

ĆWICZENIE LABORATORYJNE - FLUTTER 1

Dart · Flutter · Widżety · Stan · Nawigacja · REST API

Programowanie Aplikacji Mobilnych - moduł Flutter

Projekt: WeatherApp - aplikacja pogodowa z Open-Meteo API

Wymagania wstępne: znajomość podstaw programowania obiektowego, zainstalowany Flutter SDK

Spis treści

1. Czym jest Flutter i dlaczego warto go poznać?	1
2. Instalacja i konfiguracja środowiska	2
3. Struktura projektu Flutter	4
4. Dart - podstawy języka	5
5. Widżety - fundamenty UI we Flutterze	8
6. Zarządzanie stanem - od setState do Provider	11
7. Nawigacja - Navigator 2.0 i GoRouter	14
8. Pobieranie danych z REST API	15
9. Motyw aplikacji - Material Design 3	18
10. Projekt WeatherApp - architektura i struktura	20
11. Zadania do wykonania	21
12. Kryteria oceniania	22
13. Dart vs Kotlin - tabela porównawcza	22
14. Najczęstsze błędy i ich rozwiązania	23

1. Czym jest Flutter i dlaczego warto go poznać?

Flutter to otwartoźródłowy framework firmy Google do tworzenia aplikacji na wiele platform jednocześnie z jednej bazy kodu. Jedna aplikacja napisana we Flutterze działa natywnie na Androidzie, iOS, Windows, macOS, Linuksie i w przeglądarce - bez konieczności przepisywania. W tym ćwiczeniu skupimy się na Androidzie, ale cała wiedza przenosi się na pozostałe platformy.

1.1. Flutter kontra natywny Android - kluczowa różnica

Zanim przejdziemy do kodu, zrozummy fundamentalną różnicę architektoniczną. Ma ona ogromne znaczenie dla wydajności i filozofii pisania UI.

Analogia: architekt kontra tłumacz

NATYWNY ANDROID (Kotlin/Compose): Twój kod rozmawia bezpośrednio z systemem Android.

To jak architekt, który osobiście wydaje polecenia murarskim mistrzom. Pełna kontrola, ale każda platforma (iOS, Android) to inni mistrzowie - musisz znać oba języki.

FLUTTER: Framework rysuje KAŻDY piksel interfejsu samodzielnie na płótnie (Canvas).
To jak artysta z własnym zestawem farb - nie zależy od platformy, bo sam maluje.
Silnik renderujący (Impeller / Skia) zamienia widgety w piksele z prędkością 60/120 fps.
Konsekwencja: wygląd aplikacji Flutter jest IDENTYCZNY na Androidzie i iOS.
Minusem: nie używa natywnych komponentów UI (przycisków systemu) - ma własne.

1.2. Flutter vs inne podejścia do cross-platformowe

Spójrzmy na paradygmaty tworzenia aplikacji wieloplatformowych z perspektywy architektury wykonawczej i kosztów utrzymania. Flutter implementuje własny silnik renderujący (Skia/Impeller), eliminując warstwę pośrednią (bridge), co skutkuje deterministycznym UI i wysoką wydajnością. W przeciwieństwie do tego, podejścia takie jak React Native wykorzystują most komunikacyjny między kodem wysokiego poziomu, a natywnymi komponentami, co wprowadza narzut latencji. Kotlin Multiplatform separuje logikę od warstwy prezentacji, maksymalizując zgodność z natywnym UI kosztem duplikacji widoków. Xamarin/MAUI bazuje na runtime .NET i również stosuje warstwę translacji. Wybór technologii jest kompromisem między wydajnością, spójnością UI i kosztami rozwoju.

Podejście / przykład	Jak działa / wady i zalety
Flutter (Dart)	Własny silnik renderujący. Brak mostu do natywnego UI - pełna wydajność. Jeden kod, wiele platform. Wymaga nauki Dart.
React Native (JavaScript)	Most JS↔natywne UI. Używa prawdziwych widжетów platformy. Wolniejszy (bridge). Większa społeczność frontendowa.
Kotlin Multiplatform (KMP)	Dzieli logikę biznesową, UI piszesz osobno dla każdej platformy. Najlepsza integracja z natywnym UI. Brak jednego UI.
Xamarin / MAUI (.NET)	Most do natywnych widжетów przez .NET. Dobry dla firm z kodem C#. Mniejsza społeczność mobilna.
Natywny (Android/iOS)	Oddzielne projekty, pełna kontrola, najlepsza integracja. Dwukrotny nakład pracy developerskiej.

Dlaczego Dart? Czy nie można było użyć popularniejszego języka?

Dart był decyzją inżynierską wymuszoną przez architekturę Flutter. Dart jest kompilowany AOT (Ahead Of Time) do kodu maszynowego - to daje wydajność porównywalną z natywnym kodem. Języki interpretowane (JS, Python) nie dają takiej wydajności przy renderowaniu 60fps z własnym silnikiem.

Dart jest też kompilowany JIT (Just In Time) podczas developmentu - stąd Hot Reload działa błyskawicznie.

Po kilku godzinach z Dartem zauważysz, że jest bardzo podobny do Kotlin: null safety, extension functions, async/await, generyki, data class (przez freezed). Ktoś kto zna Kotlin lub Swift uczy się Darta w jeden dzień.

2. Instalacja i konfiguracja środowiska

Konfiguracja środowiska Flutter jest dłuższa niż instalacja Android Studio, ale wykonuje się ją tylko raz. Narzędzie flutter doctor precyzyjnie wskaże, czego brakuje. Sekcja ta prowadzi przez cały proces krok po kroku. Kluczowym elementem jest narzędzie diagnostyczne `flutter doctor`, które analizuje stan środowiska i identyfikuje brakujące komponenty. Proces konfiguracji ma charakter deterministyczny i jednorazowy, lecz wymaga poprawnego ustawienia zmiennych środowiskowych oraz zgodności wersji narzędzi.

2.1. Wymagania wstępne

Wymaganie	Wersja minimalna / uwagi
System operacyjny	Windows 10 (64-bit), macOS 12+, Ubuntu 20.04+. Na macOS wymagane Xcode dla iOS.
Flutter SDK	3.x (stable channel). Pobierz z docs.flutter.dev/get-started/install.
Android Studio	Giraffe (2022.3.1) lub nowszy. Wymagany do AVD i narzędzi Android SDK.
Android SDK	API level 21+ (Android 5.0). Flutter domyślnie targetuje minSdk 21.
Git	2.x - Flutter używa Git do zarządzania wersjami SDK i pakietów.
Dysk	Minimum 2.5 GB na Flutter SDK + 8 GB na Android SDK + emulator.

2.2. Instalacja Flutter SDK

Formalna procedura instalacji SDK jest procesem wieloetapowym: pobranie, ekstrakcja, konfiguracja PATH oraz walidacja. Istotnym ograniczeniem jest brak wsparcia dla ścieżek zawierających spacje lub znaki spoza ASCII, co wynika z ograniczeń narzędzi build systemu. Instalacja kończy się pozytywną walidacją w `flutter doctor`, co oznacza gotowość środowiska do kompilacji i uruchamiania aplikacji.

#	Proces instalacji Flutter SDK (Windows/Linux/macOS)
1	Pobierz archiwum Flutter SDK ze strony docs.flutter.dev/get-started/install. Wybierz odpowiedni system operacyjny i kanał stable.
2	Wypakuj archiwum do katalogu BEZ spacji i BEZ znaków specjalnych w ścieżce. Windows: C:\src\flutter (NIE: C:\Program Files\flutter). Linux/macOS: ~/development/flutter.
3	Dodaj flutter/bin do zmiennej środowiskowej PATH. Windows: Ustawienia → Zmienne środowiskowe → PATH. Linux/macOS: dodaj do ~/.bashrc lub ~/.zshrc: export PATH="\$HOME/development/flutter/bin:\$PATH".
4	Otwórz nowy terminal i uruchom: flutter doctor. Narzędzie sprawdzi wszystkie zależności i wypisze co brakuje.
5	Zainstaluj brakujące komponenty wskazane przez flutter doctor. Najczęściej: Android licenses (flutter doctor --android-licenses), Xcode na macOS (xcode-select --install).
6	Zainstaluj wtyczkę Flutter i Dart w Android Studio: File → Settings → Plugins → wyszukaj 'Flutter' → zainstaluj (Dart instaluje się automatycznie).
7	Uruchom flutter doctor ponownie - wszystkie checkmarki powinny być zielone (oprócz opcjonalnych platform jak Chrome czy Linux desktop).

Ścieżka instalacji NIE może zawierać spacji ani polskich znaków!

Flutter SDK w ścieżce ze spacjami lub znakami spoza ASCII (np. ą, ę, ó) powoduje trudne do zdiagnozowania błędy kompilacji.

BŁĘDNE lokalizacje: C:\Users\Jan Kowalski\flutter, /home/użytkownik/flutter

POPRAWNE lokalizacje: C:\dev\flutter, C:\src\flutter, /home/jan/dev/flutter

Ten problem dotyczy przede wszystkim Windows - konta użytkownika ze spacją w nazwie.

Jeśli Twój profil Windows ma spację, zainstaluj Flutter na C:\dev\flutter i ustaw PATH ręcznie.

2.3. Tworzenie pierwszego projektu

Opisuje inicjalizację projektu jako operację generowania struktury katalogów i plików konfiguracyjnych przez CLI (`flutter create`). Wskazuje na znaczenie parametrów takich jak identyfikator organizacji, platformy docelowe oraz typ szablonu. Podrozdział wprowadza również mechanizmy iteracyjnego developmentu: Hot

Reload (inkrementalna aktualizacja UI) oraz Hot Restart (rekonstrukcja stanu aplikacji), które skracają cykl edycyjny.

Tworzenie i uruchamianie projektu

```
# Tworzenie nowego projektu Flutter (w terminalu)
flutter create weather_app

# Parametry opcjonalne (zalecane dla nowych projektów):
flutter create \
  --org pl.edu.pam \           # Identyfikator organizacji (odwrócona domena)
  --project-name weather_app \ # Nazwa projektu (małe litery, podkreślniki)
  --platforms android,ios \    # Tylko wybrane platformy
  --template app \             # Szablon: app, plugin, package, skeleton
  weather_app                  # Katalog docelowy

# Uruchamianie aplikacji:
cd weather_app
flutter run                    # Na domyślnym urządzeniu (emulator lub fizyczne)
flutter run -d emulator-5554  # Na konkretnym emulatorze (ID z: flutter devices)

# Skrótów klawiszowe w terminalu podczas flutter run:
# r = Hot Reload (przeładuj UI bez restartu)   ← najważniejszy!
# R = Hot Restart (restart aplikacji)
# q = Quit (zakończ)
# h = Pomoc ze wszystkimi skrótami
```

Hot Reload vs Hot Restart; dwa tryby szybkiego developmentu

HOT RELOAD (r): Aktualizuje tylko zmieniony kod widżetów. Stan aplikacji (np. wpisany tekst, aktywna karta) jest ZACHOWANY. Działa w milisekundy. Używaj podczas pracy nad UI.

HOT RESTART (R): Restartuje aplikację od początku. Stan jest ZEROWANY (jak czyste uruchomienie). Szybszy niż pełny rebuild. Używaj gdy zmieniasz initState(), konstruktory klas, lub main().

PEŁNY BUILD (flutter run od nowa): Rekompiluje cały kod. Potrzebny po zmianach w pubspec.yaml (dodaniu paczek) lub po zmianach w kodzie natywnym (Android Manifest, Info.plist).

Dla porównania: w Kotlin/Compose masz tylko Hot Reload od Androida Studio. Flutterowy Hot Reload jest szybszy (kilkadziesiąt milisekund vs kilka sekund) i bardziej niezawodny.

3. Struktura projektu Flutter

Nowy projekt Flutter ma z góry określoną strukturę katalogów. Każdy katalog ma konkretną rolę - mylenie ich (np. wstawianie kodu Dart do folderu android/) to jeden z pierwszych błędów początkujących. Standardowy layout projektu jako strukturę modułową, w której separowane są: logika aplikacji (`lib/`), warstwa natywna (`android/`, `ios/`), zasoby (`assets/`) oraz konfiguracja (`pubspec.yaml`). Struktura ta wspiera skalowalność i zgodność z narzędziami build systemu Flutter.

3.1. Mapa katalogów

Mapa ta opisuje semantykę poszczególnych katalogów. Kluczowym elementem jest `lib/`, zawierający kod Dart i punkt wejścia aplikacji. Katalogi platformowe pełnią rolę integracyjną i są modyfikowane jedynie w specyficznych przypadkach (np. uprawnienia systemowe). Struktura ta wymusza separację warstw i minimalizuje ryzyko błędów architektonicznych.

Struktura projektu weather_app/

```
weather_app/
```

lib/	← TUTAJ PISZESZ KOD DART (główny folder!)
main.dart	← Punkt wejścia aplikacji - funkcja main()
test/	← Testy jednostkowe (Dart, bez emulatora)
widget_test.dart	← Przykładowy test widżetu
android/	← Kod natywny Android (XML, Kotlin) - rzadko edytujesz
app/src/main/	
AndroidManifest.xml	← Uprawnienia: INTERNET tutaj!
kotlin/.../	← MainActivity.kt - pusta, Flutter ją przejmie
build.gradle	← Konfiguracja Gradle (minSdk, targetSdk)
ios/	← Kod natywny iOS - edytujesz przez Xcode
web/	← Pliki HTML/JS dla wersji webowej
assets/	← Obrazy, fonty, pliki JSON (stwórz ręcznie!)
pubspec.yaml	← MANIFEST PROJEKTU - paczki, assets, wersja
pubspec.lock	← Zamrożone wersje paczek (commituj do Git!)
analysis_options.yaml	← Konfiguracja lintera Dart

3.2. Plik pubspec.yaml; manifest projektu

Plik `pubspec.yaml` pełni funkcję manifestu zależności oraz konfiguracji zasobów projektu. Rozróżnia zależności `runtime` (`dependencies`) i `developerskie` (`dev_dependencies`). Definiuje również wersjonowanie semantyczne oraz zakres kompatybilności SDK. Sekcja `flutter` umożliwia deklaratywne dołączanie zasobów statycznych, co jest wymagane do ich bundlowania w procesie build.

kluczowe sekcje pubspec.yaml	Wyjaśnienie
<code>name: weather_app</code>	Nazwa paczki (małe litery, podkreślniki). Używana jako identyfikator.
<code>description: Aplikacja pogodowa</code>	Opis projektu - widoczny w pub.dev jeśli publikujesz.
<code>publish_to: 'none'</code>	Zapobiega przypadkowemu opublikowaniu na pub.dev.
<code>version: 1.0.0+1</code>	Wersja: major.minor.patch+buildNumber. +1 = versionCode na Androidzie.
<code>environment:</code>	
<code>sdk: '>=3.0.0 <4.0.0'</code>	Minimalna wersja Dart SDK. Zawsze ustaw zakres, nie samą dolną granicę.
<code>dependencies:</code>	Paczki RUNTIME - używane w działającej aplikacji.
<code>flutter:</code>	Obowiązkowa zależność na framework Flutter.
<code>sdk: flutter</code>	<code>sdk: flutter</code> = paczka wbudowana, nie z pub.dev.
<code>http: ^1.2.1</code>	Paczka http z pub.dev. ^ = dowolna kompatybilna wersja (1.x.x).
<code>dev_dependencies:</code>	Paczki DEVELOPERSKIE - tylko podczas kompilacji i testów.
<code>flutter_test:</code>	Biblioteka do testów widżetów - wbudowana.
<code>sdk: flutter</code>	
<code>flutter:</code>	Sekcja konfiguracji Flutter (assets, fonts).
<code>uses-material-design: true</code>	Włącza ikony Material Design (wymagane dla <code>Icons.xxx</code>).
<code>assets:</code>	Lista zasobów statycznych do dołączenia do APK.
- <code>assets/images/</code>	Cały folder images/ (ze / na końcu = cały katalog).
- <code>assets/data/cities.json</code>	Konkretny plik JSON.

