (tzw. plug-iny). Poznasz techniki poprawy wrażeń użytkownika i dowiesz się, jak tworzyć dobre, intuicyjne interfejsy. Dzięki licznym wyjaśnieniom, przykładom i wskazówkom nauczysz się pisać aplikacje wolne od błędów i gotowe do wdrożenia w App Store i Google Play. W efekcie dobrze przygotujesz się do projektowania aplikacji we Flutterze na wysokim, profesjonalnym poziomie.

W książce:

- podstawy języka programowania Dart i koncepcje interfejsu Fluttera
- pisanie wtyczek i widżetów Fluttera
- stylizacja aplikacji i poprawa wrażeń użytkownika we Flutterze
- stosowanie komponentu AnimatedBuilder
- uzyskiwanie natywnej wydajności aplikacji

Alessandro Biessek pochodzi z Brazylii. Jest twórcą aplikacji mobilnych do systemów Android i iOS. Programuje w różnych językach: w Delphi, PHP, Node.js, Golang, pracuje też z Apache Flex, a także Javą i Kotlinem. Jest pasjonatem nowych technologii. Od dłuższego czasu fascynuje się frameworkiem Flutter.

ISBN 978-83-283-7825-4
9 788328 378254
Cena: 89,00 zł

Packt