Krzysztof Baranowski

Flutter

Podstawy

Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

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

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

ISBN: 978-83-8322-644-6

Printed in Poland.

Spis treści

Rozdział 4.
Aplikacja Demo

Zaczynamy naprawdę grubo, bo już za chwilę będziesz wykorzystywać Fluttera do tworzenia swojej pierwszej aplikacji w tym frameworku. Jako że tworzenie w nim aplikacji jest szybkie, to i my nie będziemy zostawać w tyle — w tyle zostawimy teorię na rzecz praktyki. Przynajmniej do momentu, kiedy będzie potrzebna. Do dzieła!

W wybranym przez siebie IDE[1] stwórz nowy projekt po wybraniu szablonu pustej aplikacji Fluttera. W tym momencie nie ma znaczenia, jakie opcje wybierzesz w kreatorze. Po prostu użyj domyślnych ustawień (upewnij się jedynie, że platformy iOS i/lub[2] Android zostały zaznaczone w kreatorze nowego projektu) i wpisz jakąś ciekawą nazwę projektu, np. **My First Million Mobile App**. Nie ma co się ograniczać, mierzymy wysoko! Ważna dygresja: w nazwie projektu musisz zastosować **snake case**. Czyli nadaj mu nazwę *my_first_million_mobile_app*. Kliknij *Dalej* i poczekaj, aż środowisko skończy inicjalizację zasobów (rysunek 4.1).

Twoim oczom powinien ukazać się kawałek szablonowego kodu w pliku *main.dart*. Domyślną zawartością nowego projektu jest prosta aplikacja, która pozwala zaktualizować licznik za pomocą przycisku[3]. Uruchom ją, aby się upewnić, że wszystko zostało skonfigurowane poprawnie. Jeśli masz problemy z jej uruchomieniem na rzeczywistym urządzeniu, upewnij się, czy włączyłeś opcje programistyczne oraz debugowanie USB na swoim urządzeniu. Przypominam, że możesz to zrobić na podstawie instrukcji z części I książki, gdzie został opisany proces konfiguracji urządzenia, emulatora (Android) oraz symulatora (iOS). Przejdź do rozdziału 8. „Mechanizm Hot Restart oraz Reload" w części III książki, aby dowiedzieć się o funkcjonalności „Hot Restart" oraz „Hot Reload", które pozwolą Ci na szybsze obserwowanie zmian wprowadzonych w kodzie aplikacji.

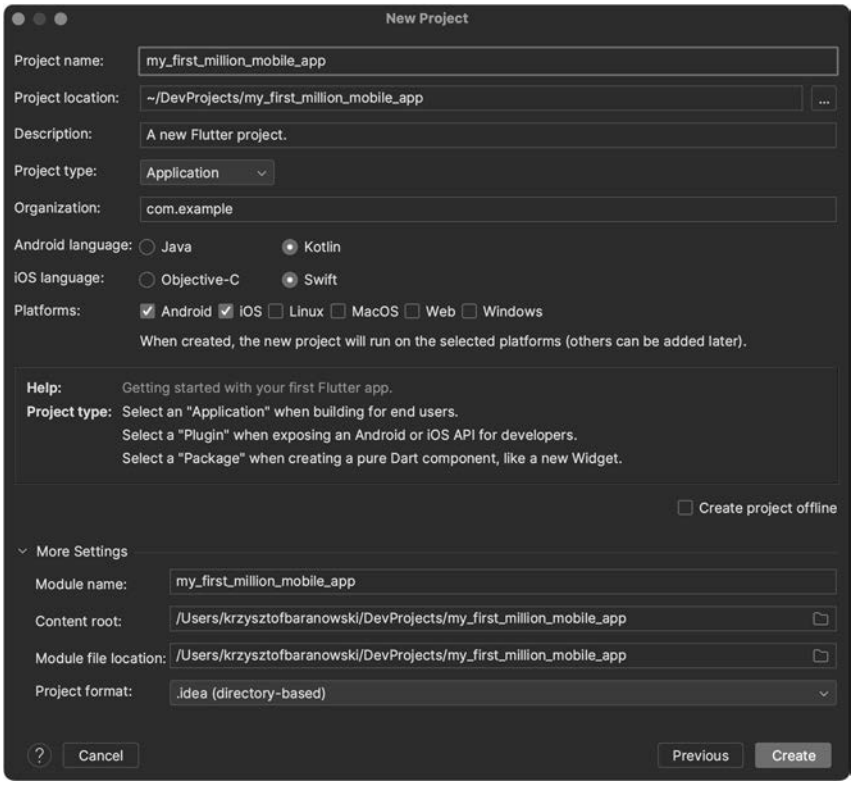
To, co we Flutterze jest najlepsze, to fakt, że stosunkowo niewielka ilość kodu pozwala przygotować solidną aplikację. Już wkrótce przekonasz się o tym na własnej skórze.

---

[1] IDE (ang. *Integrated Development Environment*). Oznacza po prostu wybrane przez Ciebie środowisko pracy, np. Android Studio czy IntelliJ.

[2] Jeśli pracujesz w systemie Windows, zaznacz tylko system Android. W przypadku macOS — oba.

[3] Ten przycisk nosi nazwę *floating action buton*, w skrócie FAB. Jest to jeden z komponentów systemy Material Design. Ciekawostką jest to, że Flutter ma specjalne kontrolki, które pozwalają zbudować aplikację opartą na tym systemie.



RYSUNEK 4.1. Kreator tworzenia nowego projektu w IntelliJ IDEA

Następnym krokiem będzie usunięcie kodu, który znajduje się w pliku *main.dart*. Usuń wszystko poza funkcją `main` oraz jej zawartością. Nie przejmuj się tym, że pojawią się błędy. Wkrótce się nimi zajmiemy. Za chwilę krok po kroku dodamy szczyptę magii i napiszesz swój pierwszy kod w języku Dart, wykorzystujący SDK[4] Fluttera.

Zanim dotkniemy kodu...

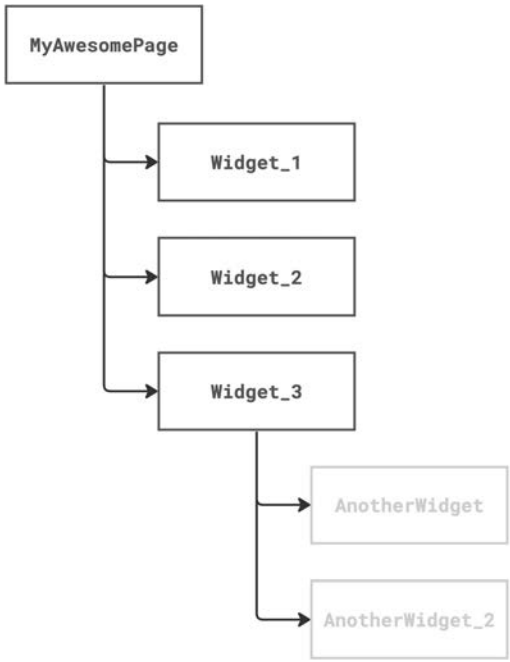
W wielu językach programowania funkcja `main` to punkt startowy aplikacji. Nie inaczej jest tutaj. Wewnątrz niej wywoływana jest funkcja `runApp`, która pozwala uruchomić aplikację Flutter. Wystarczy, że przekażemy do niej obiekt typu `Widget`, aby po uruchomieniu programu naszym oczom ukazała się aplikacja. Więcej o tym typie dowiesz się wkrótce, ale teraz zapamiętaj, że **widżet jest budulcem i zarazem kamieniem węgielnym każdego elementu interfejsu użytkownika**[5], który będziemy tworzyć.

