Flutter

Systemy mobilne

Dr inż. Szymon Sieciński

Wydział Informatyki AGH

Kraków, 24.11.2025 r.

Wymagania

- Instalacja Flutter SDK (przez pobranie do znanego folderu lub instalację przez pakiet)
- Zainstalowanie wtyczki Flutter w preferowanym IDE
- Około 2,5 GB wolnego miejsca na dysku na Flutter SDK + miejsce na projekty
- Instalacja Git (do klonowania szkieletu projektu oraz instalacji nowych wersji)
- Kompilatory na platformy docelowe:
    - Web: nie są wymagane dodatkowe kompilatory, ale zalecana jest przeglądarka Google Chrome
    - Windows: Microsoft Visual Studio Build Tools, MSVC, Windows SDK, opcjonalnie MSBuild, CMake itp.
    - Linux: Clang, CMake, ninja-build, pkg-config, libgtk-3 (dev), libstdc++ (dev)
    - macOS, iOS: XCode (Clang, Swift, iOS/macOS SDK, xcodebuild, Command Line Tools),
    - Android: JDK (Java/Kotlin), Android SDK, Android SDK Build Tools, Android Platform Tools, Gradle, Android NDK oraz CMake (opcjonalnie; kod natywny C/C++)

  Wymagany jest co najmniej jeden kompilator do trybu debugowania.

Instalacja

Microsoft Visual Studio Code (VS Code)

1. **Uruchom VS Code**

Jeśli nie jest jeszcze otwarty, otwórz program VS Code, wyszukując go za pomocą paska Szukaj (Windows), funkcji Spotlight (macOS), Aktywności (GNOME) lub otwierając go ręcznie z katalogu, w którym jest zainstalowany.

2. **Dodaj rozszerzenie Flutter do VS Code**

Aby dodać rozszerzenia Dart i Flutter do programu VS Code, odwiedź stronę rozszerzenia Flutter w witrynie Marketplace, a następnie kliknij przycisk **Zainstaluj**. Jeśli przeglądarka wyświetli monit, zezwól jej na otwarcie programu VS Code.

3. **Zainstaluj Flutter w VS Code**

    1. Otwórz paletę komend w VS Code.

Przejdź do **View** > **Command Palette** lub wciśnij Control/Command + Shift + P.

    2. W palecie komend wpisz `flutter`.
    3. Wybierz **Flutter: New Project**.
    4. VS Code poprosi o wskazanie ścieżki do Flutter SDK na komputerze. Jeśli nie zainstalowano, wybierz **Download SDK**.
    5. Po wyświetleniu okna **Select Folder for Flutter SDK** wybierz folder, w którym zostanie zanistalowany Flutter.

4. Kliknij **Clone Flutter**.

Podczas pobierania Flutter, VS Code wyświetli komunikat:

> Downloading the Flutter SDK. This may take a few minutes.

5. Pobieranie może trwać kilka minut. Jeśli podejrzewasz, że pobieranie stanęło, kliknij **Cancel** i rozpocznij instalację od nowa.
6. Kliknij **Add SDK to PATH**.

Po zainstalowaniu pojawi się komunikat:

> The Flutter SDK was added to your PATH
>
> *Flutter SDK został dodany do ścieżki systemowej (PATH)*

VS Code może wyświetlić komunikat o Google Analytics.

Jeśli się zgadzasz, kliknij **OK**.

7. Aby sprawdzić, czy Flutter działa we wszystkich terminalach:
    a. Zamknij i otwórz ponownie okna terminala (wiersza poleceń).
    b. Uruchom ponownie VS Code.

IntelliJ IDEA, Android Studio i podobne

1. Pobierz Flutter SDK
2. Rozpakuj archiwum z SDK do folderu użytkownika

3. Dodaj Flutter SDK do Ścieżki systemowej
4. Sprawdź instalację poleceniami
   ```
   flutter --version
   dart --version
   ```
5. Uruchom IntelliJ IDEA / Android Studio lub podobne IDE
6. Przejdź do **Settings > Plugins**
7. Wybierz lub wyszukaj „Flutter” i kliknij „Install”. Można też dodatkowo wybrać „Dart”
8. Po instalacji uruchom IntelliJ / Android Studio lub podobne IDE

Instalacja ręczna

1. Pobierz Flutter SDK
2. Rozpakuj archiwum z SDK do folderu użytkownika
3. Dodaj Flutter SDK do Ścieżki systemowej
4. Sprawdź instalację poleceniami
   ```
   flutter --version
   dart --version
   ```

Tworzenie projektu

Microsoft Visual Studio Code (VS Code)