4. Dart - podstawy języka

Dart jest silnie typowanym, obiektowym językiem z null safety. Jeśli znasz Kotlin lub Swift, Dart będzie znajomy, a różnice są kosmetyczne (składnia i semantyka są zbliżone). W tej sekcji skupiamy się na elementach, które będą używane w projekcie WeatherApp. Dart jako język jest silnie typowany z mechanizmem null safety. Podkreśla to jego dualny model kompilacji (AOT/JIT), który umożliwia zarówno wysoką wydajność produkcyjną, jak i szybki cykl developmentu.

4.1. Typy, zmienne i null safety

Podrozdział ten formalizuje system typów w Darcie, obejmujący inferencję typów (`var`) oraz deklaracje jawne. Kluczowym mechanizmem jest null safety, który statycznie wymusza obsługę wartości opcjonalnych (`?`).

Wprowadza również rozróżnienie między `final` (inicjalizacja jednokrotna w runtime) a `const` (stała kompilacyjna), co ma implikacje wydajnościowe i pamięciowe.

Dart - typy i zmienne	Wyjaśnienie
// Inferencja typów - Dart sam wykrywa typ var name = 'Warszawa'; // String var temp = 22.5; // double var isRaining = false; // bool	Podobnie jak val w Kotlinie z automatyczną inferencją var = typ jest wnioskowany, ale niezmienny po ustaleniu Zmiennoprzecinkowy - domyślnie double w Darcie Booleanowy - zawsze bool (nie Boolean jak w Javie)
// Jawna deklaracja typów String city = 'Kraków'; int humidity = 65; // całkowity double windSpeed = 12.4;	Preferowana w publicznym API i parametrach funkcji int w Darcie jest 64-bitowy na VM
// NULL SAFETY - kluczowa cecha Dart 2.12+ String nonNull = 'zawsze ma wartość'; String? nullable = null; // może być null	Podobne do Kotliny: ? = nullable, bez ? = non-null Non-nullable - kompilator WYMUSI inicjalizację Nullable - musisz sprawdzić null przed użyciem
// final = przypisz raz (runtime), const = stała kompilacji final city2 = 'Gdańsk'; // jak val w Kotlinie const pi = 3.14159; // optymalizacja! const colors = [1, 2, 3]; // lista niemutowalna	final: wartość ustalona w runtime const: wbudowane w binarny kod - szybsze const kolekcje są niemutowalne i dzielone w pamięci

4.2. Funkcje i parametry

Model funkcji w Darcie można zdefiniować jako elementy pierwszej klasy z jawnym typem zwracanym. Wprowadza on dwa podstawowe tryby przekazywania parametrów: pozycyjne oraz nazwane (z użyciem nawiasów klamrowych), co zwiększa czytelność API i redukuje błędy wywołań. Mechanizm `required` wymusza obecność parametrów nazwanych. Skrócona składnia strzałkowa (`=>`) umożliwia definiowanie funkcji jednowyrażeniowych. Istotnym elementem jest wsparcie dla programowania asynchronicznego poprzez `Future` oraz konstrukcje `async/await`, które implementują model współbieżności bez blokowania wątku UI.

Dart - funkcje	Wyjaśnienie
// Podstawowa funkcja z typami String formatTemp(double celsius) { return '\${celsius.toStringAsFixed(1)}°C'; }	Dart wymaga jawnego typu zwracanego lub void Parametr pozycyjny - wymagany, bez nazwy przy wywołaniu \${expr} = interpolacja stringa. Zawsze cudzysłów lub apostrof.
// Nazwane parametry (curly braces) Widget buildCard({ required String title, // obowiązkowy String subtitle = '', // opcjonalny bool isHighlighted = false, }) { ... }	Jak w Kotlinie: fun(name: String = ...) { } = parametry nazwane. Wywołanie: buildCard(title: 'x') required = brak wartości domyślnej. Musi być podany. Wartość domyślna - nie musisz podawać przy wywołaniu
// Arrow function (=>) - jednoliniowe funkcje double toCelsius(double f) => (f - 32) * 5 / 9;	=> expr jest skrótem dla { return expr; } Identyczne z: { return (f - 32) * 5 / 9; }

Dart - funkcje	Wyjaśnienie
// Async / await - identyczne jak w Kotlinie	
Future<String> fetchCity() async {	Future<T> = odpowiednik Kotlin suspend fun
final data = await apiCall(); // czeka!	await wstrzymuje bez blokowania wątku UI
return data.cityName;	
}	

4.3. Klasy i konstruktory w Darcie

Sformalizujmy model obiektowy języka. Klasy w Darcie są domyślnie publiczne, a enkapsulacja realizowana jest konwencją nazewniczą (`_` dla prywatności na poziomie biblioteki). Konstruktor z parametrami nazwanymi upraszcza inicjalizację obiektów i wspiera niemutowalność poprzez pola `final`. Wprowadzone są konstruktory `const`, które umożliwiają alokację obiektów w czasie kompilacji. Wzorzec `factory` pozwala na kontrolę procesu tworzenia instancji, szczególnie w kontekście deserializacji (np. JSON). Metoda `copyWith` implementuje idiom niemutowalnych modyfikacji obiektów.

Dart - klasy	Wyjaśnienie
class WeatherData {	Klasa Dart - domyślnie publiczna
final String city; // pole final = readonly	Brak private/public na polach - użyj <code>_</code> dla prywatnych
final double temp;	
final int humidity;	
final String? iconCode; // nullable!	Może być null jeśli API nie zwróci ikony
// Konstruktor nazwany z inicjalizacją pól (this.x)	this.x = skrót dla: this.city = city;
const WeatherData({	const constructor = obiekty WeatherData mogą być const
required this.city,	required = parametr obowiązkowy (brak wartości domyślnej)
required this.temp,	
required this.humidity,	
this.iconCode, // opcjonalny nullable	Brak required = opcjonalny. Domyślnie null.
});	
// Fabryka z JSON - typowy wzorzec dla API	
factory WeatherData.fromJson(Map<String, dynamic> json) {	factory = zwraca istniejący lub nowy obiekt
return WeatherData(	
city: json['name'] as String,	as String = rzutowanie z dynamic
temp: (json['main']['temp'] as num).toDouble(),	num = Int lub double; toDouble() bezpieczne
humidity: json['main']['humidity'] as int,	
);	
}	
// copyWith - niezmiennosc + modyfikacja	Dart nie generuje copyWith automatycznie (użyj freezed)
WeatherData copyWith({double? temp}) =>	=> arrow function, zwraca nowy obiekt
WeatherData(city: city, temp: temp ?? this.temp,	?? = jeśli null, użyj this.temp
humidity: humidity);	
}	

4.4. Kolekcje i operatory

Dart opisuje takie (podstawowe) struktury danych jak listy (List) i mapy (Map) oraz operacje wyższego rzędu (map, where). Wprowadza operator kaskadowy (`..`), umożliwiający sekwencyjne wywołania metod na tym

samym obiekcie. Istotnym elementem są operatory null-aware (`?.`, `??`, `??=`), które stanowią rozszerzenie mechanizmu null safety. Dart wspiera również deklaratywne budowanie kolekcji poprzez spread operator (`...`) oraz warunkowe elementy w listach, co jest szczególnie użyteczne w konstrukcji UI.

Kolekcje i operatory Dart

```
// Lista (List) - dynamicznie typowana
final List<String> cities = ['Warszawa', 'Kraków', 'Gdańsk'];
cities.add('Wrocław'); // dodaj element
final sorted = cities.sort(); // .. = cascade operator (sortuj i zwróć listę)
final upper = cities.map((c) => c.toUpperCase()).toList(); // map jak w Kotlinie
final wawList = cities.where((c) => c.startsWith('W')).toList(); // filter

// Mapa (Map) - odpowiednik HashMap
final Map<String, double> temps = {'Warszawa': 22.5, 'Kraków': 19.0};
final waw = temps['Warszawa']; // double? - może być null!
temps['Gdańsk'] = 18.5; // dodaj/nadpisz

// Spread operator (...) - scalanie list
final all = [...cities, 'Poznań', 'Łódź']; // nowa lista z rozpakowaniem

// Conditional elements w listach (if/for wewnątrz listy)
final widgets = [
  Text('Temperatura'),
  if (isRaining) const Icon(Icons.umbrella), // warunkowy element
  for (final c in cities) Text(c), // pętla w liście!
];

// Null-aware operators - analogiczne do Kotliny
String? maybeNull = null;
final length = maybeNull?.length; // ?. = null-safe call (jak ?. w Kotlinie)
final safe = maybeNull ?? 'domyślna'; // ?? = Elvis operator
maybeNull ??= 'przypisz jeśli null'; // ??= = przypisz tylko jeśli null
```

5. Widżety, fundamenty UI we Flutterze

Jeśli Jetpack Compose opiera się na funkcjach `@Composable`, to Flutter opiera się na klasach `Widget`. Wszystko we Flutterze jest widżetem: padding, kolumna, tekst, obraz, nawet aplikacja. To trochę jak klocki LEGO - z gotowych, małych elementów budujesz złożone UI. Model UI ma charakter deklaracyjny i opiera się na niemutowalnych strukturach danych (drzewo widżetów). Każda zmiana stanu skutkuje rekonstrukcją drzewa i jego różnicową aktualizacją (diffing), co minimalizuje koszt renderowania. Architektura ta jest analogiczna do podejścia Virtual DOM, lecz zoptymalizowana pod kątem bezpośredniego renderowania.

Analogia: drzewo widżetów jak drzewo genealogiczne

Wyobraź sobie drzewo rodzinne. Każdy widżet to osoba - ma rodziców i może mieć dzieci.

MaterialApp (pradziadek) → Scaffold (dziadek) → Column (rodzic) → [Text, Button, Image] (dzieci)

Każdy widżet wie tylko o swoich dzieciach. Rodzic mówi dziecku: 'masz do dyspozycji tyle miejsca'.

Dziecko odpowiada: 'zajmę tyle'. Rodzic rozmieszcza dzieci zgodnie z ich rozmiarami.

Ta hierarchia jest NIEZMIENNA (immutable). Gdy coś się zmienia, Flutter tworzy NOWE drzewo widżetów i porównuje z poprzednim - renderuje tylko różnice (podobnie jak Virtual DOM w React).

To klucz do wydajności Flutter: tworzenie obiektów `Widget` jest tanie, bo są immutable.

Ciężka praca (układ i rendering) odbywa się tylko raz, przy zmianach.

5.1. StatelessWidget, czyli widget bez stanu

Komponenty deterministyczne to takie, których wynik renderowania zależy wyłącznie od przekazanych parametrów. StatelessWidget jest niemutowalny, a jego metoda `build()` jest funkcją czystą względem wejścia. Wzorec ten sprzyja testowalności i optymalizacjom (np. użycie `const`). Komunikacja z otoczeniem odbywa się przez parametry oraz callbacki.

WeatherCard.dart - StatelessWidget	Wyjaśnienie
<code>class WeatherCard extends StatelessWidget {</code>	<code>extends StatelessWidget</code> = ten widget nigdy nie zmienia stanu
<code>final WeatherData weather; // dane z zewnątrz</code>	Dane przekazywane przez konstruktor - immutable
<code>final VoidCallback? onTap; // callback opcjonalny</code>	<code>VoidCallback</code> = <code>void Function()</code> - callback bez argumentów
<code>const WeatherCard({ // const constructor!</code>	<code>const</code> = Flutter może optymalizować: nie tworzy jeśli dane te same
<code>super.key, // przekaz key do rodzica</code>	Key pomaga Flutter śledzić widżety w listach - zawsze podawaj
<code>required this.weather,</code>	
<code>this.onTap,</code>	
<code>});</code>	
<code>@override</code>	<code>@override</code> = implementujesz metodę abstrakcyjną z StatelessWidget
<code>Widget build(BuildContext context) {</code>	<code>build()</code> = Flutter woła ją kiedy widget ma się narysować
<code>return Card(// Card=Material widget z cieniem</code>	Card to gotowy widget z Material Design - cień, zaokrąglenia
<code>child: InkWell(// Reaguje na dotyk z ripple effect</code>	InkWell dodaje efekt fali (ripple) przy tapnięciu
<code>onTap: onTap,</code>	Przekazujemy callback - null = wyłącz tapowanie
<code>child: Padding(// Padding wewnątrz karty</code>	Padding to WIDGET - nie property jak w Androidzie
<code>padding: const EdgeInsets.all(16),</code>	<code>EdgeInsets.all(16)</code> = 16dp ze wszystkich stron
<code>child: Column(// Kolumna potomków pionowo</code>	Column = LinearLayout pionowy
<code>children: [</code>	children = lista dzieci widgetu
<code>Text(weather.city,</code>	Text wyświetla string
<code>style:</code>	
<code>Theme.of(context).textTheme.headlineMedium,</code>	<code>Theme.of(context)</code> = styl z tematu aplikacji
<code>),</code>	
<code>Text('\${weather.temp.toStringAsFixed(1)}°</code>	Interpolacja stringa w Dart: <code>{wyrażenie}</code>
<code>C')',</code>	
<code>],</code>	
<code>),</code>	
<code>),</code>	
<code>),</code>	
<code>);</code>	
<code>}</code>	
<code>}</code>	

5.2. StatefulWidget, czyli widżet ze stanem

Gdy widget musi pamiętać jakąś wartość i zmieniać ją w czasie (np. licznik, pole tekstowe, lista po załadowaniu danych), potrzebujemy StatefulWidget. Składa się z dwóch klas: widgetu (immutable, konfiguracja) i stanu (mutable, dane). Widżet ten jest przykładem modelu komponentów stanowych, w którym rozdzielono konfigurację (StatefulWidget) od mutowalnego stanu (State). Cykl życia stanu obejmuje inicjalizację (`initState`), aktualizację (`setState`) oraz renderowanie (`build`). Kluczową operacją jest `setState`, która

sygnalizuje konieczność rekonstrukcji UI. Mechanizm ten zapewnia reaktywność przy zachowaniu kontroli nad zakresem aktualizacji.

Dlaczego dwie klasy dla StatefulWidget?

StatefulWidget MUSI być immutable - jak każdy widget. Ale stan jest mutable. Jak to pogodzić?

Flutter rozdziela te odpowiedzialności na dwie klasy:

1. WIDGET (np. `WeatherScreen`): Tylko konfiguracja i klucz. Immutable. Może być odtworzony wiele razy.
2. STATE (np. `_WeatherScreenState`): Przechowuje mutable dane. Przeżywa odtworzenia widżetu.

Gdy Flutter odtwarza widget (np. przy obroceniu ekranu), STATE jest zachowany!