[4] SDK (ang. *Software Development Kit*) — zestaw narzędzi niezbędny do tworzenia aplikacji korzystających z danej biblioteki.

[5] Od teraz będę używał akronimu UI (ang. *User Interface*), kiedy będę mówił o interfejsie użytkownika.

Każdy element UI jest widżetem. Prawie każdy widżet może zawierać inny widżet. Budowanie UI we Flutterze polega na umieszczaniu jednego widżetu w drugim. Dzięki takiej funkcji, w zasadzie prawie bez żadnych ograniczeń, Flutter pozwala budować niesamowicie zaawansowane widoki przy niskim stopniu skomplikowania dodawanego kodu. Na rysunku 4.2 znajdziesz wizualne przedstawienie działania widżetów.

RYSUNEK 4.2.
Drzewo widżetów



Metoda tworzenia UI we Flutterze nosi nazwę **programowania deklaratywnego**, co jest przeciwnością programowania imperatywnego. O różnicy między nimi powiemy sobie później.

Tymczasem wróćmy do tworzenia widoków. Zarówno iOS, jak i Android umożliwiają ich tworzenie o dowolnej złożoności. Flutter kwestię tworzenia zaawansowanych widoków wynosi na wyższy poziom. Jest to zdecydowanie łatwiejsze niż przy podejściu natywnym[6] i w wielu przypadkach wymaga mniejszych nakładów pracy.

W ostatnim czasie, idąc śladami Fluttera[7], Google wprowadziło Android Compose, a Apple SwiftUI. Oba te rozwiązania umożliwiają tworzenie widoków na wzór tego dostępnego we Flutterze. Trzeba szczerze przyznać, że to właśnie Flutter przetarł szlaki i pokazał, że budowanie widoków w ten sposób jest znacznie korzystniejsze i daje większe możliwości. Za jakiś czas przedstawię bardziej szczegółowo

[6] Podejście natywne oznacza bezpośrednie użycie rozwiązań z domysłu przygotowanych pod daną platformę. W przypadku Androida będzie to użycie języka Java/Kotlin oraz Android SDK. Jeśli mowa o iOS będzie to Swift i iOS SDK.

[7] To moja własna obserwacja.

i technicznie, czym różni się budowanie widoków we Flutterze od tego, do którego programiści aplikacji mobilnych byli przyzwyczajeni, zanim pojawił się on na rynku.

Plik pubspec.yaml

W kilku żołnierskich słowach przedstawię Ci plik konfiguracyjny *pubspec.yaml*. Znajdziesz go w głównym folderze projektu. Będziesz z niego korzystał stosunkowo często, dlatego warto znać podstawowe kwestie z nim związane[8]. Jego zadanie to przechowywanie podstawowej konfiguracji projektu Flutter. Nie chcę zanudzać Cię opisem każdej możliwej sekcji już teraz. Poszczególne sekcje tego pliku będziesz poznawał w kolejnych rozdziałach. Sam plik jest łatwy do zrozumienia i jestem pewny, że jeżeli spróbujesz zapoznać się z nim samodzielnie, to będziesz w stanie zrozumieć, co tam się dzieje.

Znajdź i otwórz plik *pubspec.yaml*. W tym momencie najważniejsza jest dla nas sekcja `dependencies`. To tutaj będziemy dodawać zewnętrzne biblioteki z repozytorium `pub.dev`. To, co jest niesamowite, to fakt, jak dużo gotowych rozwiązań zostało przygotowanych i wciąż jest rozwijanych przez społeczność Fluttera — począwszy od nowych, zaawansowanych widżetów przez frameworki[9], a skończywszy na operowaniu na obrazach.

Aby uprościć Ci zadanie, stworzyłem bibliotekę, która ułatwi Ci pracę nad projektami w tej książce. Zawarłem w niej m.in. gotowe palety kolorów, różne stałe, np. wartości marginesów i kilka pomocniczych klas. Zaktualizuj sekcję `dependencies` zgodnie z listingiem poniżej.

```
dependencies:
  flutter:
    sdk: flutter

  #dodaj poniższą linjkę
  flutter_podstawy_utilities: 1.0.0
```

Aby zmiany zostały przetworzone, musimy odświeżyć konfigurację. W tym celu w terminalu (w wybranym przez Ciebie środowisku) wprowadź poniższe polecenie[10]. Pamiętaj, aby przejść do głównego katalogu aplikacji.

```
flutter pub get
```

Po krótkiej chwili biblioteka będzie dostępna do użytku w naszym projekcie. Inny sposób to wykorzystanie pluginu Fluttera, który jest zainstalowany w Twoim IDE. W przypadku IntelliJ IDEA oraz Android Studio wystarczy, że na górnej belce klikniesz przycisk *Pub get*. W Visual Code będzie to jeszcze szybsze, bo środowisko zaktualizuje konfigurację od razu po zapisie pliku *pubspec*.

---

[8] Więcej o *pubspec.yaml* możesz znaleźć w części III książki.

[9] Framework — spójny pakiet rozwiązań, który definiuje szkielet tworzenia aplikacji.

[10] Listę przydatnych komend wraz z ich opisami znajdziesz w części III książki.

Pierwszy kontener

Usunęliśmy trochę kodu, więc mamy problem z kompilacją projektu. Naprawmy to i stwórzmy klasę `MyApp` dziedziczącą po typie `StatelessWidget`. Zrób to poniżej metody `main`. Kompilator powinien Cię powiadomić o brakującej implementacji metody `build()`. Dodaj ją. To, co zostanie zwrócone jako rezultat tej metody, pokaże się na ekranie. Na początek zróbmy coś łatwego i użyjmy widżetu `Container` z kolorem `melon` z biblioteki pomocniczej `flutter_podstawy_utilities`. Jest to najprostszy widżet, który pozwala ustawić kolor tła, a przecież chcemy zobaczyć już na ekranie jakieś fajerwerki. Domyślnie `Container` zajmuje całą dostępną przestrzeń.

```
import 'package:flutter/material.dart';
import 'package:flutter_podstawy_utilities/colors/my_colors.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      color: MyColors.melon,
    );
  }
}
```

Pamiętaj, aby dodać wymagany import.

```
import 'package:flutter_podstawy_utilities/colors/my_colors.dart';
```

Po uruchomieniu aplikacji[11] zobaczysz jednokolorowe tło. Zwróć uwagę, z jak małej ilości kodu składa się Twoja aplikacja. Te kilka linijek sprawiło, że masz już ekstra-aplikację z melonowym tłem! OK, racja, nie ma tu nic spektakularnego… Ale gdyby tak móc zmieniać kolor tego tła po każdym kliknięciu ekranu? Użyjmy zatem widżetu `GestureDetector`. Umożliwi on nam dodanie logiki, która zostanie wykonana jako odpowiedź na jakiś gest.