3. **Uruchom VS Code**

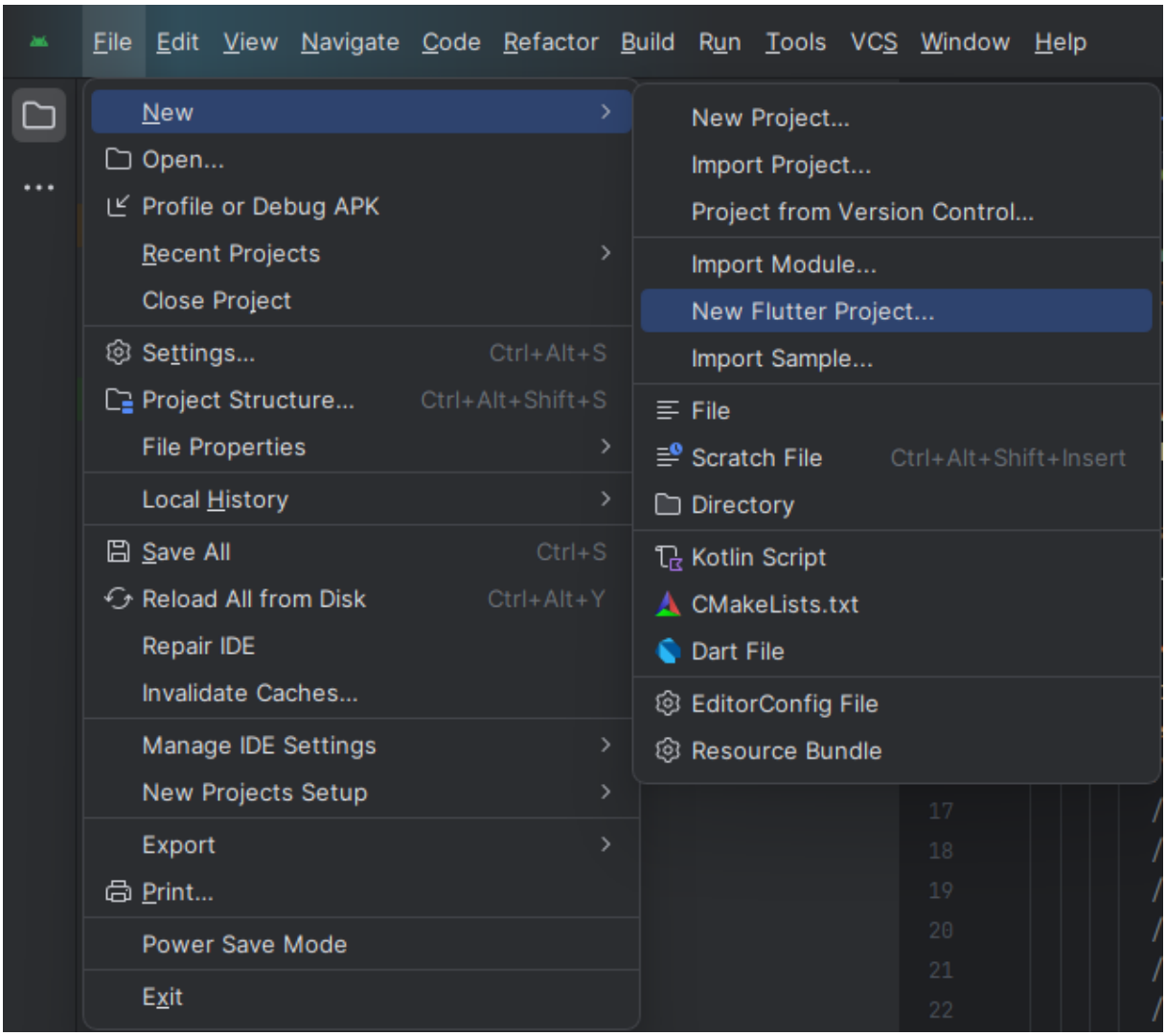
Jeśli nie jest jeszcze otwarty, otwórz program VS Code, wyszukując go za pomocą paska Szukaj (Windows), funkcji Spotlight (macOS), Aktywności (GNOME) lub otwierając go ręcznie z katalogu, w którym jest zainstalowany.

2. **Otwórz paletę komend**
    1) Przejdź do **View** > **Command Palette** lub wciśnij Control/Command + Shift + P.
    2) W palecie komend wpisz flutter.
    3) Wybierz **Flutter: New Project**.
    4) Wybierz **Application**
    5) Wpisz nazwę aplikacji. Nazwa musi być zgodna z konwencją snake_case.
    6) Wybierz folder, w którym zostanie utworzona aplikacja
3. Zaczekaj na utworzenie projektu
4. Można działać – punktem startowym jest plik **lib/main.dart**