Podkreślnik (podłoga) `_` = prywatność w Darcie. `_WeatherScreenState` jest prywatny dla biblioteki.

To nie jest konwencja - to celowy sygnał: 'ta klasa to implementacja, nie API publiczne'.

WeatherScreen.dart - StatefulWidget	Wyjaśnienie
<pre>class WeatherScreen extends StatefulWidget { const WeatherScreen({super.key}); @override State<WeatherScreen> createState() => _WeatherScreenState(); }</pre>	Widget (immutable) - tylko konfiguracja Przekazuj <code>super.key</code> zawsze - pomaga Flutter śledzić widget
<pre>class _WeatherScreenState extends State<WeatherScreen> { WeatherData? _weather; // null = jeszcze nie załadowano bool _isLoading = false; // spinner kontrolka String? _error; // komunikat błędu @override void initState() { // wywoływane raz, przy tworzeniu stanu super.initState(); // ZAWSZE wywołaj super.initState() pierwsza! } _loadWeather(); // załaduj dane po inicjalizacji } Future<void> _loadWeather() async { setState(() => _isLoading = true); // zaktualizuj UI try { final data = await WeatherService().fetch('Warszawa'); setState(() { // WSZYSTKIE zmiany stanu wewnątrz setState! _weather = data; _isLoading = false; }); } catch (e) { setState(() { _error = e.toString(); _isLoading = false; }); } }</pre>	<code>State<T></code> gdzie <code>T</code> = klasa widgetu Stan: mutable, może zmieniać się w czasie <code>initState</code> = odpowiednik <code>LaunchedEffect(Unit)</code> z Compose Brak <code>super.initState()</code> = crash w runtime Wołamy metodę, która uruchomi pobieranie danych <code>async</code> = ta funkcja może używać <code>await</code> <code>setState()</code> = powiedz Flutterowi: przerysuj widget <code>await</code> = zaczekaj na wynik bez blokowania UI <code>setState(() { ... })</code> = atomic update UI
<pre>@override Widget build(BuildContext context) { if (_isLoading) return const CircularProgressIndicator(); if (_error != null) return Text('Błąd: \$_error');</pre>	<code>build()</code> = rysuj na podstawie AKTUALNEGO stanu Stan: ładowanie → spinner Stan: błąd → komunikat

WeatherScreen.dart - StatefulWidget	Wyjaśnienie
<pre>if (_weather == null) return const Text('Brak danych'); return WeatherCard(weather: _weather!); }</pre>	Stan: pusty → placeholder Stan: sukces → karta pogody. ! = non-null assertion

5.3. Podstawowe widżety układu; cheat sheet

Rzućmy teraz okiem na zestaw prymitywów layoutu, takich jak Column, Row, Stack, Container czy ListView. Widgety te implementują model kompozycji przestrzennej oparty na constraints (ograniczeniach rozmiaru przekazywanych w dół drzewa). Zrozumienie propagacji constraints i rozmiarów jest kluczowe dla poprawnego projektowania UI. Wprowadzone są również komponenty strukturalne (Scaffold, AppBar) oraz kontrolki wejścia (TextField, Button).

Widget	Opis i odpowiednik w Android/Compose
Column / Row	Układ pionowy / poziomy. Odpowiednik: Column / Row w Compose. Właściwości: mainAxisAlignment, crossAxisAlignment.
Stack	Nakładanie widgetów na siebie (z-index). Odpowiednik: Box w Compose. Użyj Positioned dla dokładnego pozycjonowania.
Container	Wielofunkcyjny kontener: rozmiar, kolor tła, padding, margin, border, gradient. Odpowiednik: Modifier w Compose (ale to widget).
SizedBox	Stały rozmiar lub odstęp. SizedBox(height: 16) = Spacer o stałym rozmiarze. SizedBox.expand() = wypełnij dostępne miejsce.
Padding	Dodaje padding wokół child. Preferowany nad Container gdy tylko padding jest potrzebny - lżejszy widget.
Expanded / Flexible	Rozciągnij widget w Column/Row. Expanded = zajmij cały dostępny obszar. Flexible = zajmij proporcjonalnie (flex).
ListView / GridView	Przewijalne listy. ListView.builder = leniwa lista (jak LazyColumn). GridView.builder = siatka (jak LazyVerticalGrid).
Scaffold	Szkielet ekranu: AppBar, body, bottomNavigationBar, floatingActionButton, drawer. Obowiązkowy na każdym ekranie.
AppBar	Pasek nawigacyjny na górze. title, actions (ikony po prawej), leading (ikona po lewej - auto Back button).
ElevatedButton / TextButton / OutlinedButton	Przyciski Material 3. ElevatedButton = wyróżniony, TextButton = płaski, OutlinedButton = z ramką.
TextField	Pole tekstowe. Kontrolowane przez TextEditingController. Odpowiednik: BasicTextField w Compose.
Image / Image.network / Image.asset	Obrazy: z URL (Image.network), z assets (Image.asset). Brak automatycznego cache - użyj cached_network_image.

6. Zarządzanie stanem: od setState do Provider

Przeanalizujmy problem skalowalności zarządzania stanem. setState() działa świetnie dla lokalnego stanu jednego widżetu. Ale gdy wiele widżetów potrzebuje tych samych danych (np. lista ulubionych miast wyświetlana na kilku ekranach), potrzebujemy globalnego zarządzania stanem. setState jest efektywny lokalnie, lecz prowadzi do problemu propagacji danych (prop drilling) w większych aplikacjach. Wprowadzenie wzorców globalnego stanu, takich jak Provider, umożliwia separację logiki biznesowej od warstwy prezentacji oraz centralizację danych. W tym ćwiczeniu użyjemy Provider - najprostszego i oficjalnie zalecanego rozwiązania dla początkujących.

6.1. Problem z setState() na dużą skalę

Ograniczenia podejścia lokalnego są następujące: silne sprzężenie komponentów, trudność w reużywalności oraz brak separacji odpowiedzialności. Wraz ze wzrostem złożoności aplikacji rośnie liczba zależności między widgetami, co utrudnia utrzymanie i testowanie. Problem ten uzasadnia zastosowanie wzorców zarządzania stanem opartych na obserwowalności i iniekcji zależności.

Analogia: szeptana wiadomość przez rząd dzieci

Wyobraź sobie klasę szkolną ustawioną w rząd. Chcesz przekazać informację z lewego końca do prawego. Z setState() musisz szeptać wiadomość przez każde dziecko po drodze - prop drilling.

Widget A (App) ma dane → przekazuje do Widget B → B do C → C do D → D w końcu wyświetla. Każde pośrednie B, C musi 'wiedzieć' o danych mimo że ich nie używa - tylko przekazuje dalej.

PROVIDER rozwiązuje to jak tablica ogłoszeń w klasie: A wypisuje wiadomość, D bezpośrednio odczytuje - bez pośredników. B i C nie muszą wiedzieć nic.

Provider to 'InheritedWidget z uprzejmą twarzą' - mechanizm, który Flutter ma wbudowany, ale trudny w użyciu wprost. Provider opakowuje go w czytelne API.

6.2. Konfiguracja Provider

Provider implementuje wzorec dependency injection połączony z mechanizmem obserwowalności. Dane współdzielone są udostępniane w drzewie widgetów poprzez kontekst (BuildContext), co eliminuje konieczność ręcznego przekazywania ich przez kolejne warstwy komponentów. Kluczową rolę pełni ChangeNotifier, który umożliwia emisję sygnału o zmianie stanu (notifyListeners). Widgety konsumujące dane mogą reagować selektywnie na zmiany, co minimalizuje zakres przebudowy UI. Istotne jest rozróżnienie między odczytem reaktywnym (watch) a niereaktywnym (read), co wpływa na wydajność.

Instalacja Provider

```
# pubspec.yaml - dodaj zależność
dependencies:
  provider: ^6.1.2

# Po dodaniu wykonaj:
flutter pub get # Pobierz nowe paczki (jak gradle sync)
```

6.3. ChangeNotifier - model danych powiadamiający widgety

Warstwa prezentacji (UI) powinna być odseparowana od logiki biznesowej i dostępu do danych. Provider umożliwia wprowadzenie pośredniej warstwy modelu widoku (ViewModel), który agreguje dane i operacje. UI jedynie obserwuje stan i renderuje jego reprezentację. Logika biznesowa (np. pobieranie danych z API) jest enkapsulowana w serwisach. Taki podział redukuje sprzężenie, zwiększa testowalność oraz umożliwia równoległy rozwój komponentów.

weather_provider.dart - ChangeNotifier	Wyjaśnienie
<code>class WeatherProvider extends ChangeNotifier {</code>	ChangeNotifier = klasa z mechanizmem powiadamiania
<code> WeatherData? _weather; // _ = prywatne</code>	Prywatne pole - zmień tylko przez metody publiczne
<code> bool _isLoading = false;</code>	
<code> String? _error;</code>	
<code> // Gettery publiczne - widgety przez nie czytają</code>	Udostępniamy tylko do odczytu (immutable z zewnątrz)
<code> WeatherData? get weather => _weather;</code>	get weather = właściwość computed (jak val w Kotlinie)
<code> bool get isLoading => _isLoading;</code>	
<code> String? get error => _error;</code>	

weather_provider.dart - ChangeNotifier	Wyjaśnienie
<code>// Metoda zmieniająca stan</code>	
<code>Future<void> loadWeather(String city) async {</code>	Publiczna metoda - widgety mogą ją wywołać
<code> _isLoading = true;</code>	Zmień prywatne pole...
<code> notifyListeners(); // POWIADOM widgety!</code>	notifyListeners() = odpowiednik setState() dla Provider
<code> try {</code>	
<code> _weather = await WeatherService().fetch(city);</code>	await = asynchroniczne pobieranie
<code> _error = null;</code>	
<code> } catch (e) {</code>	
<code> _error = e.toString();</code>	
<code> } finally {</code>	
<code> _isLoading = false;</code>	
<code> notifyListeners(); // Powiadom po zakończeniu</code>	Flutter przerysuje wszystkie nasłuchujące widżety
<code> }</code>	
<code>}</code>	
<code>}</code>	

6.4. Rejestracja i odczyt Provider

Provider - rejestracja i odczyt (trzy metody)
<pre>// main.dart - rejestracja na szczycie drzewa widgetów void main() { runApp(// ChangeNotifierProvider MUSI być powyżej widgetów które go używają ChangeNotifierProvider(create: (_) => WeatherProvider(), // Utwórz instancję (raz na całą aplikację) child: const MyApp(),),); } // W widżecie konsumującym - trzy sposoby dostępu: // 1. Consumer<T> - rebuilda TYLKO swoje children (najwydajniejszy) Consumer<WeatherProvider>(builder: (context, provider, child) { // child = część NIE przebudowywana if (provider.isLoading) return const CircularProgressIndicator(); return WeatherCard(weather: provider.weather!); },) // 2. context.watch<T>() - rebuilda cały widget (prostszy, ale mniej wydajny) final provider = context.watch<WeatherProvider>(); // Używaj w build() - Widget odbuduje się przy każdej zmianie WeatherProvider // 3. context.read<T>() - jednorazowy odczyt BEZ nasłuchiwanie (dla akcji/callbacków) ElevatedButton(onPressed: () => context.read<WeatherProvider>().loadWeather('Kraków'), child: const Text('Załaduj'),) // UWAGA: NIE używaj context.read() w build() - nie odświeży UI przy zmianie!</pre>

context.watch() tylko w **build()**, a nie w **initState()** ani **callbackach**!

context.watch<T>() subskrybuje się na zmiany i triggeruje rebuild. Wywołanie go poza metodą build() (np. w initState(), onPressed, onChanged) spowoduje wyjątek:

'Bad state: Tried to listen to a value exposed with provider...'

ZASADA: `context.watch()` = tylko w `build()` (do odczytu i wyświetlenia)

ZASADA: `context.read()` = w callbackach (`onPressed`, `initState`) - gdy chcesz wywołać metodę

Pomocna mnemotechnika: `watch` = 'obserwuję ekran i reaguję', `read` = 'jednorazowe działanie'.

7. Nawigacja - Navigator 2.0 i GoRouter

Nawigacja opiera się na modelu deklaratywnym, w którym stan aplikacji determinuje aktualny stos ekranów. Flutter ma dwa systemy nawigacji. Stary Navigator 1.0 (`push/pop`) jest prosty dla małych aplikacji. Navigator 2.0 wprowadza kontrolę nad routinguem poprzez zarządzanie listą stron (*Pages*), zamiast imperatywnych wywołań `push/pop`. GoRouter stanowi warstwę abstrakcji upraszczającą konfigurację tras, obsługę parametrów oraz deep linking. Podejście to umożliwia synchronizację URL (szczególnie dla web) oraz deterministyczne odtwarzanie stanu aplikacji. GoRouter to oficjalnie zalecane rozwiązanie dla aplikacji produkcyjnych. Jest oparty na URL-ach, wspiera deep linking i działa tak samo na wszystkich platformach.

7.1. Navigator 1.0 - push i pop

Model stosu ekranów implementuje klasyczny mechanizm LIFO. Operacja `push` dodaje nowy ekran na stos, natomiast `pop` usuwa bieżący i przywraca poprzedni. Każdy ekran funkcjonuje jako odrębny kontekst UI. Parametry mogą być przekazywane w trakcie nawigacji, co umożliwia komunikację między ekranami. Mechanizm ten jest prosty, lecz w większych aplikacjach prowadzi do trudności w zarządzaniu stanem nawigacji.

Podstawowa nawigacja Navigator	Wyjaśnienie
<pre>// Przejście do nowego ekranu (push) Navigator.of(context).push(MaterialPageRoute(builder: (ctx) => DetailScreen(weatherData: selectedWeather, // przekaż dane),),);</pre>	<code>Navigator.of(context)</code> = znajdź Navigator w drzewie <code>MaterialPageRoute</code> = ekran z animacją Material (slide z prawej) <code>builder</code> = funkcja tworząca nowy ekran Dane przekazywane PRZEZ KONSTRUKTOR - nie przez URL
<pre>// Powrót do poprzedniego ekranu (pop) Navigator.of(context).pop(); Navigator.of(context).pop(result);</pre>	<code>pop</code> = zdejmij ekran ze stosu. Jak Back button. <code>pop</code> z wartością = przekaż dane do poprzedniego ekranu
<pre>// Odebranie wartości z powracającego ekranu final result = await Navigator.push<String>(context, MaterialPageRoute(builder: (_) => SearchScreen()),); // result = String?; Co przekażał SearchScreen przez pop(result) if (result != null) useResult(result);</pre>	<code>await push</code> = czekaj na powrót z ekranu

7.2. GoRouter - nawigacja dla większych aplikacji

Routing deklaratywny definiuje mapowanie między stanem aplikacji a strukturą tras. Parametry ścieżki (np. `/weather/:city`) umożliwiają dynamiczne generowanie widoków. System parsuje URL i rekonstruuje odpowiedni ekran wraz z danymi wejściowymi. Podejście to zwiększa spójność aplikacji, ułatwia obsługę deep linków oraz wspiera testowanie scenariuszy nawigacyjnych.