A więc chcemy zmienić kolor kontenera po jego kliknięciu. Aby tak się stało, `Gesture↪Detector` musi stać się rodzicem widżetu `Container`. To oznacza, że obszarem roboczym, który weźmie udział w przechwyceniu gestu, będzie właśnie obszar kontenera. Widżet `GestureDetector` umożliwia reagowanie na wiele gestów. My wykorzystamy najprostszy z nich, czyli zwykłe dotknięcie ekranu. Wewnątrz tego widżetu została zaimplementowana logika, która wywołuje odpowiednie metody (`onTap()`, `onTapCancel()`, `onLongPress()` itd.) w zależności od rodzaju gestu wykonanego przez użytkownika. Aby to zrobić, musimy przekazać delegat (ang. *delegate*), który zostanie wywołany po tym zdarzeniu. Dla jasności: delegatem nazywamy funkcję lub metodę, którą przekazujemy jako argument innej metody lub funkcji. Twoim zadaniem jako programisty jest przekazanie odpowiedniej funkcji w wybranym przez ciebie zdarzeniu.

[11] Jak uruchomić aplikację, opisałem w rozdziale 10.

Po każdym kliknięciu będziemy zmieniać kolor tła. W tym celu stwórzmy zmienną typu `Color`, a w delegacie przypisanym do zdarzenia `onTap` użyjemy warunku logicznego do zmiany koloru.

```
class MyApp extends StatelessWidget {
  Color containerColor = MyColors.melon;

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: () {
        debugPrint("onTap called!!");
        containerColor = containerColor == MyColors.melon
            ? MyColors.ufoGreen
            : MyColors.melon;
      },
      child: Container(
        color: containerColor,
      ),
    );
  }
}
```

Szybsze dodawanie widżetów

Możesz przyspieszyć dodawanie widżetów poprzez użycie skrótów klawiszowych. Plugin Fluttera zainstalowany w twoim IDE ma możliwość manipulowania hierarchią widżetów. Zamiast ręcznie wpływać na kształt kodu umieść kursor na wybranym przez siebie elemencie, np. *Container*, użyj skrótu klawiszowego *option + ENTER* (macOS) lub *alt + ENTER* (Windows), aby skorzystać z menu podręcznego, które pozwoli Ci np. na dodanie nowego widżetu jako rodzica (dostępnych jest kilka predefiniowanych), usunięcie obecnego czy zamienienie miejscami dwóch sąsiadujących ze sobą widżetów.

Niestety nasz kod nie zadziała zgodnie z oczekiwaniami. O ile w konsoli zobaczysz informację o kliknięciu, o tyle z kolorem tła już gorzej. Logika wydaje się poprawna, ale sprawa jest nieco bardziej skomplikowana.

Flutter ma szereg optymalizacji. Jedną z nich jest podział widżetów na dwa typy. Na takie, które przechowują stan, oraz takie, które tego nie robią. Nie chcę zamęczać Cię teraz szczegółowymi różnicami pomiędzy nimi. Tym zajmiemy się później. W tej chwili zapamiętaj tylko, że `StatelessWidget`, którego używasz w swoim widżecie (mam na myśli `MyApp` i jej klasę bazową), nie może samodzielnie odświeżyć się na ekranie. Może to zrobić jakiś inny, nadrzędny widżet, którego jeszcze nie mamy. Widżet typu `StatelessWidget` tworzy się raz, metoda `build()` wywoływana jest jednokrotnie i jego stan wewnętrznie nie ulega zmianie. Natomiast typ `StatefulWidget` jest dynamiczny. Może samodzielnie zaktualizować swój stan. Może reagować na zdarzenia wyzwolone przez interakcję użytkownika lub kiedy otrzyma jakieś dane. To właśnie tego drugiego potrzebujemy, aby móc odświeżyć kolor kontenera.

Implementacja `StatefulWidget` jest trochę bardziej skomplikowana niż obecnego `StatelessWidget`, ale to nic trudnego. Aby stworzyć widżet oparty na tym typie, potrzebujesz dwóch klas. Zaktualizuj swój kod na podstawie poniższego listingu.

```
class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  Color containerColor = MyColors.melon;

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: () {
        debugPrint("onTap called!!");
        containerColor = containerColor == MyColors.melon
            ? MyColors.ufoGreen
            : MyColors.melon;
      },
      child: Container(
        color: containerColor,
      ),
    );
  }
}
```

Nie przejmuj się, jeśli jeszcze nie rozumiesz, co tutaj się dzieje. Dobrze, mamy wspomniane już użycie `StatefulWidget`. Brakuje nam ostatniego elementu w tej układance. Jest nim żądanie odświeżenia widoku. **Musimy wskazać środowisku, że stan naszego widżetu uległ zmianie.** W naszym przypadku wiemy, że dzieje się to wtedy, gdy stukamy w ekran i przypisujemy nowy kolor do pola `_containerColor`. Aby poinformować środowisko, że stan się zmienił, musimy wywołać metodę `setState()`, która jest dostępna w klasie `State`, czyli obecnej klasie bazowej. Musimy przenieść kod przypisania koloru do wnętrza metody `setState()`. Zadanie jest prozaiczne — ta metoda przyjmuje jako swój argument delegat. Oznacza to, że możemy w niej umieścić kod zmiany koloru. To dzięki niej Flutter wie, że należy przebudować drzewo widżetów.

```
debugPrint("onTap called!!");
setState(() {
  containerColor = containerColor == MyColors.melon
      ? MyColors.ufoGreen
      : MyColors.melon;
});
```

Metoda `setState()`

Jej zadaniem jest poinformowanie środowiska, że wewnętrzny stan widżetu został zmieniony. Oznacza to, że musisz ją wywołać zawsze wtedy, gdy wymagana jest zmiana reprezentacji wizualnej twojego widżetu, np. gdy chcesz zmienić wartość koloru czy tekstu w trakcie działania aplikacji.

Jako swój argument przyjmuje delegat, w którym powinna znaleźć się logika mająca wpływ na zmianę stanu, np. aktualizacja jakiegoś pola, którego wartość jest wykorzystywana w widżecie.

Uruchom teraz aplikację. Po dotknięciu ekranu kolor tła powinien już zmieniać się poprawnie. Sprawdź inne możliwości i w ramach praktyki spróbuj zaimplementować zmianę koloru np. przez podwójne stuknięcie w ekran. Odpowiedni parametr znajdziesz w widżecie `GestureDetector`.

Animacja zmiany koloru

Jak myślisz, ile nowego kodu musimy dodać, aby zmiana koloru była płynnie animowana? Odpowiedź brzmi: dwie linijki. Po pierwsze, zamień widżet `Container` na `AnimatedContainer`. Po drugie, ta zmiana wymusi uzupełnienie wymaganego argumentu `duration`. Oznacza on długość animacji. Stwórz i przypisz nowy obiekt `Duration` oraz ustaw czas trwania np. na 700 milisekund. Uruchom aplikację i sprawdź efekt.

```
AnimatedContainer(
  duration: Duration(milliseconds: 700),
  color: containerColor,
)
```

Świetne, prawda? A dopiero się rozkręcamy.