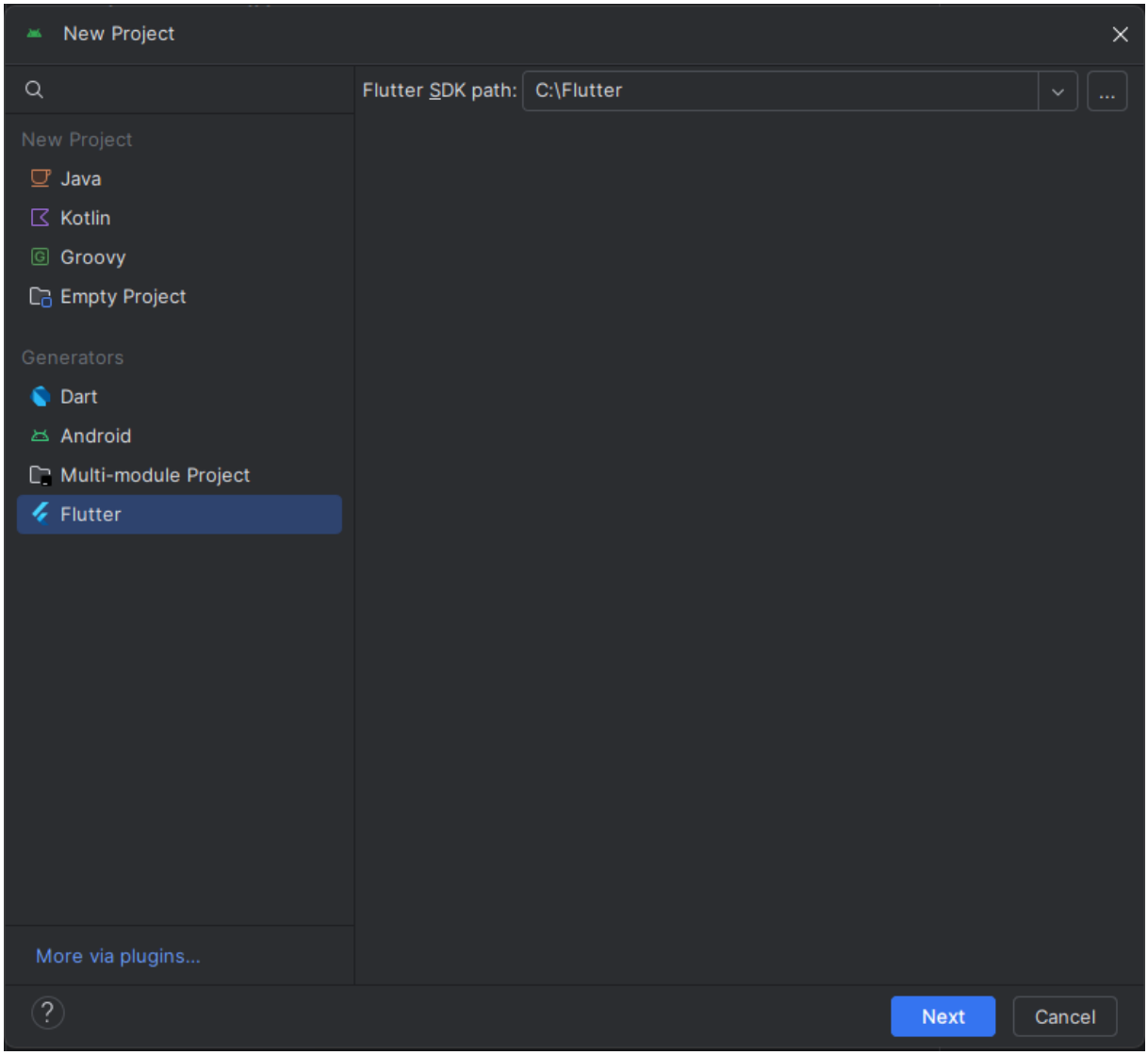
IntelliJ IDEA, Android Studio i podobne

Poniższe kroki zostały przedstawione dla Android Studio. Ponieważ Android Studio wywodzi się z IntelliJ IDEA, można w tym IDE i pochodnych zastosować te same kroki.

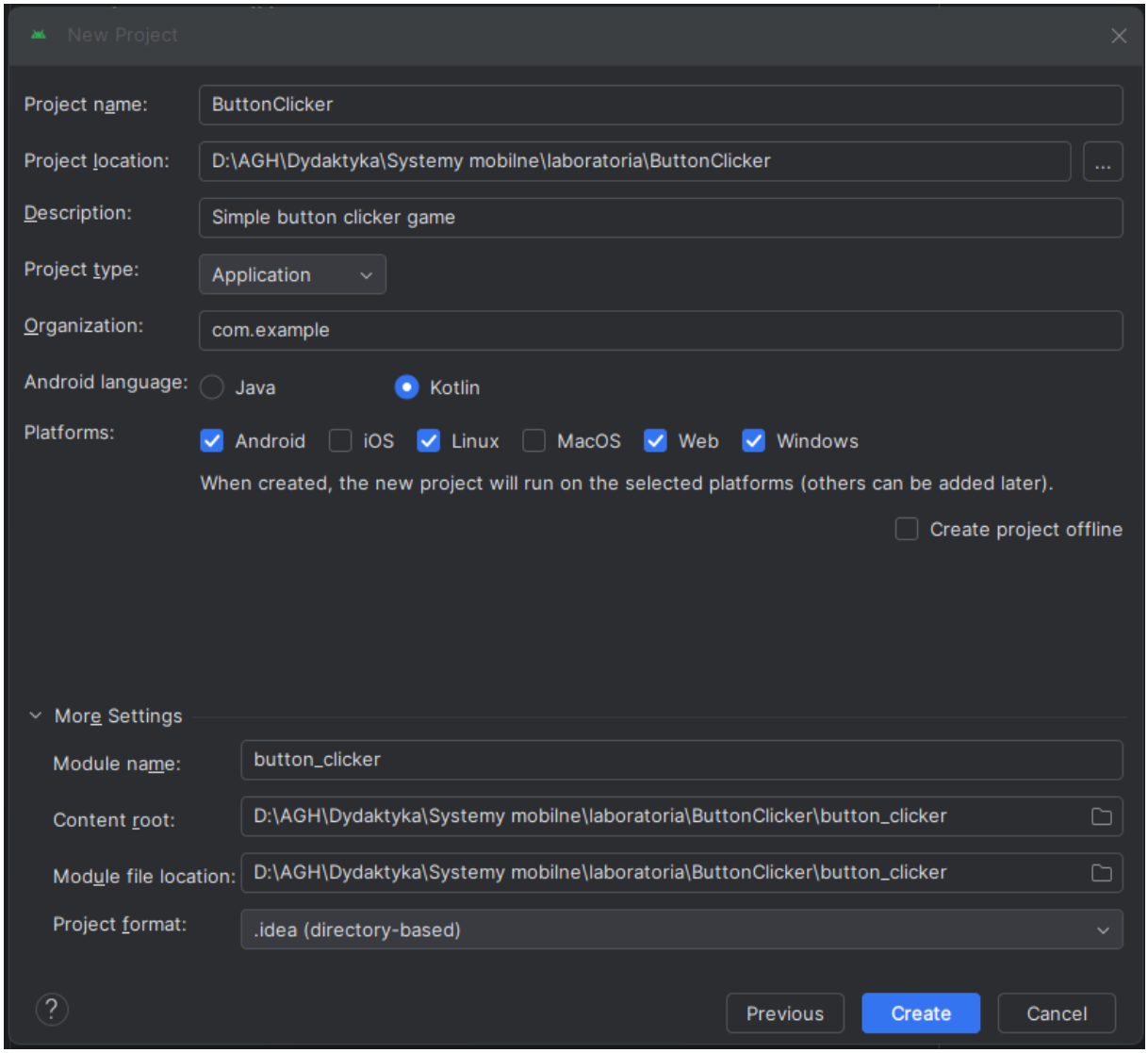
1. Uruchom IntelliJ IDEA
2. Wybierz **File > New > New Flutter Project**