Instalacja GoRouter

```
# pubspec.yaml
dependencies:
  go_router: ^14.2.7
```

router.dart - konfiguracja GoRouter	Wyjaśnienie
<code>final appRouter = GoRouter(</code>	GoRouter = singleton konfiguracji routera
<code> initialLocation: '/', // startowy ekran</code>	URL startowy - jak trasa domyślna w web routerze
<code> routes: [</code>	Lista zdefiniowanych tras
<code> GoRoute(</code>	Jedna trasa = jeden ekran lub grupa
<code> path: '/', // URL trasy</code>	Trasa główna - lista miast
<code> name: 'home', // Nazwa symboliczna</code>	Użyj nazwy zamiast stringa URL - bezpieczniejsze
<code> builder: (ctx, state) => const HomeScreen(),</code>	builder tworzy widżet dla tej trasy
<code> routes: [// Zagnieżdżone trasy</code>	Dziecko inherituje ścieżkę rodzica: /detail/:city
<code> GoRoute(</code>	
<code> path: 'detail/:city', // :city = parametr</code>	Dwukropek = parametr dynamiczny w URL
<code> name: 'detail',</code>	
<code> builder: (ctx, state) {</code>	
<code> final city = state.pathParameters['city']!</code>	Odczyt parametru z URL. ! = zakładamy że istnieje
<code> return DetailScreen(city: city);</code>	
<code> },</code>	
<code>],</code>	
<code>),</code>	
<code>],</code>	
<code>);</code>	
<code>// Nawigacja przy użyciu GoRouter</code>	
<code>context.go('/');</code>	<code>go()</code> = jak <code>pushReplacement</code> , czyści stos
<code>context.push('/detail/\$city');</code>	<code>push()</code> = jak <code>Navigator.push</code> - stos rośnie
<code>context.goNamed('detail',</code>	Bezpieczniej niż string - refactoring nie psuje
<code> pathParameters: {'city': 'Gdansk'})</code>	
<code>context.pop();</code>	<code>pop()</code> jak <code>Navigator.pop</code>

context.go() vs context.push() - kluczowa różnica!

`context.go('/home')` = ZASTĘPUJE cały stos nawigacji. Użytkownik nie może wrócić Back.
Używaj gdy: logujesz użytkownika i chcesz usunąć ekran logowania ze stosu.

`context.push('/detail/pikachu')` = DODAJE ekran na stos. Użytkownik może wrócić Back.
Używaj gdy: otwierasz widok szczegółów i chcesz normalnego powrotu.

Błąd: użycie `go()` zamiast `push()` powoduje, że przycisk Back znika lub wychodzi z aplikacji.

8. Pobieranie danych z REST API

WeatherApp będzie pobierać dane z Open-Meteo, darmowego, publicznego API pogodowego, które nie wymaga klucza API. Użyjemy pakietu `http` (oficjalny pakiet Dart) do wykonywania żądań HTTP. Komunikacja z API opiera się na protokole HTTP oraz serializacji danych w formacie JSON. Operacje są asynchroniczne i zwracają obiekty typu `Future`. Kluczowe etapy obejmują: wysłanie zapytania, walidację odpowiedzi (status code), deserializację oraz mapowanie na modele domenowe. Niezbędna jest obsługa błędów sieciowych i wyjątków. Warstwa dostępu do danych powinna być izolowana od UI.

8.1. Przegląd Open-Meteo API

Pakiet `http` dostarcza funkcje do wykonywania zapytań sieciowych. Zapytanie GET jest najczęściej używane do pobierania danych. Odpowiedź zawiera kod statusu oraz ciało (body), które wymaga parsowania. Operacje te powinny być wykonywane w sposób asynchroniczny, aby nie blokować wątku UI. Ważna jest walidacja odpowiedzi przed dalszym przetwarzaniem.

Endpoint	Opis i przykładowy URL
Geolokalizacja miast	https://geocoding-api.open-meteo.com/v1/search?name=Warszawa&count=1&language=pl - zwraca współrzędne dla nazwy miasta.
Dane pogodowe	https://api.open-meteo.com/v1/forecast?latitude=52.23&longitude=21.01&current=temperature_2m,wind_speed_10m,relative_humidity_2m,weather_code - aktualna pogoda
Prognoza godzinowa	Parametr: <code>&hourly=temperature_2m</code> - dane co godzinę na 7 dni. Duża odpowiedź JSON.
Prognoza dzienna	Parametr: <code>&daily=temperature_2m_max,temperature_2m_min</code> - dane dzienne na 7 dni.

8.2. Uprawnienie INTERNET, czyli krok którego nie wolno pominąć

JSON jest mapowany na struktury Dart poprzez konwersję do `Map<String, dynamic>`. Następnie dane są rzutowane na konkretne typy i inicjalizują obiekty domenowe. Wzorec `fromJson` standaryzuje proces deserializacji. Należy uwzględnić niekompletność danych oraz typy opcjonalne, co wymaga integracji z mechanizmem null safety.

android/app/src/main/AndroidManifest.xml
<pre> <!-- android/app/src/main/AndroidManifest.xml --> <manifest xmlns:android="http://schemas.android.com/apk/res/android"> <!-- Dodaj PRZED tagiem <application> --> <uses-permission android:name="android.permission.INTERNET" /> <application android:label="weather_app" android:name="\${applicationId}" android:icon="@mipmap/ic_launcher"> <!-- reszta konfiguracji... --> </application> </manifest> </pre>

Bez uprawnienia INTERNET aplikacja cicho ignoruje żądania!

Flutter nie rzuca wyjątku gdy brak uprawnienia INTERNET - po prostu żądanie HTTP się nie udaje.

Objaw: `HttpException` lub `SocketException` 'Connection refused' albo brak odpowiedzi.

Zawsze dodaj uprawnienie jako PIERWSZY krok po stworzeniu projektu, zanim napiszesz kod HTTP.

Plik: `android/app/src/main/AndroidManifest.xml`, przed tagiem `<application>`.

8.3 Model danych i serwis API

Interakcje sieciowe wymagają jawnego modelowania stanów: ładowanie, sukces, błąd. Stan ładowania sygnalizuje operację w toku, błąd reprezentuje niepowodzenie (np. brak sieci), a sukces zawiera dane. Separacja tych stanów umożliwia deterministyczne renderowanie UI. Obsługa wyjątków (`try/catch`) powinna być zintegrowana z aktualizacją stanu aplikacji.

weather_service.dart - pełna implementacja	Wyjaśnienie
<code>import 'dart:convert'; // jsonDecode()</code>	dart:convert = wbudowana biblioteka JSON (bez import z pub.dev!)
<code>import 'package:http/http.dart' as http;</code>	as http = alias - unikamy konfliktu nazw
<code>class WeatherService {</code>	
<code> static const _baseGeo =</code>	static const = stała na poziomie klasy (jak companion object)
<code> 'https://geocoding-api.open-meteo.com/v1/search';</code>	
<code> static const _baseWeather =</code>	
<code> 'https://api.open-meteo.com/v1/forecast';</code>	
<code> // Pobierz współrzędne dla nazwy miasta</code>	
<code> Future<({double lat, double lon})> _getCoords(String city) async {</code>	Record type (lat, lon) - Dart 3.0+. Jak Pair w Kotlinie.
<code> final uri = Uri.parse(_baseGeo).replace(queryParameters: {</code>	Uri.replace = bezpieczne budowanie URL z parametrami
<code> 'name': city, 'count': '1', 'language': 'pl',</code>	queryParameters musi być Map<String, String>
<code> });</code>	
<code> final response = await http.get(uri);</code>	http.get() = suspend żądanie GET, czeka na odpowiedź
<code> if (response.statusCode != 200) {</code>	Sprawdź kod HTTP ZAWSZE
<code> throw Exception('Geocoding error: \${response.statusCode}');</code>	Rzuć wyjątek z opisem, łatwiejsze debugowanie
<code> }</code>	
<code> final data = jsonDecode(response.body) as Map<String, dynamic>;</code>	jsonDecode = parsuj JSON string do mapy Dart
<code> final results = data['results'] as List?;</code>	Lista? - może być null jeśli miasto nieznane
<code> if (results == null results.isEmpty) {</code>	
<code> throw Exception('Miasto nie znaleziono: \$city');</code>	
<code> }</code>	
<code> final first = results.first as Map<String, dynamic>;</code>	
<code> return (lat: first['latitude'] as double,</code>	Zwróć record - destrukuryzacja w callerze
<code> lon: first['longitude'] as double);</code>	
<code> }</code>	
<code> Future<WeatherData> fetchWeather(String city) async {</code>	Główna metoda, pobierz pogodę dla miasta
<code> final coords = await _getCoords(city); // krok 1</code>	Najpierw geolokalizacja, potem pogoda
<code> final uri = Uri.parse(_baseWeather).replace(queryParameters: {</code>	
<code> 'latitude': '\${coords.lat}',</code>	Parametry jako String (wymóg queryParameters)
<code> 'longitude': '\${coords.lon}',</code>	
<code> 'current':</code>	
<code>'temperature_2m,wind_speed_10m,relative_humidity_2m,weather_code',</code>	
<code> 'timezone': 'Europe/Warsaw',</code>	Strefa czasowa - ważne dla hourly/daily
<code> });</code>	
<code> final response = await http.get(uri);</code>	
<code> if (response.statusCode != 200) {</code>	
<code> throw Exception('Weather API error: \${response.statusCode}');</code>	
<code> }</code>	
<code> return WeatherData.fromJson(</code>	Parsuj JSON do obiektu Dart przez fabrykę
<code> jsonDecode(response.body) as Map<String, dynamic>,</code>	
<code> cityName: city,</code>	Dodajemy nazwę miasta (API jej nie zwraca)
<code>);</code>	
<code> }</code>	
<code>}</code>	

8.4 FutureBuilder: wyświetlanie asynchronicznych danych

FutureBuilder to widget który automatycznie obsługuje stany Future: ładowanie, sukces i błąd. Idealny dla jednorazowych żądań (nie Flow jak w Androidzie). Dla danych, które się zmieniają w czasie, użyj StreamBuilder.

FutureBuilder - użycie	Wyjaśnienie
<pre>FutureBuilder<WeatherData>(future: _weatherFuture, // Future przypisany raz! builder: (context, snapshot) { // Stan 1: czekanie na dane (loading) if (snapshot.connectionState == ConnectionState.waiting) { return const Center(child: CircularProgressIndicator()); } // Stan 2: błąd if (snapshot.hasError) { return Center(child: Text('Błąd: \${snapshot.error}')); } // Stan 3: brak danych (Future zakończony, ale null) if (!snapshot.hasData) { return const Center(child: Text('Brak danych')); } // Stan 4: sukces final weather = snapshot.data!; // bezpieczne po hasData return WeatherCard(weather: weather); },)</pre>	Generic typ = co Future zwróci. Tu WeatherData. WAŻNE: przypisz Future w initState(), nie w build()! snapshot = AsyncSnapshot<WeatherData> waiting = Future jest w toku hasError = Future rzucił wyjątek snapshot.error = złapany wyjątek hasData gwarantuje non-null - ! nie spowoduje crashu

Nie twórz Future wewnątrz build(). To anty-wzorec!

```
// ŹLE
FutureBuilder(future: api.fetchWeather(city), ...) // w build()
// Za każdym razem gdy widget się odbudowuje, tworzone jest NOWE żądanie HTTP!

// DOBRZE
class _MyState extends State {
  late final Future<WeatherData> _future;
  void initState() { super.initState(); _future = api.fetchWeather(city); }
  Widget build(_) => FutureBuilder(future: _future, ...); // ten sam Future
}
```

9. Motyw aplikacji wg Material Design 3

System motywów definiuje spójny zestaw stylów wizualnych: kolory, typografię, komponenty. Flutter domyślnie implementuje Material Design 3 (Material You). Material Design 3 wprowadza dynamiczne schematy kolorystyczne oraz rozszerzoną personalizację. Motyw jest propagowany w drzewie widgetów i dostępny przez kontekst. Umożliwia to centralne zarządzanie wyglądem aplikacji i redukcję duplikacji stylów. ThemeData centralizuje wszystkie decyzje projektowe: kolory, typografię, kształty. Prawidłowe użycie Theme eliminuje hardcoded kolory i sprawia, że aplikacja automatycznie obsługuje dark mode.

9.1. Konfiguracja ThemeData

ThemeData agreguje konfigurację wizualną aplikacji. Zawiera definicje palety kolorów, stylów tekstu oraz komponentów. Dostęp do motywu odbywa się przez Theme.of(context), co umożliwia kontekstowe stylowanie widgetów. Spójność wizualna wynika z używania wspólnych definicji zamiast lokalnych stylów.

Konfiguracja MaterialApp z ThemeData i GoRouter

```
// main.dart - konfiguracja motywu aplikacji
MaterialApp(
  title: 'WeatherApp',
  theme: ThemeData(
    // ColorScheme.fromSeed = generuje cały schemat kolorów z jednego koloru bazowego
    // Material 3 automatycznie tworzy 25+ kolorów (primary, secondary, tertiary, itp.)
    colorScheme: ColorScheme.fromSeed(
      seedColor: const Color(0xFF027DFD), // Kolor bazowy (niebieski)
      brightness: Brightness.light,        // light lub dark
    ),
    useMaterial3: true, // WYMAGANE dla Material 3 (domyślnie true od Flutter 3.16)

    textTheme: TextTheme( // Opcjonalna customizacja Typografii
      displayLarge: GoogleFonts.notoSansTextTheme().displayLarge,
      headlineMedium: const TextStyle(fontWeight: FontWeight.bold),
    ),

    cardTheme: const CardTheme( // Kształty komponentów
      elevation: 4,
      shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.all(Radius.circular(16)),
      ),
    ),
  ),

  darkTheme: ThemeData( // Dark mode - osobny motyw
    colorScheme: ColorScheme.fromSeed(
      seedColor: const Color(0xFF027DFD),
      brightness: Brightness.dark,
    ),
    useMaterial3: true,
  ),
  themeMode: ThemeMode.system, // system = śledź ustawienia telefonu

  routerConfig: appRouter, // GoRouter (zamiast home:)
)
```

9.2. Używanie kolorów z Theme - bez hardcodowania wartości

Obsługa trybu jasnego i ciemnego polega na definiowaniu alternatywnych zestawów motywów. System może automatycznie wybierać tryb na podstawie ustawień urządzenia. Kluczowe jest zapewnienie odpowiedniego kontrastu i czytelności. Przełączanie motywów może być dynamiczne, co wymaga reaktywnego podejścia do stanu aplikacji.

Kolory z Theme.of(context)

```
// ŻŁE - hardcoded kolor
Container(color: const Color(0xFF027DFD), ...) // Złamię dark mode!

// DOBRZE - kolor z Theme (automatycznie zmienia się z dark/light mode)
Container(
  color: Theme.of(context).colorScheme.primary, // Kolor główny schematu
  child: Text('Tekst',
    style: TextStyle(
      color: Theme.of(context).colorScheme.onPrimary, // Tekst na primary
    ),
  ),
)

// Dostępne kolory w ColorScheme (Material 3):
```

```
// primary, onPrimary, primaryContainer, onPrimaryContainer
// secondary, onSecondary, secondaryContainer, onSecondaryContainer
// surface, onSurface, surfaceVariant, onSurfaceVariant
// error, onError, errorContainer, onErrorContainer

// Krótszy zapis (extension)
final cs = Theme.of(context).colorScheme; // Przypisz do zmiennej
Container(color: cs.primaryContainer, child: Text('', style: TextStyle(color: cs.onPrimaryContainer)))
```

10. Projekt WeatherApp - architektura i struktura

Aplikacja jest podzielona na warstwy: UI, logika biznesowa oraz dostęp do danych. Modele reprezentują strukturę danych (np. pogodę), serwisy odpowiadają za komunikację z API, a widgety prezentują dane. Struktura katalogów powinna odzwierciedlać tę separację (np. `models/`, `services/`, `screens/`, `widgets/`). Architektura wspiera modularność i skalowalność. Połączymy teraz wszystkie poznane elementy w spójną aplikację. WeatherApp pozwoli wyszukać miasto, wyświetli aktualną pogodę i prognozę na 7 dni, z możliwością zapisania ulubionych miast.