Układy widoków, czyli layouty

Z pomocą tylko kontenera za wiele nie zdziałamy. Musimy mieć możliwość dodawania wielu elementów w jakiejś zdefiniowanej strukturze. A mowa tu o układach (ang. *layouts*). Pierwszy przykład to `Row` oraz `Column`. Stwórz nowy plik w folderze *lib* projektu i nazwij go `my_animated_container.dart`[12]. Utwórz tam widżet `MyAnimatedContainer` typu `StatefulWidget` i przenieś cały kod odpowiedzialny za zbudowanie naszego animowanego kontenera z klasy `MyApp`. Nie będę już przedstawiał gotowego kodu na listingu. W zasadzie będzie to kopia kodu, który już dodaliśmy. W klasie `MyApp` zwróć nasz nowy widżet.

```
import 'package:flutter/material.dart';
import 'package:flutter_podstawy_utilities/colors/my_colors.dart';
import 'my_animated_container.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => _MyAppState();
}
```

[12] Przyjęło się, aby pliki języka Dart miały nazwy oparte na tzw. **snake case**, tzn.: *my_file.dart*, *my_second_great_class.dart*. Używamy tylko małych liter, a odstępy pomiędzy wyrazami zastępujemy znakiem podkreślenia.

```
class _MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MyAnimatedContainer();
  }
}
```

Działania, które podjąłeś, pozwolą Ci zachować porządek w projekcie. Jeśli pojawią się błędy kompilacji, to upewnij się, czy dodałeś wymagane wywołania `import` u góry pliku.

W klasie `MyAnimatedContainer` przypisz parametrom `width` oraz `height` wartości `50` w widżecie `AnimatedContainer`. Ułatwi Ci to pracę za chwilę.

Zajmijmy się teraz stworzeniem kilku animowanych kontenerów ułożonych na ekranie obok siebie. Wróć do pliku *main.dart* i wprowadź kilka zmian w klasie `MyApp`.

```
class _MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: Container(
          color: MyColors.onahau,
          child: _buildContent(),
        ),
      ),
    );
  }

  Widget _buildContent() {
    return Column(
      children: [
        _buildTitle(),
        _buildFirstRow(),
      ],
    );
  }

  Widget _buildTitle() {
    return Container();
  }

  Widget _buildFirstRow() {
    return Row(
      children: [
        MyAnimatedContainer(),
        Padding(
          padding: const EdgeInsets.all(Dimens.normalMargin),
          child: MyAnimatedContainer(),
        ),
        MyAnimatedContainer()
      ],
    );
  }
}
```

Pamiętaj o dodaniu odpowiedniego importu dla stałych `Dimens`.

```
import 'package:flutter_podstawy_utilities/constants/dimens.dart';
```

Zarówno `Row`, jak i `Column` zawierają podstawowy argument `children`, który pozwala dodać kilka elementów do układu. Nie zapomnij o zaktualizowaniu metody `build()` — dodaj wywołanie naszej nowej metody. Jak widzisz, użyliśmy nowych widżetów. Widżet `Padding` może zostać dodany w zasadzie do dowolnego innego jako jego rodzic, aby wstawić pewien odstęp. Co więcej, będzie to jeden z najczęściej używanych przez Ciebie widżetów. `MaterialApp` oraz `Scaffold` są widżetami, które pozwalają korzystać z wielu gotowych zachowań i konfigurują najczęściej używane funkcjonalności w aplikacji Flutter. W tym momencie traktuj go jako *must-have.* Podzieliłem kod na kilka logicznych części, aby znacznie łatwiej było nam w nim nawigować. Mam na myśli to, że stworzyłem kilka metod, z których każda buduje jakiś kawałek UI. Dzielenie kodu na mniejsze metody jest we Flutterze niesamowicie ważne. W kolejnych rozdziałach wyniesiemy porządki na jeszcze wyższy poziom.

Rada warta milion dolarów: koniecznie naucz się aktywnie korzystać ze skrótu klawiszowego opisanego w poprzedniej wskazówce. Zdecydowanie poprawi to Twoją skuteczność w szybkim tworzeniu widoków.

Widżet `Padding`

Zadaniem tego widżetu jest dodanie odstępu pomiędzy jego dzieckiem a innymi elementami w hierarchii. Do ustawienia wartości odstępu służy argument `padding`, a ustawienie odpowiedniej wartości odbywa się za pomocą klasy `EdgeInsets` i jej kilku statycznych metod:

- `EdgeInsets.all(value)` — przypisanie tej samej wartości dla każdej strony.
- `EdgeInsets.only(left, top, right, bottom)` — przypisanie dowolnych wartości dla dowolnych stron, wszystkie są opcjonalne.
- `EdgeInsets.symmetric(horizontal, vertical)` — symetryczne przypisanie wartości odstępu.

Po uruchomieniu aplikacji zauważysz, że kontenery są częściowo zasłonięte przez elementy paska statusu systemu. Rozwiązaniem tego problemu jest dodanie odpowiedniego górnego marginesu. Spróbuj to zrobić samodzielnie i sprawdź, przy jakiej wartości marginesu elementy staną się w pełni widoczne. Wykorzystaj do tego informacje zawarte w sekcji poniżej. W dalszej części książki dowiesz się, jak wykonać taki odstęp poprawnie, bez używania stałych wartości. Jest to o tyle ważne, że każde urządzenie może mieć inną wysokość górnej belki. Oznacza to, że używanie stałej wartości jest równoznaczne z proszeniem się o kłopoty.

Widżet `SizedBox`

Zawiera argumenty `width` oraz `height` do określenia wielkości. Jeśli ma dziecko, wymusza jego rozmiar (są jednak przypadki, kiedy tak się nie stanie).

Najczęściej korzystam z tego widżetu, kiedy chcę dodać odstęp pomiędzy elementami, ale bez wprowadzania widżetu `Padding`.

Zarówno `Column`, jak i `Row` mają dwa takie identyczne argumenty `mainAxisAlignment` oraz `crossAxisAlignment`. Użycie ich pozwala ustalić, w jaki sposób elementy mają być układane względem siebie na głównej i przeciwnej osi. Dla obu widżetów `mainAxis↪Alignment` będzie oznaczało co innego. Dla `Row` główną osią jest oś pozioma — układamy element jeden obok drugiego. W przypadku `Column` będzie to oś pionowa — układamy elementy jeden pod drugim. Analogicznie działa argument `crossAxisAlignment`, ale dla osi przeciwnej. Poniżej znajduje się opis dostępnych wartości. Poświęć chwilę na zrozumienie działania tych dwóch argumentów oraz wszystkich dostępnych wartości. Eksperymentuj. Sprawdzaj, jak te wartości wpływają na układ dzieci w przypadku `Column` oraz `Row`. Jednak zanim zaczniesz, dodaj jeszcze jeden widżet `Row` do widżetu `Column` w metodzie `_buildContent` i dodaj do niego kilka kontenerów `MyAnimatedContainer`. Tym sposobem będzie Ci łatwiej zrozumieć ich zachowanie.