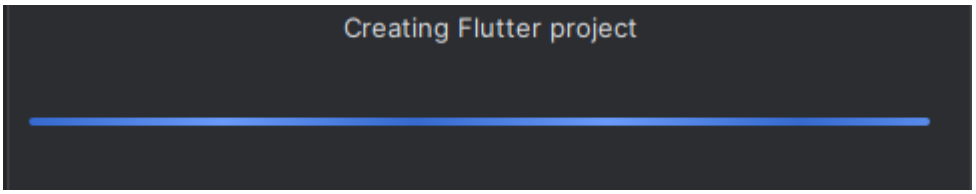
3. W poniższym oknie (New Project) wybierz **Flutter** i podaj ścieżkę do Flutter SDK:



4. W następnym oknie wpisz nazwę aplikacji (jako moduł Dart: małymi literami z podkreśleniem (_) bez użycia cyfr



5. Zaczekaj na wygenerowanie szkieletu projektu



6. Można działać – punktem startowym jest plik **lib/main.dart**

Pierwsze kroki

Naszym pierwszym zadaniem będzie analiza przykładowego kodu w pliku `lib/main.dart`., który wygląda tak:

```
import 'package:flutter/material.dart';
```

```
void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      home: const MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}

class MyHomePage extends StatefulWidget {
  const MyHomePage({super.key, required this.title});
  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

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: Text('$_counter')),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Aby uruchomić aplikację, należy kliknąć na „Run”, „Compile” lub użyć polecenia `flutter run` w wierszu poleceń/palecie poleceń.

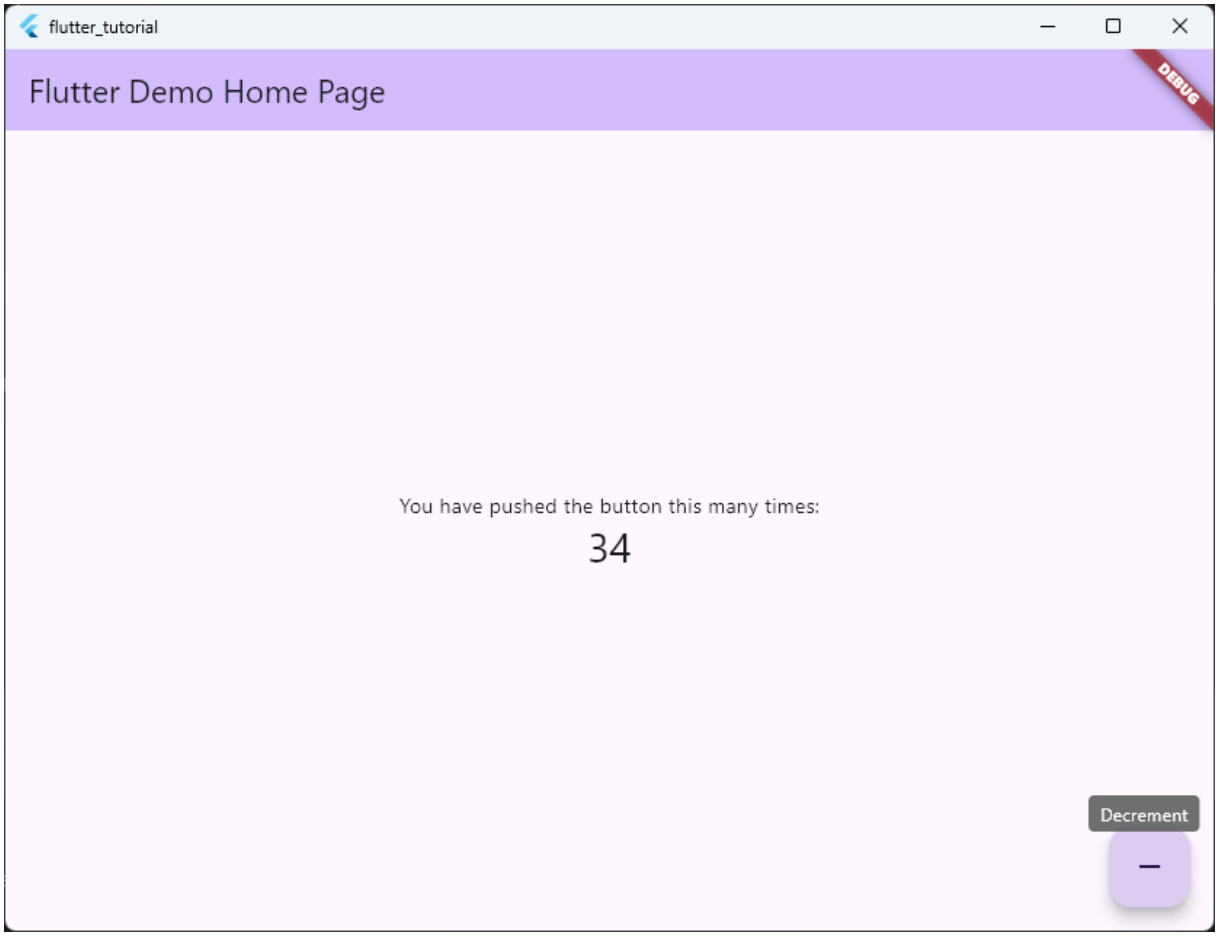
Po kompilacji powinno się pokazać okno przykładowej aplikacji:

Główną cechą Fluttera jest *hot reload* (przeładowanie na gorąco), co ułatwia testowanie aplikacji. Oznacza to przeładowywanie aplikacji wraz z zapisaniem każdej zmiany w kodzie źródłowym.

Aby się przekonać, zmieńmy jeden z następujących fragmentów kodu (plik `lib/main.dart`) i zapiszmy zmiany:

- Zmieniamy nazwę funkcji `_incrementCounter()` na `_decrementCounter()` w nagłówku oraz wszystkich wywołaniach.
- W funkcji `_decrementCounter()` zmieniamy linię `_counter++` na `_counter--`
- Zmieniamy wartość pola `title` na inny tekst
- Zmieniamy ikonę (parametr konstruktora klasy `Icon`) z `Icons.add` na `Icons.remove`
- Zmieniamy podpowiedź przycisku na `Decrement` (pole `Tooltip` w klasie `FloatingActionButton`)

Po zapisaniu zmian kod zostanie ponownie skompilowany, a uruchomiona aplikacja zostanie odświeżona. Po zastosowaniu wszystkich proponowanych zmian aplikacja będzie wyglądać tak:



Układ widżetów na ekranie

We Flutterze podstawowym elementem aplikacji jest widżet (ang. widget). Dotyczy to również układu elementów (ang. layout), ikon, tekstu, grafik itp. Układ elementów (layout) tworzymy przez składanie elementów w drzewie.

Widżety layoutu

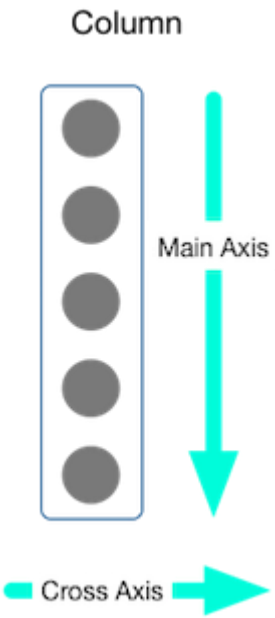
Widżety można podzielić na dwie kategorie: z jednym elementem potomnym (ang. single-child) oraz z wieloma elementami potomnymi (ang. multi-child). W pierwszej kategorii można dodać tylko jeden element (pole `child`), natomiast w drugiej elementy dodaje się elementy w liście przypisanej do pola `children`.