10.1. Struktura katalogów projektu

Model danych reprezentuje strukturę odpowiedzi API w formie typowanej. Pola są zazwyczaj niemutowalne (`final`), co zapewnia spójność danych. Konstruktor oraz metoda `fromJson` umożliwiają inicjalizację obiektów na podstawie danych wejściowych. Model stanowi kontrakt między warstwą danych a UI.

Struktura lib/ projektu WeatherApp	
lib/	
├─ main.dart	← MaterialApp + Provider setup + GoRouter
├─ core/	
│ └─ router/	
│ │ └─ app_router.dart	← Definicja tras GoRouter
│ └─ theme/	
│ │ └─ app_theme.dart	← ThemeData light i dark
├─ data/	
│ └─ models/	
│ │ └─ weather_data.dart	← Model danych + fromJson()
│ │ └─ city_geo.dart	← Model geolokalizacji
│ └─ services/	
│ │ └─ weather_service.dart	← HTTP calls do Open-Meteo API
├─ providers/	
│ └─ weather_provider.dart	← ChangeNotifier - stan aplikacji
└─ ui/	
│ └─ screens/	
│ │ └─ home_screen.dart	← Lista ulubionych miast + wyszukiwarka
│ │ └─ detail_screen.dart	← Szczegóły pogody + prognoza 7 dni
│ └─ widgets/	
│ │ └─ weather_card.dart	← Karta z aktualną pogodą
│ │ └─ forecast_tile.dart	← Wiersz prognozy dziennej
│ │ └─ city_search_bar.dart	← TextField z auto-complete
│ │ └─ error_widget.dart	← Widget błędu z przyciskiem Retry

10.2. main.dart, punkt startowy

Serwis enkapsuluje logikę komunikacji z zewnętrznym API. Odpowiada za budowę zapytań, obsługę odpowiedzi oraz mapowanie danych. Abstrakcja ta izoluje szczegóły implementacyjne i umożliwia łatwą wymianę źródła danych lub testowanie (np. mockowanie).

main.dart

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

```
import 'core/router/app_router.dart';
import 'core/theme/app_theme.dart';
import 'providers/weather_provider.dart';

void main() {
  // runApp() = uruchom aplikację z podanym widgetem jako korzeniem drzewa
  runApp(const MyApp());
}

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

  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      // Provider na SZCZYGIE drzewa - dostępny wszędzie
      create: (_) => WeatherProvider(),
      child: MaterialApp.router(
        // MaterialApp.router = integracja z GoRouter (nie używaj 'home:')
        title: 'WeatherApp',
        theme: AppTheme.lightTheme,
        darkTheme: AppTheme.darkTheme,
        themeMode: ThemeMode.system,
        routerConfig: appRouter, // GoRouter config
        debugShowCheckedModeBanner: false, // Usuń banner DEBUG
      ),
    );
  }
}
```

11. Zadania do wykonania

Zadania prowadzą przez budowę WeatherApp krok po kroku. Każde kolejne buduje na poprzednim, dlatego nie pomijaj kolejności. Każde zadanie zakończ weryfikacją zanim przejdziesz do następnego.

Zadanie 1. (20 pkt) Środowisko i projekt bazowy

- 1.1 Zainstaluj Flutter SDK (stable channel). Uruchom flutter doctor - wszystkie wymagane checkmarki zielone.
 - 1.2 Zainstaluj wtyczki Flutter i Dart w Android Studio. Utwórz i uruchom AVD (Android Virtual Device).
 - 1.3 Stwórz projekt: flutter create --org pl.edu.pam --project-name weather_app weather_app.
 - 1.4 Uruchom aplikację na AVD: flutter run. Zmodyfikuj tekst w domyślnym CounterApp, sprawdź Hot Reload (r).
 - 1.5 Dodaj uprawnienie INTERNET do AndroidManifest.xml. Dodaj pakiet http: ^1.2.1 i provider: ^6.1.2 do pubspec.yaml.
Uruchom flutter pub get. Sprawdź w pubspec.lock, że paczki zostały dodane.
- WERYFIKACJA: Screenshot AVD z działającą domyślną aplikacją + flutter doctor bez błędów.

Zadanie 2. (25 pkt) Modele danych i serwis API

- 2.1 Utwórz model WeatherData z polami: city (String), temperature (double), humidity (int), windSpeed (double), weatherCode (int), updatedAt (DateTime).
Zaimplementuj fromJson() fabrykę parsującą odpowiedź Open-Meteo.
 - 2.2 Utwórz WeatherService z metodą fetchWeather(String city): Future<WeatherData>.
Dwuetaowo: geolokalizacja miasta → dane pogodowe.
 - 2.3 Przetestuj serwis w main.dart (tymczasowo): wywołaj fetchWeather('Warszawa') i wydrukuj wynik.
Sprawdź logi Logcat / flutter run - poprawna temperatura i wilgotność dla Warszawy.
 - 2.4 Dodaj obsługę błędów: rzuć WeatherException(String message) dla 404, błędów sieci i pustych wyników.
- WERYFIKACJA: Dane pogodowe wyświetlają się w konsoli. Przetestuj z nazwą nieistniejącego miasta - pojawia się exception.

Zadanie 3. (35 pkt) Widgety i pełne UI

- 3.1 Zaimplementuj WeatherProvider (ChangeNotifier) z polami: cities (List<String> = ulubione),

weather (Map<String, WeatherData>), isLoading, error. Metody: loadWeather(), toggleFavorite().

3.2 Skonfiguruj GoRouter z dwiema trasami: '/' (HomeScreen) i '/detail/:city' (DetailScreen).

3.3 Zbuduj HomeScreen z: TextField do wyszukiwania miasta (onSubmitted wywołuje loadWeather), ListView.builder z kartami ulubionych miast, obsługą stanów loading/error/empty.

3.4 Zbuduj DetailScreen wyświetlający: nazwę miasta, aktualną temperaturę z ikoną pogody (Weather Code → ikona Material Icons), wilgotność, prędkość wiatru.

3.5 Dodaj prognozę na 7 dni przez osobne żądanie API (parametr &daily=temperature_2m_max,temperature_2m_min). Wyświetl w ListView jako 7 wierszy (ForecastTile widget).

WERYFIKACJA: Demo na AVD - wyszukanie Warszawy, widok szczegółów, prognoza 7 dni, nawigacja Back.

Zadanie 4. (20 pkt) Ulepszenia, polerowanie i funkcje dodatkowe

4.1 Zaimplementuj dark mode: sprawdź że aplikacja przełącza motyw automatycznie po zmianie ustawień systemu (themeMode: ThemeMode.system). Użyj wyłącznie kolorów z Theme.of(context).

4.2 Dodaj animacje przejścia między ekranami: customTransitionPage w GoRouter lub Hero widget na ikonie/obrazie między HomeScreen a DetailScreen.

4.3 Zapisz ulubione miasta w SharedPreferences (dodaj pakiet shared_preferences: ^2.3.2). Miasta mają być dostępne po restarcie aplikacji.

4.4 Dodaj Pull-to-Refresh (RefreshIndicator widget) na HomeScreen - odświeża dane wszystkich ulubionych miast.

WERYFIKACJA: Dark mode działa. Ulubione przetrwają restart. Pull-to-refresh odświeża dane.

12. Kryteria oceniania

12.1. Punktacja zadań

Zadanie	Punkty / Co weryfikuje prowadzący
Zad. 1: Środowisko	20 pkt - flutter doctor bez błędów, Hot Reload działa, pakiety dodane do pubspec.yaml.
Zad. 2: Modele i API	25 pkt - WeatherData.fromJson() poprawny, fetchWeather() zwraca dane, obsługa błędów.
Zad. 3: UI	35 pkt - HomeScreen z wyszukiwarką, DetailScreen z prognozą, nawigacja, stany UI.
Zad. 4: Polisz	20 pkt - Dark mode, SharedPreferences, animacje lub Pull-to-Refresh.
RAZEM	100 pkt

12.2. Skala ocen

Ocena	Punkty / wymagania
5.0	90-100 pkt - Wszystkie zadania. Dark mode, SharedPreferences, animacje. Kod bez hardcoded kolorów.
4.5	80-89 pkt - Zadania 1-3 kompletne + przynajmniej 2 elementy Zadania 4.
4.0	70-79 pkt - Zadania 1-3. Kompletny UI: wyszukiwarka, szczegóły, prognoza 7 dni.
3.5	60-69 pkt - Zadania 1-2 + podstawowy HomeScreen i DetailScreen (bez prognozy).
3.0	50-59 pkt - Zadania 1-2. API działa, dane wyświetlane w konsoli lub prostym Text.
2.0	0-49 pkt - Projekt nie kompiluje się lub API nie zwraca danych.

13. Dart vs Kotlin - tabela porównawcza

Jeśli znasz Kotlin z wcześniejszych ćwiczeń, ta tabela pozwoli szybko przełożyć wiedzę na Dart. Różnice są mniejsze niż się wydaje.

Kotlin	Dart - odpowiednik
val x = 5 / var x = 5	final x = 5 (val) / var x = 5 (var)

Kotlin	Dart - odpowiednik
fun greet(name: String): String	String greet(String name) { } lub String greet(String name) =>
fun greet(name: String = 'Jan')	String greet({String name = 'Jan'}) - parametry domyślne
data class Person(val name: String)	class Person { final String name; const Person({required this.name}); }
object MySingleton	class MySingleton { static final instance = MySingleton._(); MySingleton._(); }
companion object { fun create() }	static factory constructor lub factory keyword
?. / ?: (Elvis) / !!	?. / ?? (Elvis) / ! (non-null assertion)
listOf() / mutableListOf()	const [] (immutable) / [] (growable list)
mapOf() / mutableMapOf()	const {} (immutable) / {} (growable map)
if (x is String) x.length	if (x is String) (x as String).length - smart cast działa też w Dart
when (x) { is Int -> ..., is String -> ... }	switch (x) { case int n: ..., case String s: ... } (Dart 3 patterns)
coroutineScope { launch { } }	brak bezpośredniego odpowiednika, Future.wait([]) lub async/await
Flow<T>	Stream<T> - emituje wartości w czasie. StreamBuilder w widżetach.
suspend fun fetch(): T	Future<T> fetch() async { }
launch { } / async { }	unawaited(myFuture()) / await myFuture()
try-catch-finally	try-catch-finally - identyczna składnia!
@Composable fun MyWidget()	Widget build(BuildContext context), w klasie dziedziczącej Widget
remember { mutableStateOf(x) }	setState(() { _x = newX; }) lub ChangeNotifier + notifyListeners()
LazyColumn { items(list) { } }	ListView.builder(itemBuilder: (ctx, i) => ...)
NavHost + composable(route)	GoRouter routes: [GoRoute(path: '/', builder: ...)]
hiltViewModel<VM>()	context.read<Provider>() lub Consumer<Provider>()
@Inject constructor	Brak DI wbudowanego, użyj Provider lub GetIt (service locator)

14. Najczęstsze błędy i ich rozwiązania

Poniższa tabela zawiera błędy specyficzne dla Fluttera i Darta, które student napotka podczas realizacji zadań. Wiele z nich jest specyficznych dla podejścia Flutter i nie dotyczy programowania natywnego.

Komunikat błędu / objaw	Przyczyna i rozwiązanie
SocketException: Connection refused / Failed host lookup	Brak uprawnień INTERNET w AndroidManifest.xml lub literówka w URL. Dodaj <uses-permission android:name="android.permission.INTERNET"/> przed tagiem <application>.
setState() called after dispose()	Wywołujesz setState() po zniszczeniu widżetu (np. nawigujesz dalej przed zakończeniem Future). Sprawdź if (mounted) przed setState(): if (mounted) setState(() { ... });
type 'Null' is not a subtype of type 'String'	JSON zwrócił null dla pola które w modelu jest non-nullable. Dodaj ? do typu w modelu LUB użyj ?? w fromJson(): json['field'] as String? ?? 'default'.
Unhandled Exception: Bad state: No element	Wywołałeś .first lub .last na pustej liście/mapie. Sprawdź isEmpty przed dostępem lub użyj .firstOrNull.
Provider not found / ProviderNotFoundException	Widget próbuje użyć context.watch/read() ale Provider nie jest w drzewie powyżej. Sprawdź czy ChangeNotifierProvider jest wyżej w drzewie niż widżet konsumujący.
Hot Reload nie działa (changes not visible)	Zmieniłeś kod poza metodą build() (np. initState, konstruktor, static). Hot Reload nie przeładowuje stanu - użyj Hot Restart (R) lub pełnego restartu.
FutureBuilder wyświetla loading w nieskończoność	Future jest tworzony w build() - za każdym rebuildem nowe Future. Przenieś Future do initState() jako late final Future<T> _future.
The argument type 'String?' can't be assigned to 'String'	Próbujesz przekazać nullable String? tam gdzie wymagany String. Użyj ! (pewny non-null), ?? 'default' lub sprawdź null wcześniej.
RenderFlex overflowed by X pixels on the bottom	Column/Row ma za dużo dzieci i nie mieści się na ekranie. Zawień Column w SingleChildScrollView lub użyj Expanded/Flexible na dzieciach rozciągających.

Komunikat błędu / objaw	Przyczyna i rozwiązanie
<code>pubspec.yaml</code> - <code>Unrecognized keys</code>	Błąd wcięcia w YAML (spacje, nie tabulatory!) lub literówka w kluczu. YAML jest wrażliwy na wcięcia - sprawdź dokładnie alignment.

Narzędzia diagnostyczne dla Flutter

flutter analyze	- statyczna analiza kodu (linter). Uruchom przed każdym commit.
flutter test	- testy jednostkowe i widgetów.
flutter inspect / DevTools	- debugger UI, timeline, memory. Otwórz przez URL z terminala.
debugPrint('wartość: \$variable')	- bezpieczny print (nie blokuje UI thread).
Android Studio → Flutter Inspector	- wizualizacja drzewa widżetów na żywo.