Widżety `Row` i `Column`, wyliczenie `MainAxisAlignment`

Aby wpłynąć na główną oś układu któregoś z tych widżetów, przypisz wartość do `mainAxisAlignment`. Możliwe wartości:

- `MainAxisAlignment.start` — układa dzieci względem początku głównej osi.
- `MainAxisAlignment.end` —układa dzieci względem końca głównej osi.
- `MainAxisAlignment.center` — układa dzieci względem środka głównej osi.
- `MainAxisAlignment.spaceBetween` — dodaje równą wolną przestrzeń tylko pomiędzy dziećmi.
- `MainAxisAlignment.spaceAround` — dodaje równą wolną przestrzeń pomiędzy dziećmi oraz dodatkowo połowę tej przestrzeni przed pierwszym i za ostatnim dzieckiem.
- `MainAxisAlignment.spaceEvenly` — dodaje równą wolną przestrzeń pomiędzy dziećmi oraz dodatkowo przed pierwszym i za ostatnim dzieckiem.

Widżety `Row` i `Column`, wyliczenie `CrossAxisAlignment`

Aby wpłynąć na główną oś układu któregoś z tych widżetów, przypisz wartość do `mainAxisAlignment`. Możliwe wartości:

- `CrossAxisAlignment.start` — układa dzieci względem początku przeciwnej osi.
- `CrossAxisAlignment.end` — układa dzieci względem końca przeciwnej osi.
- `CrossAxisAlignment.center` — układa dzieci względem środka przeciwnej osi.
- `CrossAxisAlignment.stretch` — wymaga od dzieci, aby wypełniły przestrzeń na przeciwnej osi.
- `CrossAxisAlignment.baseline` — umieszcza dzieci wzdłuż przeciwnej osi, tak aby ich linie bazowe były wyrównane względem siebie. Prawdopodobnie nie będziesz musiał korzystać z tej wartości.

Pokażmy trochę tekstu!

Wiesz już, jak układać elementy względem siebie. Teraz pokrótce się dowiesz, jak wyświetlić prosty tekst. Zaktualizujmy metodę `_buildTitle()`.

```
Widget _buildTitle() {
  return Text("My first million app!");
}
```

Pierwszym nienazwanym i jedynym wymaganym argumentem jest ciąg znaków. Widżet `Text` ma mnóstwo argumentów, które pozwolą Ci dostosować styl wyświetlanego tekstu do swoich potrzeb. Ich listę możesz przejrzeć poprzez kliknięcie ikony żarówki, która się pokaże, gdy ustawisz kursor na widżecie, lub poprzez skrót klawiszowy[13].

Dodajmy trochę finezji i zmieńmy styl tekstu. Przypisanie stylu odbywa się poprzez przekazanie nowego obiektu typu `TextStyle` do argumentu `style`.

```
Widget _buildTitle() {
  return Text("My first million app!",
      style: TextStyle(
        color: MyColors.darkCharcoal,
        fontWeight: FontWeight.bold,
        fontSize: 30,
      ));
}
```

Więcej czasu spędzimy z tym widżetem w dalszej części książki.

Warunkowe tworzenie widoku

Możemy zwrócić dowolny widżet jako rezultat metody `build()`, bazując np. na jakimś warunku. Dodajmy możliwość personalizacji `MyAnimatedContainer`, co pozwoli na wybór kolorów oraz kształtu.

Potrzebujemy trzech argumentów. Dla klarowności każdy z nich będzie nazwany oraz będzie miał domyślną wartość. Oznacza to, że będą one opcjonalne. Stwórz konstruktor z trzema parametrami: `firstColor` i `secondColor` typu `Color` oraz ostatni `shouldUseFancyShape` typu `bool`. Te zmienne będziemy przechowywać w klasie `MyAnimatedContainer`.

```
class MyAnimatedContainer extends StatefulWidget {
  final Color firstColor;
  final Color secondColor;
  final bool shouldUseFancyShape;
```

[13] Dla macOS to domyślnie *option + Enter*, dla Windows *alt +*

```
  const MyAnimatedContainer(
      {this.firstColor = MyColors.indigo,
      this.secondColor = MyColors.vividGamboge,
      this.shouldUseFancyShape = false});

  @override
  State<MyAnimatedContainer> createState() => MyAnimatedContainerState();
}
```

Te argumenty wykorzystamy w klasie stanu do przygotowania odpowiednich zmian. Zaktualizuj kod klasy stanu `_MyAnimatedContainer`.

```
class MyAnimatedContainerState extends State<MyAnimatedContainer> {
  late Color containerColor;

  @override
  void initState() {
    super.initState();

    containerColor = widget.firstColor;
  }

  @override
  Widget build(BuildContext context) {
    if (widget.shouldUseFancyShape) {
      return GestureDetector(
        onTap: () {
          debugPrint("onTap called!!");
          setState(() {
            containerColor = containerColor == widget.firstColor
                ? widget.secondColor
                : widget.firstColor;
          });
        },
        child: AnimatedContainer(
          // New feature
          decoration: BoxDecoration(
            color: containerColor,
            borderRadius: BorderRadius.circular(30),
          ),
          width: 50,
          height: 50,
          duration: Duration(milliseconds: 700),
        ),
      );
    }
    return GestureDetector(
      onTap: () {
        debugPrint("onTap called!!");
        setState(() {
          containerColor = containerColor == widget.firstColor
              ? widget.secondColor
              : widget.firstColor;
        });
      },
      child: AnimatedContainer(
        width: 50,
        height: 50,
```

```
        duration: Duration(milliseconds: 700),
        color: containerColor,
      ),
    );
  }
}
```

Użyłem pola `widget` w klasie stanu `_MyAnimatedContaineState`. To standardowy sposób, aby uzyskać dostęp do pól umiejscowionych w klasie widżetu. Sprawa jest prosta. Na podstawie wartości `shouldUseFancyShape` zwracam poprzednią implementację widoku lub nową, korzystającą z nowego kształtu (patrz komentarz w kodzie). Zwróć uwagę na to, że kiedy przypisujesz nową implementację dla `decoration` i chcesz zmienić kolor widżetu `Container`, musisz skorzystać z właściwości `color` w obiekcie `BoxDecoration` zamiast tej dostępnej bezpośrednio w `Container`. W innym przypadku środowisko rzuci odpowiedni wyjątek. Sprawdź to. W stworzonym kodzie spróbuj usunąć przypisanie koloru w `BoxDecoration` i dodaj je bezpośrednio w widżecie `Container`. Pamiętaj jednak o cofnięciu zmian.

Czytelny kod to rzecz święta

Nie wiem, czy się ze mną zgodzisz, ale kod naszego widżetu jest niesamowicie nieczytelny. Metoda jest długa, skomplikowana i na pierwszy rzut oka nie wiadomo, jak działa i jaki kawałek UI buduje. Co gorsza, kod wydaje się zduplikowany. Obecna implementacja jest dość naiwna, więc zróbmy z tym porządek.