Pierwsza kategoria obejmuje:

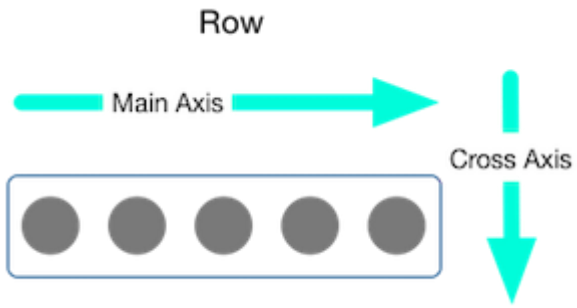
- Align (wyrównaj wewnątrz)
- AspectRatio (ustaw stosunek rozmiarów)
- Baseline (ustaw w linii)
- Center (wyśrodkuj)
- ConstrainedBox (ustaw według wymiarów)
- Container (te same właściwości dla wszystkich elementów potomnych)
- Expanded (rozszerz)
- Padding (wyrównaj)

Druga kategoria obejmuje:

- Column (kolumna – elementy ustawione pionowo w kolumnie)



- Row (wiersz – elementy ustawione poziomo w wierszu)

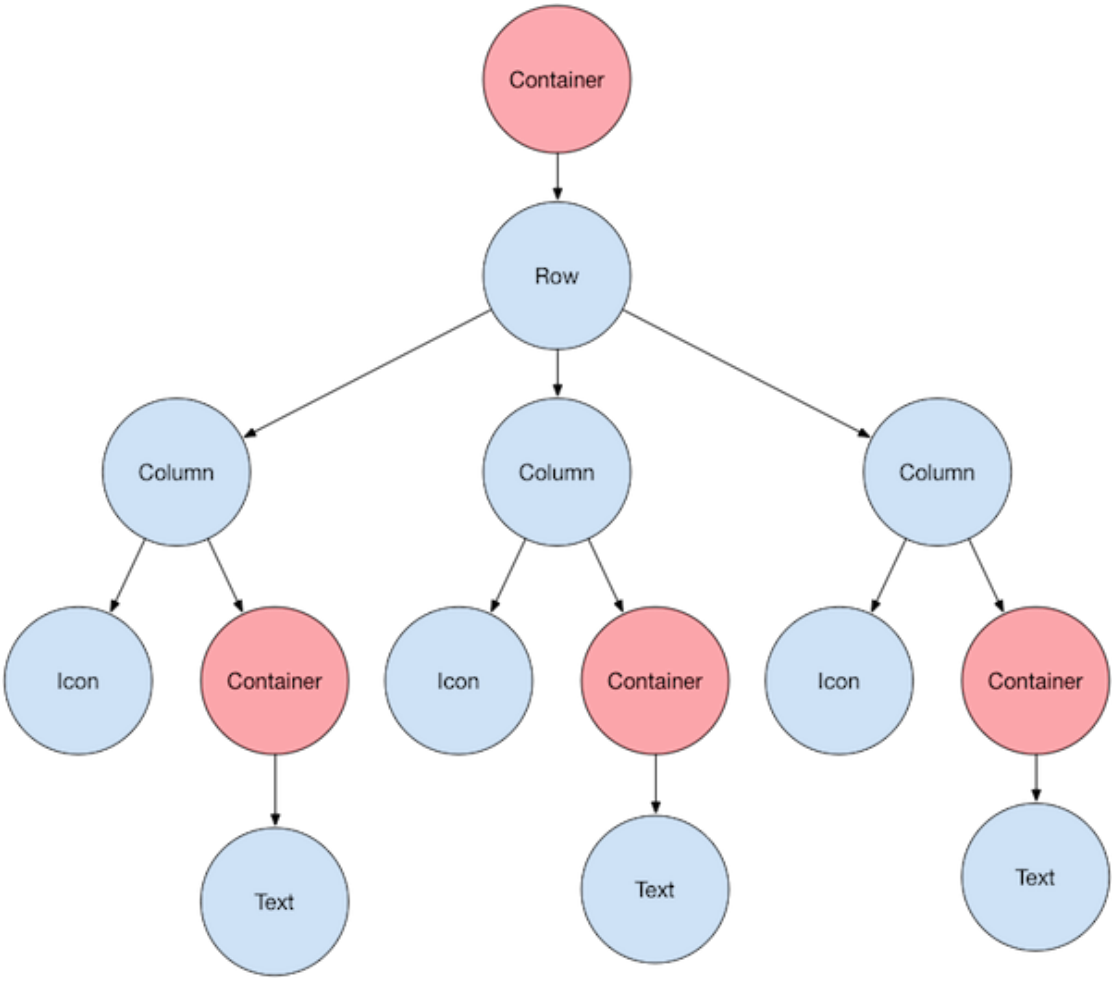


- GridView – ustaw elementy w siatce
- ListView – ustaw elementy na przewijanej liście
- Stack – ustaw elementy jeden na drugim z możliwością nakładania się
- Table – ustaw elementy w wierszach i kolumnach

Przykład

Dla elementów:

Mamy poniższe drzewo widżetów:



W tym drzewie mamy wyróżniony na różowo węzeł – klasę Container, która pozwala na dostosowanie układu elementów – marginesów, tła, wyrównania i innych właściwości.

Pozostałe właściwości sterujące wyglądem są zawarte w poszczególnych widżetach.

Widżety

Podstawowe widżety we Flutterze można podzielić na dwie kategorie: Material components (elementy zgodne z Material Design 3) oraz Cupertino (elementy zgodne z Apple's Human Interface Guidelines). Można ich używać niezależnie od platformy docelowej.

Material components

A. Szkielet widoku i elementy nawigacji

- **Scaffold**
  Główny kontener: miejsce na appBar, body, floatingActionButton, drawer, dolny pasek itd.
  Często używany w aplikacjach korzystających z Material Design.
- **AppBar**
  Górny pasek aplikacji: tytuł, ikony akcji (np. lupa, menu), przycisk powrotu.
  Często używany razem ze `Scaffold(appBar: AppBar(...))`.
- **BottomNavigationBar**
  Dolna belka do przełączania się między głównymi sekcjami aplikacji.
  Najczęściej używana w aplikacjach z 3–5 głównymi zakładkami.
- **TabBar + TabBarView**
  Zakładki na górze ekranu (np. „Dzisiaj / Tydzień / Miesiąc").
  Zwykle opakowane w DefaultTabController.
- **Drawer / NavigationDrawer**
  Wysuwane z boku menu nawigacyjne (tzw. hamburger menu).
  Standard w aplikacjach z bardziej rozbudowanym menu.

B. Tekst, przyciski, ikony

- **Text**
  Podstawowy widget do wyświetlania tekstu.
  Można kontrolować styl (TextStyle), max. liczbę linii, overflow itd.
- **Icon**
  Wyświetla ikonę z Icons.* lub własny IconData.
  Często używany w przyciskach i paskach aplikacji.

- **ElevatedButton**
  „Wyniesiony” przycisk z cieniem – klasyczny przycisk głównej akcji.
  Przykład:

```
ElevatedButton(
    onPressed: () {},
    child: const Text('Zapisz'),
)
```

- **TextButton**
  Płaski przycisk tekstowy, bez tła – używany do mniej ważnych akcji, dodatkowych linków.
- **OutlinedButton**
  Przycisk z konturem, często stosowany do akcji drugorzędnych.
- **IconButton**
  Samodzielna ikonka jako przycisk (np. w AppBar – ikonka „szukaj”, „udostępnij” itd.).
- **FloatingActionButton**
  Okrągły przycisk akcji w prawym dolnym rogu ekranu (dodaj, nowy, plus).
  Ustawiany zwykle w `Scaffold(floatingActionButton: ...)`.

C. Pola wprowadzania danych i formularze

- **TextField / TextFormField**
  Podstawowe pole tekstowe (np. login, e-mail, wyszukiwarka).
  TextFormField jest wersją z integracją z Form i walidacją.
  Częste opcje:
  - keyboardType (np. TextInputType.emailAddress)
  - obscureText dla haseł
  - decoration: InputDecoration(...) dla etykiety, ikon, podpowiedzi.
- **Checkbox**
  Kwadratowy przełącznik dla opcji typu TAK/NIE (wielokrotny wybór).
- **Radio**
  Kółeczko do wyboru **jednej** opcji z wielu (np. płeć, sposób płatności).
- **Switch**
  Przełącznik ON/OFF w stylu Material (np. „Powiadomienia włączone”).

- **Slider**
  Suwak do wyboru wartości numerycznej z zakresu (np. głośność, jasność).
- **DropdownButton**
  Lista rozwijana z wyborem jednej wartości (np. kraj, waluta).

D. Listy, elementy listy, karty

- **ListTile**
  Standardowy „wiersz" w liście: tytuł, podtytuł, ikonka na początku i/lub na końcu. Idealny do list ustawień, kontaktów, elementów menu.
- **Card**
  Karta z cieniem i zaokrąglonymi rogami.
  Używana do grupowania treści (np. kafel informacyjny, element listy z większą ilością danych).
- **Chip / FilterChip / ChoiceChip**
  Małe „pastylki" z tekstem i ewentualnie ikoną.
  Używane np. do tagów, filtrów, wyboru pojedynczej opcji.

E. Dialogi, snackbary, bottom sheets

- **AlertDialog**
  Klasyczne okno dialogowe z tytułem, treścią i przyciskami akcji („OK", „Anuluj").
  Wywoływane przez `showDialog(...)`.
- **SimpleDialog**
  Prostszy dialog – głównie lista opcji do wyboru.
- **SnackBar**
  Krótki komunikat na dole ekranu (np. „Zapisano", „Błąd połączenia").
  Pokazywany przez
  `ScaffoldMessenger.of(context).showSnackBar(...)`.
- **BottomSheet / showModalBottomSheet**
  Panel wysuwany od dołu z dodatkowymi opcjami, formularzem, menu.
  Często używany zamiast pełnego dialogu.

F. Inne przydatne komponenty

- **CircularProgressIndicator / LinearProgressIndicator**
  Wskaźnik postępu – okrągły lub liniowy (ładowanie, oczekiwanie).
- **Tooltip**
  Krótka podpowiedź pojawiająca się po najechaniu/kliknięciu na element (głównie na desktopie/webie).

Cupertino

A. Struktura i nawigacja w stylu iOS

- **CupertinoApp**
  Odpowiednik MaterialApp, start aplikacji w stylu iOS.
- **CupertinoPageScaffold**
  Podstawowy szablon strony: navigationBar, child itd.
- **CupertinoNavigationBar**
  Górny pasek nawigacyjny w stylu iOS: tytuł, przycisk „Back", przyciski po bokach.
- **CupertinoTabScaffold + CupertinoTabBar**
  Dolna belka zakładek w stylu iOS (ikony + podpis), z automatycznym zarządzaniem ekranami dla każdej zakładki.
- **CupertinoTabView**
  Widok zawartości dla konkretnej zakładki, często z własnym stosem nawigacji (Navigator).

B. Przyciski i interakcje

- **CupertinoButton**
  Podstawowy przycisk iOS (tekstowy lub z ikoną).
  Może być w wersji pełnej (wypełnione tło) lub jako przycisk bez tła.
- **CupertinoSlidingSegmentedControl**
  Kontrolka z segmentami, między którymi można się przełączać (np. „Dzisiaj / Tydzień / Miesiąc") – w stylu natywnego iOS.
- **CupertinoSwitch**
  Przełącznik ON/OFF w stylu iOS.
- **CupertinoActionSheet / CupertinoActionSheetAction**
  Dolnego menu akcji w iOS.

C. Pola tekstowe i wybór wartości

- **CupertinoTextField**
  Pole tekstowe w stylu iOS (zaokrąglone rogi, inny styl kursora i placeholdera).
- **CupertinoPicker**
  Rolkowy picker (tzw. „slot machine") – wybór jednej wartości z listy (np. godzina, miasto).
- **CupertinoDatePicker**
  Wybór daty/czasu w stylu iOS (obrócone rolki).

D. Dialogi, alerty, wskaźniki

- **CupertinoAlertDialog**
  Okno dialogowe w stylu iOS z przyciskami CupertinoDialogAction.
- **CupertinoDialogAction**
  Przyciski akcji używane wewnątrz CupertinoAlertDialog.
- **CupertinoActivityIndicator**
  Okrągły wskaźnik ładowania w stylu iOS (kręcące się kółeczko).
- **CupertinoModalPopup**
  Funkcja/rozwiązanie do wyświetlania modali w stylu iOS – np. CupertinoActionSheet, pickery.

E. Typografia i styl

- **CupertinoTheme / CupertinoThemeData**
  Ustawianie globalnej kolorystyki i stylu czcionek dla całej aplikacji w stylu iOS.
- **DefaultTextStyle** (używane także w Cupertino)
  Definiowanie domyślnego stylu tekstu dla poddrzewa widgetów.

Przekazywanie danych między widżetami

We Flutterze dane są przekazywane zazwyczaj z góry na dół hierarchii widżetów. Przekazywanie w odwrotnym kierunku jest implementowany przez callbacki (funkcje obsługujące zdarzenia) lub zarządzanie stanem.

Jest kilka mechanizmów:

1. **Przekazywanie przez konstruktor**
   Jest to najbardziej podstawowy sposób przekazywania danych do elementów podrzędnych. Polega na przekazywaniu danych przez parametry konstruktora, które są definiowane jako pola z modyfikatorem `final`.