Instrukcja Laboratoryjna Flutter 1: Dart, Flutter, Widgety, Nawigacja i REST API

Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki

MobileHub

Następne ćwiczenie: Flutter 2, Zarządzanie stanem (Riverpod), animacje, testy widgetów

ĆWICZENIE LABORATORYJNE - FLUTTER 2

Riverpod, animacje, Hive i testy widgetów

Programowanie Aplikacji Mobilnych - moduł Fluttera

Projekt: WeatherApp v2 - rozszerzenie projektu z Flutter Lab 1

Wymagania wstępne: ukończony Flutter Lab 1 (Dart, widgety, Provider, GoRouter, http)

Spis treści

1. Od Provider do Riverpod - dlaczego warto przejść?	25
2. Typy providerów - kiedy którego użyć?	27
3. Hive - lokalna baza danych NoSQL	30
4. Animacje niejawne - Animated* widgety	33
5. Animacje jawne - AnimationController	36
6. Testy widgetów - flutter_test	38
7. Mocktail - mockowanie zależności	41
8. Architektura projektu - Feature-First i Clean Architecture	42
9. Dostępność i internacjonalizacja	44
10. Flutter DevTools - diagnostyka i profilowanie	46
11. Zadania do wykonania	47
12. Kryteria oceniania	48
13. Najczęstsze błędy i ich rozwiązania	48

1. Od Provider do Riverpod - dlaczego warto przejść?

W Flutter Lab 1 zarządzaliśmy stanem przy pomocy Provider i ChangeNotifier. Provider jest świetny na start, ale w miarę rozrostu aplikacji ujawnia ograniczenia: trudność z zależnościami między providerami, brak wsparcia dla asynchroniczności, konieczność ręcznego dispose(). Riverpod rozwiązuje te problemy i jest oficjalnie rekomendowanym następcą Providera przez tego samego autora - Rémi Rousselet.

1.1. Problemy z Provider w dużych aplikacjach

Wraz ze wzrostem skali aplikacji pojawiają się ograniczenia związane z globalnym kontekstem i niejawnością zależności. Provider opiera się na BuildContext, co utrudnia dostęp do stanu poza drzewem widgetów (np. w warstwie serwisów lub testach jednostkowych). Złożone drzewo providerów prowadzi do problemów z czytelnością oraz trudności w śledzeniu przepływu danych. Dodatkowo, nieprecyzyjne użycie watch skutkuje nadmiarowymi przebudowami UI, co wpływa na wydajność. Brakuje również silnego wsparcia dla kompozycji i modularności stanu.

Problem z Provider	Jak Riverpod to rozwiązuje
Zależności między providerami wymagają uważnej kolejności w drzewie widgetów.	Providery Riverpoda są globalne i mogą odczytywać siebie nawzajem bez zależności od drzewa widgetów.

Problem z Provider	Jak Riverpod to rozwiązuje
Błędna kolejność to brak dostępu.	
ChangeNotifier musi być ręcznie dispose()owany. Zapomnienie to wyciek pamięci.	Riverpod automatycznie zarządza cyklem życia. Niszczy providera gdy nikt go nie obserwuje.
Brak natywnego wsparcia dla Future/Stream w jednym providerze. Trzeba samemu obsługiwać loading/error.	AsyncNotifierProvider i FutureProvider wbudowują obsługę trzech stanów: loading, data, error (AsyncValue).
Context.read() w callbackach może rzucić wyjątek jeśli widget jest nieaktywny.	ref.read() jest bezpieczne niezależnie od cyklu życia widgetu.
Trudne testowanie - trzeba owijać widgety w ProviderScope i dbać o kolejność.	Testy Riverpoda są proste: nadpisz jeden provider przez ProviderScope(overrides:[]).

Analogia: tablica ogłoszeń vs. radio na żywo

[PROVIDER] Tablica ogłoszeń w korytarzu szkoły: ogłoszenia wiszą w konkretnym miejscu. Możesz je przeczytać tylko jeśli jesteś w tym korytarzu (kontekst!). Jeśli zmienisz korytarz (przejdiesz do innego ekranu), musisz szukać innej tablicy.

[RIVERPOD] Radio na żywo: możesz dostroić się do dowolnej stacji z dowolnego miejsca w domu. Nie musisz wiedzieć gdzie jest nadajnik - po prostu podajesz nazwę stacji. Radio pamięta czy ktoś słucha i wyłącza się samo gdy nikt nie słucha (auto dispose).

1.2. Konfiguracja Riverpod

Riverpod eliminuje zależność od `BuildContext`, wprowadzając globalny kontener providerów zarządzany przez `ProviderScope`. Każdy provider jest deklaratywną jednostką logiki, która może być niezależnie testowana i komponowana. Konfiguracja sprowadza się do opakowania aplikacji w `ProviderScope` oraz definiowania providerów jako globalnych obiektów. System automatycznie zarządza cyklem życia zależności, w tym ich inicjalizacją i usuwaniem (`dispose`), co redukuje ryzyko wycieków pamięci.

pubspec.yaml - Riverpod

```
# pubspec.yaml - zastąp lub dodaj obok provider
dependencies:
  flutter_riverpod: ^2.5.1      # Riverpod dla Flutter (z integracją widgetów)
  riverpod_annotation: ^2.3.5  # Opcjonalnie - generowanie kodu przez @riverpod

dev_dependencies:
  riverpod_generator: ^2.4.0    # Generuje providery z adnotacji (jak KSP w Androidzie)
  build_runner: ^2.4.9         # Runner dla generatora

# Uruchom po dodaniu
flutter pub get

# Dla generatora kodu (riverpod_generator) - opcjonalne, ale wygodne:
dart run build_runner watch --delete-conflicting-outputs
```

main.dart - konfiguracja ProviderScope (owiń całą aplikację)

```
void main() {
  runApp(
    const ProviderScope(      // WYMAGANE - rejestr wszystkich providerów Riverpoda
      child: MyApp(),         // Bez ProviderScope: wyjątek MissingProviderScopeException
    ),
  );
}
```

```

}
// MyApp.dart - użyj ConsumerWidget zamiast StatelessWidget
class MyApp extends ConsumerWidget { // <- ConsumerWidget, nie StatelessWidget
  const MyApp({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) { // <- dodany WidgetRef ref
    // ref = dostęp do providerów (jak context w Provider, ale bezpieczniejszy)
    return MaterialApp.router(routerConfig: appRouter, ...);
  }
}

```

2. Typy providerów - kiedy którego użyć?

Riverpod oferuje kilka wyspecjalizowanych typów providerów. Wybór właściwego jest kluczowy - błędny typ to nadmiarowy kod i trudne debugowanie. Poniżej: kompletna mapa decyzji.

Typ providera	Kiedy używać / Co przechowuje / Przykład
Provider<T>	Wartości tylko do odczytu - stałe, konfiguracja, obiekty serwisów. Nie zmienia się w czasie. Przykład: <code>Provider<WeatherService>((ref) => WeatherService())</code> .
StateProvider<T>	Prosty stan mutable - licznik, przełącznik, wybrany indeks. Nie używaj dla złożonych obiektów. Przykład: <code>StateProvider<String>((ref) => 'Warszawa')</code> .
FutureProvider<T>	Dane asynchroniczne tylko do odczytu - jednorazowe żądanie API. Automatycznie zarządza AsyncValue (loading/data/error). Przykład: <code>FutureProvider<WeatherData>((ref) => service.fetch())</code> .
StreamProvider<T>	Dane asynchroniczne ze strumienia - WebSocket, Firestore real-time, sensor. Jak FutureProvider ale dla Stream<T>. Przykład: <code>StreamProvider<double>((ref) => locationStream())</code> .
NotifierProvider<N, T>	Złożony mutable stan z logiką. Klasa Notifier ma metody zmieniające stan. Synchroniczny odpowiednik ChangeNotifier. Przykład: lista zadań add/remove/toggle.
AsyncNotifierProvider<N, T>	Jak NotifierProvider ale dla danych asynchronicznych. Build() zwraca Future<T>. Najlepszy do zarządzania danymi z API z możliwością odświeżania. Przykład: WeatherNotifier z metodą reload().

Dlaczego tyle typów? Czy nie wystarczyłby jeden?

Każdy typ jest zoptymalizowany dla swojego przypadku użycia.

FutureProvider<T> automatycznie anuluje poprzednie żądanie gdy parametr się zmieni, Provider<Future<T>> tego nie robi.

StateProvider<T> dla prostego inta jest szybszy niż NotifierProvider; mniej boilerplate. Typy są jak narzędzia: młotek, śrubokręt, klucz. Każde robi jedno dobrze, zamiast jedno robi wszystko średnio.

2.1. NotifierProvider - przykład: ulubione miasta

NotifierProvider służy do zarządzania stanem synchronicznym z możliwością jego modyfikacji. Stan jest enkapsulowany w klasie rozszerzającej `Notifier<T>`, gdzie `T` reprezentuje model danych (np. lista miast). Operacje biznesowe (dodawanie, usuwanie, aktualizacja) są implementowane jako metody tej klasy. Zmiana stanu odbywa się poprzez przypisanie nowej wartości do `state`, co zachowuje niemutowalność i umożliwia precyzyjne śledzenie zmian.

favorites_provider.dart - NotifierProvider	Wyjaśnienie
<code>// Klasa Notifier - logika i stan</code>	Notifier<T> = klasa przechowująca stan i metody go zmieniające. Generyk T = typ stanu. Tu: Lista stringów (nazwy miast).
<code>class FavoritesNotifier extends Notifier<List<String>> {</code>	
<code> @override</code>	build() = tworzy stan początkowy. Wywoływane raz przy inicjalizacji.
<code> List<String> build() {</code>	
<code> return ['Warszawa', 'Kraków']; // stan domyślny</code>	Możesz tu async ładować z Hive/SharedPreferences
<code> }</code>	Metoda publiczna - widgety wywołują ją przez ref.read
<code> void addCity(String city) {</code>	
<code> if (state.contains(city)) return; // guard</code>	Sprawdź duplikat przed dodaniem
<code> state = [...state, city]; // NIEZMIENNOŚĆ!</code>	state = nowa lista (nie mutuj istniejącej)! Riverpod powiadomi widgety.
<code> }</code>	
<code> void removeCity(String city) {</code>	
<code> state = state.where((c) => c != city).toList();</code>	Filtruj przez where() zamiast remove() - zwraca nową listę
<code> }</code>	
<code> bool contains(String city) => state.contains(city);</code>	Metoda pomocnicza - getter bez zmiany stanu
<code>}</code>	
<code>// Definicja providera (globalna, poza klasą)</code>	Provider definiowany globalnie - dostępny z każdego miejsca
<code>final favoritesProvider =</code>	final = przypisz raz (stały globalny)
<code> NotifierProvider<FavoritesNotifier, List<String>>(FavoritesNotifier.new,</code>	Generyki: <KlasaNotifier, TypStanu>
<code>);</code>	Przełącz konstruktor klasy (nie instancję)

2.2. AsyncNotifierProvider - dane z API

AsyncNotifierProvider rozszerza model o operacje asynchroniczne i reprezentuje stan jako AsyncValue<T>. Pozwala to jawnie modelować trzy podstawowe stany: ładowanie, sukces oraz błąd. Logika pobierania danych (np. zapytania HTTP) jest enkapsulowana w metodzie build(), która może być asynchroniczna. Mechanizm automatycznie zarządza retry, cache oraz odświeżaniem danych, co upraszcza integrację z API.

weather_provider.dart - AsyncNotifierProvider	Wyjaśnienie
<code>class WeatherNotifier</code>	AsyncNotifier<T> = Notifier z asynchronicznym build()
<code> extends AsyncNotifier<WeatherData?> {</code>	T = WeatherData? (nullable - przed załadowaniem)
<code> @override</code>	
<code> Future<WeatherData?> build() async {</code>	build() jest async - zwraca Future. Riverpod auto-obsługuje loading/error.
<code> return null; //stan początkowy: brak danych</code>	Null = nic nie załadowane. UI zobaczy AsyncValue.data(null).
<code> }</code>	
<code> Future<void> fetch(String city) async {</code>	Publiczna metoda - widgety wywołują ref.read(wp.notifier).fetch(city)
<code> state = const AsyncLoading(); //spinner</code>	AsyncLoading() = stan ładowania. UI zobaczy CircularProgressIndicator.
<code> state = await AsyncValue.guard(() async {</code>	AsyncValue.guard = try-catch który konwertuje wyjątek na AsyncError
<code> final svc = ref.read(weatherServiceProvider);</code>	ref.read() - odczytaj inny provider (wstrzyknięcie zależności!)

weather_provider.dart - AsyncNotifierProvider	Wyjaśnienie
<code>return svc.fetchWeather(city);</code> <code>});</code>	Wywołaj serwis. Jeśli rzuci wyjątek, state = AsyncError automatycznie.
<code>}</code> <code>Future<void> reload() async {</code> <code> final city = /*zapamiętaj ostatnie miasto*/</code> <code>_lastCity;</code> <code> if (city != null) await fetch(city);</code> <code>}</code> <code>}</code>	Odśwież dane, wywołaj <code>ref.invalidateSelf()</code> lub <code>re-fetch</code> Przechowaj ostatnie miasto w polu klasy
<code>final weatherProvider =</code> <code> AsyncNotifierProvider<WeatherNotifier,</code> <code> WeatherData?>(() =></code> <code> WeatherNotifier());</code>	<code>() => WeatherNotifier()</code> - anonimowa fabryka (nie <code>.new</code> bo ma zależności)

2.3 Odczyt i modyfikacja stanu w widżetach

Interakcja widżetów ze stanem odbywa się poprzez referencję `ref`. Odczyt reaktywny (`ref.watch`) powoduje przebudowę widżetu przy zmianie stanu, natomiast `ref.read` umożliwia dostęp bez subskrypcji. Modyfikacja stanu polega na wywołaniu metod dostawcy (`notifier`), co separuje UI od logiki biznesowej. Precyzyjne zarządzanie tymi mechanizmami pozwala kontrolować zakres renderowania i optymalizować wydajność.

Mechanizm `select` umożliwia subskrypcję tylko fragmentu stanu zamiast całego obiektu. Redukuje to liczbę przebudów widżetów w sytuacji, gdy zmienia się tylko część danych. Jest to szczególnie istotne w przypadku złożonych struktur (np. list lub obiektów z wieloma polami), gdzie pełna obserwacja byłaby kosztowna obliczeniowo.

ConsumerWidget i ConsumerStatefulWidget
<pre>// ConsumerWidget - stateless widget z dostępem do ref class HomeScreen extends ConsumerWidget { const HomeScreen({super.key}); @override Widget build(BuildContext context, WidgetRef ref) { // ref.watch() = obserwuj i przebuduj przy zmianie (jak context.watch<T>() w Provider) final favorites = ref.watch(favoritesProvider); // List<String> final weatherAsync = ref.watch(weatherProvider); // AsyncValue<WeatherData?> return Scaffold(body: weatherAsync.when(// AsyncValue.when() = elegancki switch na wszystkie 3 stany loading: () => const CircularProgressIndicator(), error: (err, stack) => ErrorWidget(message: err.toString()), data: (weather) => WeatherCard(weather: weather),), floatingActionButton: FloatingActionButton(// ref.read() = jednorazowy odczyt / akcja (nie subskrybuje zmian) onPressed: () => ref.read(weatherProvider.notifier).fetch('Kraków'), child: const Icon(Icons.refresh),),); } }</pre>

```
// ConsumerStatefulWidget - stateful widget z ref
class SearchScreen extends ConsumerStatefulWidget {
  const SearchScreen({super.key});
  @override
  ConsumerState<SearchScreen> createState() => _SearchScreenState();
}

class _SearchScreenState extends ConsumerState<SearchScreen> {
  // ref jest dostępne bezpośrednio (jak this.ref)
  void _onSearch(String city) {
    ref.read(weatherProvider.notifier).fetch(city);
    ref.read(favoritesProvider.notifier).addCity(city);
  }
  // ...
}
```

Providery w Riverpod są tworzone i usuwane dynamicznie w zależności od użycia. Mechanizmy takie jak `autoDispose` pozwalają na automatyczne zwalnianie zasobów, gdy provider przestaje być używany. Możliwe jest również utrzymanie stanu w pamięci poprzez wyłączenie `auto-dispose` lub zastosowanie `cache`. Precyzyjna kontrola cyklu życia ma znaczenie dla aplikacji o ograniczonych zasobach.