Mała dygresja: tworząc nowe widoki, możesz bardzo łatwo doprowadzić do sytuacji, w której kod Twojego widoku stanie się dla Ciebie nieczytelny. To znaczy, że trudno będzie Ci z nim pracować. Będziesz potrzebował więcej czasu na wprowadzenie nowych, pożądanych zmian. Weź pod uwagę, że projektem możesz się zajmować przez wiele długich miesięcy. Kod, który dodałeś dzisiaj, może być dla Ciebie zrozumiały, ale za jakiś czas, gdy do niego wrócisz, może nie być już tak kolorowo. **Musisz dbać o styl kodu tu i teraz.** Nie ma żadnych wymówek. Nie jest to czcze gadanie. Zadbaj o to, aby wyrobić w sobie nawyk pisania czystego kodu[14]. Podczas lektury tej książki napotkasz wiele dobrych praktyk, z których staram się korzystać każdego dnia. Zachęcam Cię do ich aktywnego stosowania.

Dobrze, a więc w myśl zasady „Dziel i zwyciężaj" skupimy się na tworzeniu małych metod, które będą zwracały jeden mały i funkcjonalny fragment UI. Nie chcemy długich i skomplikowanych metod. Zamiast tego chcemy małych i konkretnych. To pozwoli nam **czytać** UI jak dobry kryminał. Jeśli zadbasz o nadawanie dobrych nazw dla metod, będziesz w stanie znacznie szybciej odnaleźć się w kodzie i zrozumieć jego cel.

[14] Polecam książkę Roberta C. Martina *Czysty kod. Podręcznik dobrego programisty*.

Widoki

Tworząc widoki, pamiętaj o korzystaniu z małych metod, które będą generowały mały i funkcjonalny fragment interfejsu użytkownika. Te małe klocki pozwolą Ci łatwiej wprowadzać zmiany i zwiększą czytelność kodu.

To umożliwi Ci czytanie kodu UI jak dobrego kryminału.

Mamy dwa możliwe wyniki działania metody `build()`. Jedyna różnica pomiędzy nimi to użyty kształt kontenera, czyli użycie lub nie obiektu `BoxDecoration`. Wygląda na to, że możemy przenieść kod, który znajduje się wewnątrz warunku, do nowej metody i sparametryzować tworzenie takiego kontenera. Pierwszym, najprostszym rozwiązaniem jest przekazanie obiektu `BoxDecoration` jako parametru tej metody, a potem użycie go jako argumentu w parametrze `decoration`. Nazwijmy ją `_buildBaseContainer`.

```
Widget _buildBaseContainer(BoxDecoration decoration) {
  return GestureDetector(
    onTap: () {
      setState(() {
        containerColor = containerColor == widget.firstColor
            ? widget.secondColor
            : widget.firstColor;
      });
    },
    child: AnimatedContainer(
      width: 50,
      height: 50,
      decoration: decoration,
      duration: Duration(milliseconds: 700),
    ),
  );
}
```

W głównej metodzie `build()` wystarczy, że zwrócimy rezultat świeżo dodanej metody.

```
@override
Widget build(BuildContext context) {
  BoxDecoration;

  if (widget.shouldUseFancyShape) {
    boxDecoration = BoxDecoration(
      color: containerColor,
      borderRadius: BorderRadius.circular(30),
    );
  } else {
    boxDecoration = BoxDecoration(
      color: containerColor,
    );
  }

  return _buildBaseContainer(boxDecoration);
}
```

Spójrz, o ile łatwiejszy do zrozumienia jest aktualny kod. Zmniejszyła się realna liczba linijek kodu, a tworzenie kontenera przenieśliśmy do odpowiedniej metody. Teraz będzie nam znacznie łatwiej pracować. Nic nie stoi na przeszkodzie, aby dodać do niej kolejny parametr i wpłynąć na np. szerokość kontenera. Podejmiesz wyzwanie? ☺

Sposobów na refaktoryzację poprzedniego kodu jest zapewne jeszcze kilka. Być może jesteś w stanie zaproponować jakiś inny.

W ten sposób będziemy tworzyć bardziej skomplikowane widoki. Warunki możemy umieszczać na poziomie zwracanego widoku w metodzie `build()` (lub innych mniejszych metodach, które zwracają fragment UI) i wewnątrz parametrów jakiegoś obiektu. W naszym przykładzie moglibyśmy np. sterować szerokością kontenera, bazując na jakiejś fladze[15].

```
child: AnimatedContainer(
  width: shouldHaveCustomHeight ? 100 : 50,
  height: 50,
  decoration: decoration,
  duration: Duration(milliseconds: 700),
),
```

Przyciski

Pytasz: a gdzie prawdziwe przyciski? A ja mówię: za 5 minut je dodasz! Użyjemy dwóch rodzajów. Pierwszy z nich to `FilledButton`[16], drugi to `FloatingActionButton` (w skrócie FAB). Każdy z nich wywoła jakąś akcję. Pierwszy zaktualizuje kolor tła aplikacji, tak aby utworzył się gradient, a drugi będzie miał za zadanie wyświetlić `Snackbar`, czyli czasowy alert wyświetlany zazwyczaj na dole ekranu.

FilledButton

Wróćmy do pliku *main.dart*. Wrzucimy sobie tutaj kilka zmiennych, które będą przechowywały możliwe do ustawienia wartości gradientu oraz aktualną wartość gradientu tła głównego ekranu.

Stwórz dwie statyczne listy, które będą odzwierciedlały dwa możliwe ustawienia gradientu. Dodatkowo potrzebujemy jeszcze jednej, przechowującej aktualny wybór. Dodaj je w klasie stanu (`State<MyApp>`).

```
static List<Color> gradient1 = [MyColors.ufoGreen, MyColors.melon];
static List<Color> gradient2 = [MyColors.trueV, MyColors.hopbush];

List<Color> currentGradient = gradient1;
```

[15] Flagą często nazywamy zmienną typu `bool`.

[16] Flutter ma jeszcze dwa inne rodzaje: `TextButton` oraz `OutlinedButton`.

Kolejny krok to aktualizacja widżetu `Container`. Zamiast konkretnego koloru zastosujemy listę kolorów. W tym celu użyjemy znanej już klasy `BoxDecoration` do ustawienia stylu kontenera. Nowością będzie klasa `LinearGradient`. Co więcej, ustawimy gradient pod kątem 45 stopni, co wywoła ciekawy efekt.

Zanim przejdziesz dalej, dodaj następującą przestrzeń. Będzie nam potrzebna, aby uzyskać dostęp do stałej `pi`.

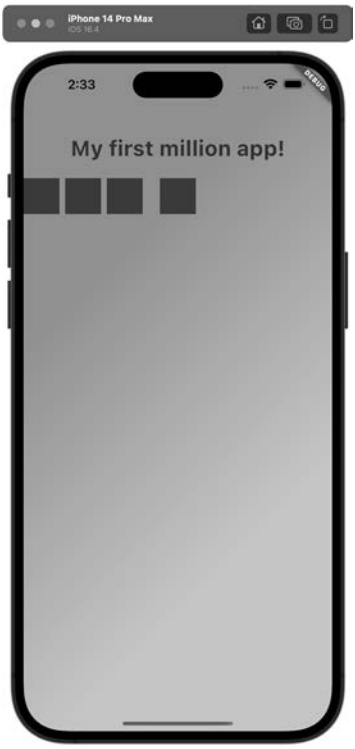
```
import 'dart:math' as math;
```

Poniżej znajdziesz zaktualizowaną implementację kontenera.

```
Container(
  decoration: BoxDecoration(
      gradient: LinearGradient(
          transform: const GradientRotation(math.pi / 4),
          colors: currentGradient)),
  child: Padding(
    padding: const EdgeInsets.only(top: 30),
    child: _buildContent(),
  ),
),
```

Uruchom aplikację, aby zobaczyć zmiany.