```
class ParentWidget extends StatelessWidget {
  final String userName = "Szymon";

  @override
  Widget build(BuildContext context) {
    return ChildWidget(
      userName: userName, // przekazywanie danych
    );
  }
}
```

```
class ChildWidget extends StatelessWidget {
  final String userName;

  const ChildWidget({super.key, required this.userName});

  @override
  Widget build(BuildContext context) {
    return Text("Witaj, $userName!");
  }
}
```

2. Przekazywanie przez callback
   Element potomny wywołuje callback elementu nadrzędnego:

```
class ParentWidget extends StatefulWidget {
  const ParentWidget({super.key});

  @override
  State<ParentWidget> createState() => _ParentWidgetState();
}

class _ParentWidgetState extends State<ParentWidget> {
  String _text = "";

  void _onTextChanged(String newText) {
    setState(() {
      _text = newText;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text("Aktualny tekst: $_text"),
        ChildInputWidget(
          onTextChanged: _onTextChanged, // przekazanie
callbacku
        ),
      ],
    );
  }
```

```
}

class ChildInputWidget extends StatelessWidget {
  final ValueChanged<String> onTextChanged;

  const ChildInputWidget({super.key, required
this.onTextChanged});

  @override
  Widget build(BuildContext context) {
    return TextField(
      onChanged: onTextChanged, // wywołanie callbacku
    );
  }
}
```

3. Przekazywanie danych przez obiekt nadrzędny

   Ta metoda polega na przekazywaniu wartości do wspólnej klasy przechowującej stan

```
class Parent extends StatefulWidget {
  const Parent({super.key});