Oddzielenie providerów od UI umożliwia ich testowanie w izolacji. Możliwe jest tworzenie instancji kontenera providerów w środowisku testowym oraz nadpisywanie zależności (`override`). Pozwala to symulować różne scenariusze (np. błędy API) bez konieczności uruchamiania aplikacji. Testy mają charakter deterministyczny i nie wymagają kontekstu widgetów.

ref.watch() tylko w build(), a nie w callbackach i initState()!

Zasada identyczna jak z `context.watch()` w Provider:

ref.watch() w `onPressed()` / `initState()` / `didChangeDependencies()` → wyjątek `ProviderException`.

ref.read() = wywołaj w callbackach i `initState()` (jednorazowe działanie).

ref.listen() = obserwuj i reaguj na zmiany bez przebudowywania widgetu (np. wyświetl `SnackBar`).

Riverpod eliminuje ograniczenia wynikające z zależności od kontekstu oraz wprowadza silniejsze typowanie i kontrolę nad zależnościami. Provider jest prostszy w użyciu na małą skalę, lecz trudniejszy do utrzymania w złożonych systemach. Riverpod oferuje lepszą modularność, testowalność oraz wydajność dzięki precyzyjnemu zarządzaniu subskrypcjami i cyklem życia stanu.

3. Hive - lokalna baza danych NoSQL

W Flutter Lab 1 wspomniano `SharedPreferences` do przechowywania prostych wartości (stringów, liczb). `SharedPreferences` jest odpowiedni dla ustawień. Do przechowywania złożonych obiektów (listy, zagnieżdżone dane, modele) znacznie lepszy jest Hive - lekka, wydajna baza NoSQL napisana w czystym Darcie.

3.1. Hive kontra inne opcje przechowywania

Hive jest lokalną bazą typu `key-value` zoptymalizowaną pod kątem wydajności i prostoty. Operuje bezpośrednio na plikach binarnych, co eliminuje narzut warstwy SQL. W porównaniu do `SQLite` (relacyjny model danych) oferuje niższą złożoność kosztem ograniczeń w zapytaniach. W zestawieniu z `SharedPreferences` zapewnia większą elastyczność typów danych i struktur. Rozwiązania takie jak `ObjectBox` czy `Isar` oferują bardziej zaawansowane możliwości indeksowania i zapytań, lecz kosztem większej złożoności. Hive jest optymalny dla lekkich, lokalnych danych bez potrzeby relacyjnych operacji.

Rozwiązanie	Kiedy używać / Wady i zalety
SharedPreferences	Proste wartości: bool, int, String, List<String>. Synchroniczne API. Nie nadaje się do złożonych obiektów ani dużych danych.
Hive	Obiekty Dart jako rekordy. Bardzo szybki (zapisy w mikros). Brak SQL - proste klucz→wartość i listy. Dobry do offline cache, ulubionych, historii.
Sqflite (SQLite)	Relacyjna baza danych. Dobra gdy potrzebne zapytania SQL, relacje, złożone filtry. Trudniejsze API niż Hive. Odpowiednik Room z Androida.
drift (dawniej moor)	Type-safe Dart wrapper na SQLite z generowaniem kodu. Jak Room dla Fluttera - bardziej ergonomiczny niż sqflite, ale więcej boilerplate.
Isar	Następca Hive od tego samego autora. Szybszy, z obsługą indeksów i zapytań. Wymaga więcej konfiguracji. Dobry wybór dla nowych projektów zamiast Hive.

3.2 Konfiguracja i inicjalizacja Hive

Inicjalizacja wymaga wskazania ścieżki systemowej oraz rejestracji adapterów typów. Proces rozpoczyna się przed uruchomieniem aplikacji, zwykle w `main()`. W środowiskach mobilnych konieczne jest użycie mechanizmu inicjalizacji asynchronicznej (np. `WidgetsFlutterBinding.ensureInitialized()`). Po konfiguracji otwierane są tzw. „boxy”, które stanowią jednostki przechowywania danych. Każdy box działa jako niezależny kontener.

pubspec.yaml - Hive	YAML
<pre>dependencies: hive: ^2.2.3 # Silnik Hive hive_flutter: ^1.1.0 # Integracja z Flutter (Hive.initFlutter) path_provider: ^2.1.3 # Ścieżka do katalogu danych aplikacji dev_dependencies: hive_generator: ^2.0.1 # Generator TypeAdapter z @HiveType/@HiveField build_runner: ^2.4.9 # Runner generatora (już dodany dla Riverpoda)</pre>	

main.dart - inicjalizacja Hive - inicjalizacja Hive PRZED runApp()	DART
<pre>Future<void> main() async { // Wymagane przed Hive.initFlutter() i runApp() WidgetsFlutterBinding.ensureInitialized(); // Inicjalizuj Hive z katalogiem danych aplikacji (DocumentsDirectory na iOS, // AppData na Androidzie) - hive_flutter robi to automatycznie await Hive.initFlutter(); // Zarejestruj adaptory dla niestandardowych typów (PRZED openBox!) Hive.registerAdapter(WeatherDataAdapter()); // generowany przez build_runner Hive.registerAdapter(CityGeoAdapter()); // Otwórz skrzynki (bazy) - nazwa = identyfikator, jak nazwa tabeli await Hive.openBox<WeatherData>('weatherCache'); // cache pogody await Hive.openBox<String>('favorites'); // ulubione miasta (String) runApp(const ProviderScope(child: MyApp())); }</pre>	

3.3. Model danych z adnotacjami Hive

Hive wykorzystuje generację kodu do mapowania obiektów Dart na format binarny. Klasy modelowe są oznaczane adnotacjami (`@HiveType`, `@HiveField`), które definiują identyfikatory pól. Każde pole musi mieć unikalny indeks, co zapewnia kompatybilność przy ewolucji schematu. Generator kodu tworzy adaptory serializacji/deserializacji. Model danych powinien być stabilny, ponieważ zmiany indeksów prowadzą do niespójności danych.

weather_data.dart - model z @HiveType	Wyjaśnienie
<code>import 'package:hive/hive.dart';</code>	Import adnotacji Hive
<code>part 'weather_data.g.dart'; //GENEROWANY plik</code>	part = część tego pliku (generowana przez build_runner)
<code>//typeId musi być unikalny! @HiveType(typeId: 0)</code>	typId 0-223: unikalne ID dla każdej klasy. Nie zmieniaj po wdrożeniu!
<code>// HiveObject! class WeatherData extends HiveObject {</code>	HiveObject = metody .save() i .delete() na obiekcie
<code> @HiveField(0) //indeks pola late String city;</code>	HiveField(index): unikalne pole w klasie. Nie zmieniaj kolejności! late = inicjalizacja przez Hive przy odczycie (nie w konstruktorze)
<code> @HiveField(1) late double temperature;</code>	
<code> @HiveField(2) late int humidity;</code>	
<code> @HiveField(3) late DateTime updatedAt;</code>	DateTime jest serializowany natywnie przez Hive
<code>// Konstruktor domyślny (wymagany przez Hive) WeatherData();</code>	Hive tworzy obiekty przez konstruktor domyślny (bezparametrowy)
<code>// Konstruktor fabryczny dla wygody factory WeatherData.from({ required String city, required double temperature, required int humidity, }) => WeatherData()</code>	
<code> ..city = city</code>	..city = cascade - przypisz pole i zwróć obiekt
<code> ..temperature = temperature ..humidity = humidity ..updatedAt = DateTime.now(); }</code>	

Generowanie kodu Hive
<pre># Generowanie kodu adaptera (uruchom w terminalu): dart run build_runner build --delete-conflicting-outputs # Lub w trybie watch (auto-generuje przy każdej zmianie): dart run build_runner watch --delete-conflicting-outputs # Efekt: plik weather_data.g.dart z klasą WeatherDataAdapter # Nie edytuj weather_data.g.dart - jest nadpisywany przy każdym build!</pre>

3.4. Operacje CRUD na skrzynce Hive

Operacje zapisu i odczytu są realizowane poprzez interfejs boxa. Dodanie lub aktualizacja danych odbywa się przez przypisanie wartości do klucza. Usuwanie polega na eliminacji wpisu lub czyszczeniu całego boxa. Odczyt jest synchroniczny i bardzo szybki, ponieważ dane są trzymane w pamięci po otwarciu. Hive wspiera również obserwację zmian (watch), co umożliwia reaktywne aktualizacje UI.

weather_cache_service.dart - operacje Hive	Wyjaśnienie
<code>class WeatherCacheService {</code>	Serwis enkapsulujący operacje na Hive
<code>// Pobierz otwarty Box (już otwarty w main())</code>	Hive.box() - synchroniczny, Box musi być już otwarty
<code>Box<WeatherData> get _box =></code> <code>Hive.box<WeatherData>('weatherCache');</code>	Nie otwieraj Box w serwisie - otwieranie jest async, rób to w main()
<code>// ZAPIS, klucz: nazwa miasta</code> <code>Future<void> cacheWeather(WeatherData data) async {</code> <code> await _box.put(data.city.toLowerCase(), data);</code> <code>}</code>	async mimo że Hive jest synchroniczny; dla spójności API put(key, value) - zapisz. Klucz: lowercase nazwa miasta
<code>// ODCZYT, zwróć null jeśli brak w cache</code> <code>WeatherData? getCache(String city) {</code> <code> return _box.get(city.toLowerCase());</code> <code>}</code>	Synchroniczny odczyt - Hive działa w pamięci get(key) - zwróć wartość lub null
<code>// SPRAWDŹ AKTUALNOŚĆ, starsze niż 30 minut to nieaktualne</code> <code>bool isFresh(String city) {</code> <code> final cached = getCache(city);</code> <code> if (cached == null) return false;</code> <code> final age = DateTime.now().difference(cached.updatedAt);</code> <code> return age.inMinutes < 30;</code> <code>}</code>	difference() = Duration między dwoma DateTime Dane aktualne przez 30 minut
<code>// USUŃ jeden wpis</code> <code>Future<void> remove(String city) async {</code> <code> await _box.delete(city.toLowerCase());</code> <code>}</code>	delete(key) - usuń klucz
<code>// LISTA WSZYSTKICH KLUCZY, (nazwy miast w cache)</code> <code>List<String> get cachedCities =></code> <code> _box.keys.cast<String>().toList();</code> <code>}</code>	keys - Iterable<dynamic>, cast<String> - typed

4. Animacje niejawne; Animated* widżety

Flutter dzieli animacje na dwie kategorie. Animacje niejawne (implicit) abstrahują zarządzanie stanem animacji, obsługują proste przejścia automatycznie, np. wystarczy zmienić wartość a Flutter sam animuje przejście (zmiana właściwości widżetu automatycznie inicjuje interpolację między stanami). Mechanizm ten upraszcza implementację kosztem ograniczonej kontroli nad przebiegiem animacji. Wewnętrznie Flutter zarządza cyklem życia animacji i synchronizacją z klatkami renderowania. Animacje jawne (explicit) dają pełną kontrolę nad czasem, przebiegiem i sekwencją. Zaczniemy od tych prostszych: niejawnych.

Analogia: automatyczna skrzynia biegów vs. manualna

ANIMACJE NIEJAWNE (Animated*) = automatyczna skrzynia biegów: Podajesz docelową wartość (temperatura, pozycja, kolor). Auto płynnie przejdzie do celu. Idealny dla prostych stanów: pokaż/ukryj, powiększ/pomniejsz, zmień kolor.

ANIMACJE JAWNE (AnimationController) = manualna skrzynia biegów: Kontrolujesz każdą milisekundę. Możesz cofnąć, zatrzymać, zapętlić, synchronizować. Potrzebne gdy animacja jest złożona, zależy od gestu użytkownika lub synchronizuje wiele elementów.

4.1. Przegląd Animated* widżetów

Rodzina `Animated*` obejmuje komponenty takie jak `AnimatedContainer`, `AnimatedOpacity`, `AnimatedPadding`. Każdy z nich interpoluje określony zestaw właściwości. Wspólnym elementem jest parametr czasu trwania (`duration`) oraz opcjonalna krzywa animacji (`curve`). Widżety te są zoptymalizowane pod kątem prostych transformacji UI.

Widget	Co animuje i przykład użycia
<code>AnimatedContainer</code>	Dowolna właściwość <code>Container</code> a: kolor, rozmiar, padding, margin, <code>borderRadius</code> . Najczęściej używany.
<code>AnimatedOpacity</code>	Przezroczystość (0.0-1.0). Efekt pokaż/ukryj. Zamień <code>Opacity</code> → <code>AnimatedOpacity</code> .
<code>AnimatedPositioned</code>	Pozycja w <code>Stack</code> (top, left, right, bottom). Animuje ruch elementu w obrębie <code>Stack</code> .
<code>AnimatedSwitcher</code>	Animowane przejście między RÓŻNYMI widżetami. Automatyczne fade/scale gdy child się zmieni.
<code>AnimatedCrossFade</code>	Płynne przejście między DWOMA konkretnymi widżetami (<code>firstChild</code> ↔ <code>secondChild</code>).
<code>AnimatedList</code>	Animowane dodawanie i usuwanie elementów listy (slide + fade). Wymaga <code>GlobalKey</code> .
<code>AnimatedAlign</code>	Animuje wyrównanie widżetu (<code>Alignment</code>) wewnątrz rodzica.
<code>TweenAnimationBuilder</code>	Animuje dowolną wartość (int, double, Color, Offset) z własnym Tween. Wszystko co nie ma dedykowanego <code>Animated*</code> widżetu.

4.2. Płynne przejście właściwości z AnimatedContainer

`AnimatedContainer` umożliwia interpolację wielu właściwości jednocześnie (rozmiar, kolor, padding, transformacje). Zmiana dowolnej z nich powoduje automatyczne przejście do nowego stanu. Jest to użyteczne w scenariuszach adaptacyjnego UI. Wewnętrznie wykorzystywany jest mechanizm tweenów dla każdej właściwości.

AnimatedContainer; karta pogody reagująca na stan	Wyjaśnienie
<code>class WeatherCard extends StatefulWidget { ... }</code>	<code>StatefulWidget</code> - stan kontroluje animację.
<code>// W _WeatherCardState</code> <code>bool _isExpanded = false; // stan karty</code>	Stan lokalny - rozwinięta/zwinięta karta.
<code>@override Widget build(BuildContext context) {</code> <code>return GestureDetector(</code>	<code>GestureDetector</code> - detekcja dotyku bez ripple
<code>onTap: () => setState(() => _isExpanded = !_isExpanded),</code>	Zmień stan po tapnięciu → <code>AnimatedContainer</code> się uruchomi.
<code>child: AnimatedContainer(</code>	Zamiast <code>Container</code> → <code>AnimatedContainer</code>
<code>duration: const Duration(milliseconds: 300),</code>	Czas trwania animacji - 200-400ms to 'naturalne' przejście.