RYSUNEK 4.3.
Oczekiwany efekt



Dodajmy teraz wspomniany już przycisk. Stwórz nową metodę.

```
Widget _buildButton() {
  return FilledButton(
      style: FilledButton.styleFrom(backgroundColor: MyColors.vividCrimson),
      onPressed: () {
        setState(() {
          currentGradient =
              currentGradient == gradient1 ? gradient2 : gradient1;
        });
      },
      child: const Text(
        "Update background",
        style: TextStyle(fontWeight: FontWeight.bold),
      ));
}
```

Zgodnie z dotychczasową konwencją dodaj wywołanie tej metody w metodzie `_build ↪Content`. Aby spersonalizować przycisk, skorzystaliśmy z wywołania `FilledButton. ↪stylesFrom()`. Ma ona wiele możliwych ustawień, co pozwala dostosować przycisk do swoich potrzeb. My skorzystaliśmy tylko z ustawienia koloru tła.

Sprawdź, jaki będzie rezultat, gdy zamienisz typ `Container` na `AnimatedContainer`. W razie problemów podejrzyj implementację widżetu `MyAnimatedContainer`.

FloatingActionButton

Geneza tego przycisku jest powiązana z systemem projektowania **Material Design**[17]. Korzystamy z niego wtedy, gdy chcemy umożliwić użytkownikowi wykonanie jakiejś jednej, głównej akcji powiązanej z ekranem, na którym się znajduje. Dzięki temu, że Flutter w pełni implementuje ten system, możemy skorzystać z predefiniowanego miejsca, gdzie możemy ten przycisk umieścić. Nie musimy martwić się o jego pozycjonowanie czy bazowy styl.

Przejdź do widżetu `Scaffold` na górze pliku *main.dart*. Wprowadź poniższą zmianę.

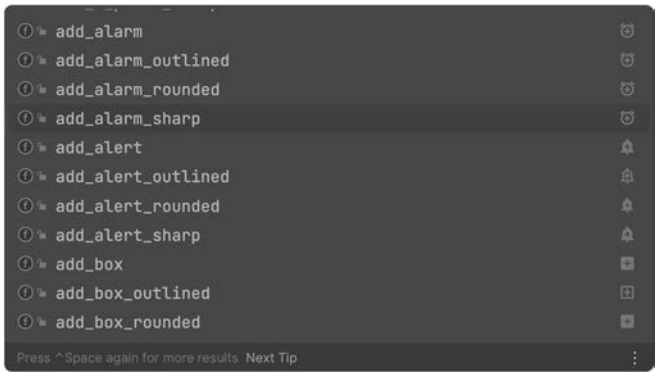
```
floatingActionButton: FloatingActionButton(
  onPressed: () {},
  backgroundColor: MyColors.fireEngineRed,
  child: Icon(
    Icons.warning_amber,
    color: MyColors.white,
  ),
),
```

Po odświeżeniu zobaczysz FAB w prawym dolnym rogu ekranu. Świetna sprawa. Nadal masz możliwość sporej personalizacji wyglądu tego przycisku z zachowaniem bazowego działania tego widżetu. Przy okazji pojawił się nowy widżet `Icon`. Pozwala on (a jakże!) wyświetlić ikony. Baza ikon Material Design jest ogromna! Wszystkie je

[17] Sprawdź szczegóły na *https://material.io*.

znajdziesz w klasie `Icons`[18]. Co więcej, możesz sprawdzić wygląd konkretnej ikony, zanim użyjesz jej w kodzie. Wystarczy, że pozwolisz środowisku na wyświetlenie listy możliwych ikon. Obok nazwy ikony zobaczysz również jej podgląd. Tak jak to widać na rysunku 4.4.

RYSUNEK 4.4. Przykładowe ikony klasy Icons



Obiecałem Ci wyświetlenie snackbara. Umieść poniższy kod w delegacie przypisanym do `onPressed` przycisku FAB.

```
ScaffoldMessenger.of(context)
    .showSnackBar(const SnackBar(content: Text("FAB was clicked!")));
```

Nie wchodząc w szczegóły, zadaniem metody `of` klasy `ScaffoldMessenger` jest — na podstawie aktualnego kontekstu — wyszukanie widżetu `Scaffold` w drzewie widżetów i na jego podstawie pokazanie snackbara.

Sam kod pokazania snackbara jest poprawny, ale obecna implementacja klasy `MyApp` nie zadziała z nim poprawnie. Przekonasz się o tym, kiedy uruchomisz aplikację i klikniesz przycisk. `Snackbar` się nie pojawi, a w konsoli zobaczysz komunikaty podobne do poniższych.

```
======== Exception caught by gesture ===============================================
The following assertion was thrown while handling a gesture:
No ScaffoldMessenger widget found.

MyApp widgets require a ScaffoldMessenger widget ancestor.
The specific widget that could not find a ScaffoldMessenger ancestor was: MyApp
  state: _MyAppState#2Ocb5
The ancestors of this widget were:
  : [root]
    renderObject: RenderView#46ff9
```

Wygląda na to, że środowisko ma problem ze znalezieniem widżetu `ScaffoldMessenger`. I wcale mu się nie dziwię. Jest jeden szkopuł, który to psuje. Dokładny opis problemu znajdziesz we wskazówce na końcu tego podrozdziału. Rozwiążmy ten problem. W zasadzie mamy to, czego potrzebujemy, ale aby wszystko zadziałało poprawnie,

[18] Pod adresem *https://fonts.google.com/icons* znajdziesz listę wszystkich ikon dostępnych w systemie Material Design.

musimy wpłynąć na strukturę naszego kodu. Stwórz nowy plik *demo_page.dart*. Przeniesiemy tam większość kodu budującego obecną zawartość ekranu. Nie zamieszczam tutaj docelowego kodu, który musi się tam znaleźć. Podpowiedzią dla Ciebie niech będzie docelowa implementacja klas `MyApp` po wprowadzeniu zmian, czyli to, co powinno zostać w pliku *main.dart* (oprócz samej funkcji `main()` oczywiście). Co więcej, nie korzystaj z widżetu `MaterialApp` w klasie `DemoPage`. W razie problemów sprawdź repozytorium kodu dołączonego do książki[19].