  @override
  State<Parent> createState() => _ParentState();
}


class _ParentState extends State<Parent> {
  int _counter = 0;


  void _increment() {
    setState(() {
      _counter++;
    });
  }
```

```
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        CounterDisplay(value: _counter),
        CounterButton(onPressed: _increment),
      ],
    );
  }
}

class CounterDisplay extends StatelessWidget {
  final int value;
  const CounterDisplay({super.key, required this.value});

  @override
  Widget build(BuildContext context) => Text("Licznik:
$value");
}

class CounterButton extends StatelessWidget {
  final VoidCallback onPressed;
  const CounterButton({super.key, required this.onPressed});

  @override
  Widget build(BuildContext context) =>
      ElevatedButton(onPressed: onPressed, child: const
Text("Dodaj"));
```

```
}
```

4. Użycie klasy dziedziczącej po `ChangeNotifier`
   Przy większych drzewach UI przekazywanie tych samych danych przez dużą liczbę konstruktorów staje się uciążliwe. Wtedy używamy:
   - niskopoziomowo: **InheritedWidget / InheritedNotifier**
   - lub **Provider**, **Riverpod**, **BLoC**, itp.

   Model stanu:
   ```
   class CounterModel extends ChangeNotifier {
     int value = 0;

     void increment() {
       value++;
       notifyListeners();
     }
   }
   ```
   Propagacja w drzewie:
   ```
   void main() {
     runApp(
       ChangeNotifierProvider(
         create: (_) => CounterModel(),
         child: const MyApp(),
       ),
     );
   }
   ```

   Widżety podrzędne:
   ```
   class CounterText extends StatelessWidget {
     const CounterText({super.key});

     @override
     Widget build(BuildContext context) {
       final counter = context.watch<CounterModel>();

       return Text("Licznik: ${counter.value}");
     }
   }

   class CounterButton extends StatelessWidget {
   ```

```
  const CounterButton({super.key});

  @override
  Widget build(BuildContext context) {
    final counter = context.read<CounterModel>();

    return ElevatedButton(
      onPressed: counter.increment,
      child: const Text("Dodaj"),
    );
  }
}
```

5. Przekazywanie zmian przez wywołanie `setState(() {})` wewnątrz klasy potomnej `StatefulWidget`.

   Jest to najprostsza metoda pozwalająca na przekazanie danych między widżetami wewnątrz jednej klasy:

```
class SimpleCounter extends StatefulWidget {

  const SimpleCounter({super.key});


  @override

  State<SimpleCounter> createState() =>
_SimpleCounterState();

}


class _SimpleCounterState extends State<SimpleCounter> {

  int _value = 0;


  @override

  Widget build(BuildContext context) {

    return Column(

      children: [

        Text("$_value"),
```

```
            ElevatedButton(
              onPressed: () => setState(() => _value++),
              child: const Text("Dodaj"),
            ),
          ],
        );
    }
  }
```

Nawigacja między ekranami

Podmiana widżetów

Najprostszą formą nawigacji jest podmiana widżetu we wskazanym miejscu widżetu nadrzędnego w odpowiedzi na zdarzenie.

Wybór odpowiedniego stanu

```
IconButton(
  onPressed: () {
    setState(() {
      pageId = 0;
    });
  },
  tooltip: 'Home',
  icon: Icon(Icons.home)
)
```

Wybór odpowiedniej klasy

```
Widget page;
switch(pageId) {
  case 0: page = WordGeneratorPage();
  case 1: page = FavoritesPage();
  default: page = WordGeneratorPage();
}
```

Podmiana widżetu

```
Expanded(
  child: Container(
    color: Theme.of(context).colorScheme.primaryContainer,
    child: page
  )
)
```

Klasa Navigator

Do bardziej profesjonalnej nawigacji między ekranami używamy klasy `Navigator`. W miejscu przejścia do konkretnego ekranu dodajemy wywołanie `Navigator.push` jak poniżej:

```
Navigator.push(
      context,
      MaterialPageRoute(
            builder: (context) => NowyEkran()
      )
);
```

Dla widżetów Cupertino używamy `CupertinoPageRoute` z identycznymi parametrami jak dla `MaterialPageRoute`. Do przekazania danych do nowego ekranu używamy parametrów konstruktora.

Aby wycofać do poprzedniego ekranu, używamy funkcji

```
Navigator.pop(context);
```

Zadania do wykonania

1. Zmodyfikuj domyślny projekt tak, aby liczba kliknięć zmniejszała się od wartości początkowej (np. 50) do zera. Osiągnięcie zera przywraca początkową wartość.
2. Stwórz aplikację, która pozwala obliczyć stosunek obwodu w pasie do wzrostu (ang. waist-to-height ratio, WHtR). W oknie aplikacji muszą się znaleźć:
   - Nagłówek z tytułem
   - Dwa pola tekstowe dla obwodu w pasie [cm] oraz wzrostu [cm]
   - Przycisk „Oblicz”
   - Wyświetlanie wyniku
   - Walidacja wejścia (wprowadzanie liczb dodatnich, unikanie dzielenia przez zero)

Wzór na WHtR: $WHtR = \frac{Wc}{Ht}$, gdzie Wc jest obwodem w pasie, a Ht wzrostem.

Można dodać wizualizację wyniku na skali otyłości brzusznej.

3. Stwórz aplikację, która będzie składać się z dwóch ekranów i ma wspólny nagłówek:
   - Na pierwszym ekranie użytkownik wpisuje imię/nazwisko/pseudonim/nazwę i zatwierdza przyciskiem
   - Na drugim ekranie widać powitanie tekstowe oraz przycisk powrotu do poprzedniego ekranu.

Prześlij kod i zrzuty ekranowe lub screencast przedstawiający/e działanie aplikacji w końcowym etapie.

Bibliografia

1. Flutter Docs. https://docs.flutter.dev/. Data dostępu: 17 listopada 2025 r.
2. Flutter API reference documentation. https://api.flutter.dev/. Data dostępu: 17 listopada 2025 r.