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(home: DemoPage());
  }
}
```

Zwróć uwagę, że zmieniła się klasa bazowa dla widżetu `MyApp`. Jeśli wszystko zrobiłeś poprawnie, to powinieneś zobaczyć wiadomość u dołu ekranu.

Snackbar

Dlaczego napotkaliśmy problem z pokazaniem się snackbara? Otóż kłopot w tym, że obiekt `context`, którego użyliśmy, został stworzony przez widżet utworzony przez widżet `Scaffold`. Oznacza to, że nie miał on informacji, że `Scaffold` istnieje w drzewie. Rozwiązaniem jest stworzenie nowego widżetu i zwrócenie go w widżecie `MaterialApp`. To spowoduje użycie poprawnego obiektu `context`, który zawiera już informację o widżecie `Scaffold`.

Myśl deklaratywnie

Flutter z natury jest deklaratywny, co oznacza, że tworząc interfejs użytkownika, bazujemy na jakimś stanie. Zwróć uwagę, w jaki sposób budowaliśmy do tej pory widoki. W widżecie `MyAnimatedContainer` zwracaliśmy różny styl kontenera, bazując na wartości zmiennej `shouldUseFancyShape`. W metodzie `build` zwracaliśmy po prostu widok i nie mieliśmy żadnego kawałka kodu, który odwoływałby się bezpośrednio np. do kontenera i przypisywał mu nowy kształt. Po prostu rezultat metody `build` był oparty na tym, jaki widżet zwrócimy. Przeanalizuj poniższe dwa przykłady.

```
// Declarative style
return ViewB(
  color: red,
  child: const ViewC(),
);

// Imperative style
b.setColor(red)
```

[19] Linki do zasobów znajdziesz w części III książki.

```
b.clearChildren()
ViewC c3 = new ViewC(...)
b.add(c3)
```

Odwrotne podejście prezentuje styl imperatywny, który widzisz w drugim przykładzie. Jest on zdecydowanie bardziej popularny. Tutaj musimy odwoływać się do konkretnych elementów interfejsu użytkownika i przypisywać jego właściwościom nowe wartości.

Podstawowa różnica jest taka, że gdy budujemy widok we Flutterze, niejako deklarujemy, jak on powinien się zachowywać. Nie mówimy, co ma się dziać krok po kroku, jak ma to miejsce w podejściu imperatywnym. Tworzymy swego rodzaju szkielet, szablon widoku.

Jakie niesie to konsekwencje? Jedną z nich jest kwestia przebudowywania. Aby widok został odświeżony, musi zostać na nowo przebudowany. Oznacza to ponowne stworzenie drzewa widżetów. Twórcy Fluttera zadbali, aby ten proces był wydajny. Zastosowano w nim wiele rozwiązań, które pozwalają go zoptymalizować, więc na początku nauki Fluttera zdecydowanie nie przejmuj się wydajnością aplikacji. Wbrew pozorom trzeba się naprawdę postarać, aby aplikacja napisana we Flutterze przestała być płynna. Pamiętaj jednak, że wywoływanie metody `setState()` nie jest mimo wszystko darmowe. Środowisko zawsze wykorzystuje jakieś zasoby sprzętowe. Jeśli nagromadzisz mnóstwo takich wywołań, może stać się to widoczne dla użytkownika.

Celem tego podrozdziału było przedstawienie, że istnieje coś takiego jak style deklaratywny i imperatywny. Dobrze jest zdawać sobie z tego sprawę.

Podsumowanie

Celem tego rozdziału było zaznajomienie Cię z podstawowymi pojęciami. Zasadniczą różnicę pomiędzy `StatelessWidget` a `StatefulWidget` stanowi to, że ten pierwszy nie może samodzielnie zmienić swojego stanu. Taką możliwość ma za to `StatefulWidget`, co oznacza, że jest typem, który pozwala programiście na znacznie więcej, a żądanie odświeżenia stanu polega na wywołaniu metody `setState()`.

Wiesz, że `Container` jest jednym z podstawowych widżetów. Dowiedziałeś się, jak ustawić inny kształt dla widżetu `Container` dzięki użyciu `BoxDecoration` oraz jak dodać margines do jakiegoś elementu dzięki widżetowi `Padding`. Dodanie prostej animacji do kontenera jest bardzo łatwe dzięki użyciu widżetów w wersji `Animated`. Miałeś okazję tego doświadczyć dzięki wykorzystaniu `AnimatedContainer`.

Jako pierwsze poznałeś widżety układu `Row` oraz `Column`. Pozwalają one kontrolować pozycjonowanie kilku elementów poziomo lub pionowo. Dzięki parametrom `CrossAxisAlignment` oraz `MainAxisAlignment` możesz wpłynąć na strategię pozycjonowania elementów w tych układach.

Podstawowa konfiguracja projektu odbywa się poprzez pracę z plikiem *pubspec.yaml*. Miałeś okazję dodać pierwszą zewnętrzną bibliotekę do projektu poprzez aktualizację sekcji `dependencies` w tym pliku.

Zadania do samodzielnego wykonania:

1. Zaktualizuj kod klasy `MyAnimatedContainer` o użycie widżetu `AnimatedPadding`. Zrób tak, aby po jego kliknięciu pojawiła się animowana zmiana jego marginesów.
2. Dodaj opcjonalny parametr, który pozwoli Ci przekazać tekst do konstruktora `MyAnimatedContainer`. Wyświetl jego wartość w widżecie `Text`. Dodatkowo wycentruj go względem kontenera (wykorzystaj widżet `Center` oraz parametr `TextAlignment` w widżecie `Text`).
3. Dodaj obsługę nowego dowolnego gestu w `MyAnimatedContainer`. Zmieniaj kolor kontenera na dowolnie przez siebie wybrany.
4. Przycisk do zmiany gradientu znajduje się obecnie bezpośrednio pod wierszami widżetów `MyAnimatedContainer`. Spróbuj go wycentrować pionowo względem dostępnej przestrzeni na ekranie. W tym celu wykorzystaj widżet `Spacer`. Eksperymentuj, aby dojść do rozwiązania.

Flutter
Podstawy

Witaj w świecie (projektowania) aplikacji mobilnych

Jesteś początkującym programistą, poszukujesz swojej ścieżki i zastanawiasz się właśnie nad tym, czy nie skierować uwagi w stronę aplikacji mobilnych? To książka dla Ciebie. Zawarty w niej materiał jest odpowiedni dla osoby dysponującej podstawami któregoś z języków programowania i bazowym doświadczeniem w pracy z platformą Android lub iOS. Ten przystępny przewodnik pozwoli Ci napisać własną aplikację, a następnie wydać ją w sklepie Google Play czy App Store. Jeśli poświęcisz trochę więcej czasu, umożliwi Ci także stworzenie aplikacji internetowej. A wszystko to z wykorzystaniem jednej bazy kodu. Brzmi zachęcająco?

Poznaj Fluttera — narzędzie do tworzenia natywnych, wieloplatformowych aplikacji mobilnych, komputerowych i internetowych. Dzięki pracy z poświęconym mu poradnikiem opanujesz podstawy języka Dart, na którym bazuje framework Flutter. Uzbrojony w niezbędną wiedzę teoretyczną, przejdziesz do części praktycznej. Przygotujesz cztery aplikacje o różnym stopniu komplikacji — od demo po aplikację służącą do rozliczania czasu spędzonego na danym zadaniu. To pozwoli Ci poznać pełnię możliwości frameworka. Zrozumiesz też zastosowanie wzorca projektowego MVVM i sposoby pracy z logiką aplikacji. Co więcej, prześledzisz dobre praktyki w zakresie działania z widokami Flutter i architekturą oprogramowania. Wszystko po to, by stworzyć własną aplikację, która podbije serca użytkowników smartfonów na całym świecie!

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-8322-644-6

9 788383 226446

Cena: 69,00 zł