AnimatedContainer; karta pogody reagująca na stan	Wyjaśnienie
<pre> curve: Curves.easeInOut, // Właściwości animowane, zmienia się płynnie height: _isExpanded ? 200.0 : 80.0, color: _isExpanded ? Theme.of(context).colorScheme.primaryContainer : Theme.of(context).colorScheme.surface, padding: EdgeInsets.all(_isExpanded ? 20.0 : 8.0), decoration: BoxDecoration(borderRadius: BorderRadius.circular(_isExpanded ? 24.0 : 12.0),), child: WeatherInfo(isExpanded: _isExpanded),),); </pre>	Krzywa animacji: easeInOut = powoli start, powoli koniec. Wszystkie właściwości będą interpolowane przez duration. Wysokość: 80 → 200 dp (animacja rozwinięcia) Kolor tła zmienia się wraz z rozwinięciem Padding też animowany! Zaokrąglenie narożników animowane Zawartość - nie animowana (tylko kontener)

4.3. Przejście między widżetami z AnimatedSwitcher

`AnimatedSwitcher` umożliwia animowaną zamianę jednego widżetu na inny. Mechanizm opiera się na identyfikacji widżetów poprzez klucze (`Key`). Przy zmianie dziecka wykonywana jest animacja wejścia/wyjścia. Możliwe jest definiowanie własnych przejść (np. fade, scale), co zwiększa elastyczność wizualną.

AnimatedSwitcher - zmiana ikony pogody	Wyjaśnienie
<pre> AnimatedSwitcher(duration: const Duration(milliseconds: 400), //Customowe przejście fade+scale (zamiast zwykłego fade) transitionBuilder: (child, animation) { return ScaleTransition(scale: animation, child: FadeTransition(opacity: animation, child: child),); }, // KLUCZOWE: każdy child musi mieć unikalny Key! child: Icon(_getWeatherIcon(weatherCode), //zmienia się key: ValueKey<int>(weatherCode), //unikalny Key! size: 64,),) </pre>	AnimatedSwitcher animuje gdy child się zmienia transitionBuilder = jak animujesz stary i nowy child ScaleTransition = animacja skali animation: 0.0→1.0 dla nowego, 1.0→0.0 dla starego Combine scale + fade dla efektu 'popping in' Bez Key Flutter myśli że to ten sam widżet → brak animacji! Gdy weatherCode się zmieni, zmieni się icon - AnimatedSwitcher to zauważy ValueKey(wartość) - identyfikuje unikalnie child dla AnimatedSwitcher

AnimatedSwitcher bez Key - najczęstszy błąd z animacjami!

Gdy child zmienia się, ale Key NIE jest ustawiony, Flutter nie 'widzi' zmiany. Myśli że to ten sam widżet o zmienionych właściwościach → brak animacji przejścia.

ZASADA: Każdy child AnimatedSwitcher MUSI mieć unikalny Key.

Dobre Keys: ValueKey(iconCode), ValueKey(weatherCode), ObjectKey(weatherData).
 Złe Keys: UniqueKey() - tworzy nowy Key przy każdym rebuild → zbyt częste animacje.

5. Animacje jawne: AnimationController

Gdy Animated* widgety nie wystarczają, potrzebujesz pełnej kontroli nad animacją, możliwości jej zatrzymania, cofnięcia lub synchronizacji z gestem - sięgamy po AnimationController. To niskopoziomowe API, ale daje nieograniczone możliwości.

5.1. AnimationController

Animacje jawne zapewniają pełną kontrolę nad przebiegiem animacji. AnimationController zarządza czasem i stanem animacji, umożliwiając ręczne sterowanie (start, stop, reverse). Wymaga synchronizacji z tickerem (TickerProvider), który dostarcza sygnały klatkowe. Podejście to jest bardziej złożone, ale niezbędne dla niestandardowych efektów.

Pulsujący wskaźnik temperatury z AnimationController	Wyjaśnienie
<pre>class PulsingTempIndicator extends StatefulWidget { ... }</pre>	StatefulWidget wymagany - AnimationController to mutable state
<pre>class _PulsingTempState extends State<PulsingTempIndicator> with SingleTickerProviderStateMixin {</pre>	with SingleTickerProviderStateMixin - dostarcza vsync (takt animacji)
<pre> late AnimationController _controller; late Animation<double> _scaleAnimation;</pre>	vsync synchronizuje animację z odświeżaniem ekranu (60fps) late = inicjalizacja w initState (nie w konstruktorze) Animation<T> = wartości w czasie (Tween zdefiniuje zakres)
<pre> @override void initState() { super.initState(); _controller = AnimationController(vsync: this, //provider taktu (this=mixin) duration: const Duration(seconds: 1),);</pre>	Twórz kontroler ZAWSZE w initState() this = SingleTickerProviderStateMixin dostarcza vsync Czas jednego cyklu animacji (0→1)
<pre> _scaleAnimation = Tween<double>(begin: 1.0, // rozmiar normalny end: 1.2, // 20% większy).animate(CurvedAnimation(parent: _controller, curve: Curves.easeInOut,));</pre>	Tween = interpolator: od begin do end begin = wartość na początku animacji (controller=0) end = wartość na końcu animacji (controller = 1) animate() = połącz Tween z kontrolerem i krzywą parent = skąd bierzemy czas (0.0-1.0) Krzywa kształtuje tempo: easeInOut = naturalne pulse
<pre> // Zapętl animację (pulsowanie) _controller.repeat(reverse: true); } // 0→1→0→1→...</pre>	reverse:true = animuj do końca, potem wstecz (ping-pong)
<pre> @override void dispose() { _controller.dispose(); // KONIECZNE! super.dispose(); }</pre>	Brak dispose() = wyciek pamięci. AnimationController nie jest GC'owany.

Pulsujący wskaźnik temperatury z AnimationController	Wyjaśnienie
<pre><code>@override Widget build(BuildContext context) { return ScaleTransition(scale: _scaleAnimation, child: Text('\${widget.temp}°C', style: const TextStyle(fontSize: 48, fontWeight: FontWeight.bold),),); }</code></pre>	ScaleTransition nasłuchuje _scaleAnimation automatycznie Nie potrzeba AnimatedBuilder, ScaleTransition jest gotowy widget.temp = dostęp do pól StatefulWidget z State

5.2. Krzywe animacji: Curves

Krzywe definiują nieliniową funkcję przejścia między wartościami początkowymi a końcowymi. Wpływają na percepcję dynamiki animacji (np. przyspieszenie, opóźnienie). Flutter udostępnia zestaw predefiniowanych krzywych (easeIn, easeOut, bounce). Dobór krzywej ma znaczenie dla ergonomii i naturalności interakcji.

Krzywa	Zachowanie i kiedy stosować
Curves.linear	Stała prędkość przez cały czas. Mechaniczne, nienatyczne. Używaj rzadko - tylko dla pasków postępu.
Curves.easeIn	Powoli startuje, przyspiesza. Dobry dla elementów wychodzących z ekranu (znikanie).
Curves.easeOut	Szybki start, zwalnia na końcu. Dobry dla elementów wchodzących (pojawianie) - naturalne wyhamowanie.
Curves.easeInOut	Powoli start, przyspiesza w środku, zwalnia na końcu. Najbardziej 'naturalny'. Użyj dla większości animacji UI.
Curves.bounceOut	Efekt odbicia na końcu (sprężyna). Dobry dla elementów które 'lądują' na docelowej pozycji.
Curves.elasticOut	Efekt sprężyny - przekracza cel i wraca. Zabawny, do efektów pull-to-refresh, ikonografii.
Curves.decelerate	Szybki start, stopniowe wyhamowanie. Podobny do easeOut ale bardziej agresywny.
CubicBezier(x1,y1,x2,y2)	Własna krzywa Beziera - pełna kontrola. Przydatny gdy projekt ma niestandardowe krzywe (Material 3: Curves.fastOutSlowIn).

5.3. Hero: animacja między ekranami

Mechanizm Hero (specjalny widżet) umożliwia płynne przejście elementu UI między dwoma ekranami poprzez współdzielenie identyfikatora. Element 'przelatuje' ze swojego miejsca na liście do miejsca w widoku szczegółów. Podczas nawigacji Flutter tworzy animację transformacji pozycji i rozmiaru między dwoma stanami. Wymaga spójnych tagów (tag) w obu ekranach. Technika ta poprawia ciągłość percepcyjną interfejsu. To jeden z najbardziej efektownych efektów UI we Flutterze i wymaga minimalnej konfiguracji.

Hero - implementacja w WeatherApp	Wyjaśnienie
<pre><code>// Na EKRANIE LISTY (HomeScreen) - karta miasta Hero(tag: 'weather-icon-\${cityName}', // UNIKALNY tag! child: Image.asset('assets/weather_\${weatherCode}.png',</code></pre>	Hero MUSI mieć identyczny tag na obu ekranach Hero opakowuje widżet który będzie animowany tag: unikalny identyfikator. Użyj danych (nie 'icon!') dla unikalności. child = widżet który 'przeleci' do DetailScreen

Hero - implementacja w WeatherApp	Wyjaśnienie
<code>width: 48, height: 48,</code>	Rozmiar może być inny niż na DetailScreen - Hero to interpoluje!
<code>),</code>	
<code>),</code>	
<code>// Na EKRANIE SZCZEGÓŁÓW (DetailScreen)</code>	Ten sam tag → Flutter wie że to 'ten sam' element
<code>Hero(</code>	
<code> tag: 'weather-icon-\$cityName', // ten sam tag!</code>	Identyczny tag jak na liście - Flutter zsynchronizuje
<code> child: Image.asset(</code>	
<code> 'assets/weather_\${weatherCode}.png',</code>	
<code> width: 128, height: 128, // większy rozmiar!</code>	Flutter animuje ROZMIAR podczas lotu - od 48x48 do 128x128
<code>),</code>	
<code>),</code>	
<code>// Nawigacja - GoRouter automatycznie obsługuje Hero!</code>	Hero działa ze wszystkimi systemami nawigacji (GoRouter, Navigator)
<code>context.push('/detail/\$cityName'); // Hero przeleci sam</code>	Wystarczy nawigować normalnie - Hero resztą zajmie się sam

Trzy reguły Hero: unikalność, spójność i prostota

- UNIKALNY TAG:** Dwa Hero z tym samym tagiem na tym samym ekranie = błąd ('There are multiple heroes...'). Użyj danych w tagu: 'weather-icon-\$cityName' zamiast 'icon'.
- SPÓJNY CHILD TYPE:** Hero na obu ekranach powinien zawierać podobne widżety (Image → Image). Można animować między różnymi typami, ale wygląd może być niespójny.
- PROSTOTA:** Hero najlepiej działa z obrazami i ikonami. Dla złożonych widżetów użyj `customTransitionPage` w GoRouter.

6. Testy widżetów - flutter_test

Flutter ma wbudowaną bibliotekę `flutter_test` do testowania widżetów. W przeciwieństwie do testów instrumentowanych Androida (które wymagają emulatora), testy widżetów Fluttera uruchamiają się wyłącznie na JVM - są szybkie i nie wymagają żadnego urządzenia.

🔍 Jak Flutter testuje widżety bez emulatora?

Flutter używa własnego silnika renderującego (Skia/Impeller). W trybie testowym Flutter uruchamia uproszczoną wersję tego silnika na maszynie deweloperskiej. Widżety są renderowane do bufora w pamięci, a testy mogą sprawdzać drzewo semantyczne, wymiary, teksty i interakcje bez wyświetlania czegokolwiek na ekranie. Jest to niezmiernie szybkie. Całą „suitę” testów widżetów uruchamia się w sekundy.

Porównanie:

Testy jednostkowe Dart: ~1ms per test

Testy widżetów Flutter: ~10-100ms per test (symulowany render)

Testy instrumentowane Android: ~5-30s per test (emulator)

6.1. Anatomia testu widżetu

Test widżetu symuluje środowisko renderowania i interakcje użytkownika. Składa się z trzech etapów: przygotowanie (`pumpWidget`), akcja (np. kliknięcie) oraz asercja (weryfikacja stanu UI). Testy są izolowane i wykonywane bez emulatora. Kluczowe jest odwzorowanie minimalnego drzewa widżetów potrzebnego do testu.

weather_card_test.dart - struktura testu	Wyjaśnienie
<code>import 'package:flutter_test/flutter_test.dart';</code>	flutter_test: główna biblioteka testów widжетów (wbudowana)
<code>import 'package:flutter/material.dart';</code>	Material: potrzebne do MaterialApp i widжетów Material
<code>import '../lib/ui/widgets/weather_card.dart';</code>	Importuj testowany widжет
<code>void main() {</code>	main() zawiera wszystkie testy - uruchamiany przez flutter test
<code> // group() = logiczna grupa testów (jak describe w Jest)</code>	group() = folder dla powiązanych testów. Opcjonalny ale zalecany.
<code> group('WeatherCard Widget Tests', () {</code>	Pierwszy argument = nazwa grupy (wyświetlana w raporcie)
<code> // Dane testowe (fixture) - używane w wielu testach</code>	
<code> final testWeather = WeatherData(</code>	Przygotuj dane poza testami (DRY)
<code> city: 'Warszawa',</code>	
<code> temperature: 22.5,</code>	
<code> humidity: 65,</code>	
<code> weatherCode: 1,</code>	
<code>);</code>	
<code> // testWidgets() = jeden test widgetu</code>	Jak @Test w JUnit - jeden przypadek testowy
<code> testWidgets('shows city name', (WidgetTester</code>	tester: WidgetTester = narzędzie do interakcji z widżetem
<code>tester) async {</code>	
<code> // ARRANGE: zbuduj widжет w środowisku testowym</code>	pumpWidget() = wyrenderuj widжет w symulowanym środowisku
<code> await tester.pumpWidget(</code>	
<code> MaterialApp(</code>	Owiń w MaterialApp - większość widжетów tego wymaga (Theme, Directionality)
<code> home: WeatherCard(weather: testWeather),</code>	Testowany widжет z danymi testowymi
<code>),</code>	
<code>);</code>	
<code> // ACT + ASSERT w jednym kroku (dla prostych przypadków)</code>	
<code> expect(find.text('Warszawa'), findsOneWidget);</code>	find.text() = szukaj widgetu z tym tekstem. findsOneWidget = oczekuj dokładnie 1.
<code> });</code>	
<code> testWidgets('displays temperature with unit',</code>	
<code>(tester) async {</code>	
<code> await tester.pumpWidget(MaterialApp(home:</code>	
<code>WeatherCard(weather: testWeather)));</code>	
<code> // Sprawdź interpolowany string</code>	Sprawdź dokładny tekst - jeśli format się zmieni test się złamie (dobra rzecz!)
<code> expect(find.text('22.5°C'), findsOneWidget);</code>	
<code> });</code>	
<code> }</code>	

6.2. API testów widжетów: Finders, Matchers i akcje

Finders lokalizują widgety w drzewie (np. po typie, tekście). Matchers definiują warunki, które muszą zostać spełnione (np. obecność, liczba instancji). API udostępnia również akcje symulujące interakcje użytkownika (tap, enterText). Kombinacja tych narzędzi umożliwia pełną kontrolę nad testowanym UI.

API	Opis i przykład
<code>find.text('tekst')</code>	Znajdź widжет(y) zawierający(e) ten tekst. Domyślnie: dokładne dopasowanie. <code>find.textContaining('War')</code> dla partial.
<code>find.byType(WeatherCard)</code>	Znajdź widжет o danym typie. Użyj gdy szukasz własnych widжетów.

API	Opis i przykład
<code>find.byKey(Key('weather-card'))</code>	Znajdź po Key. Najbardziej niezawodny selektor - jak <code>testTag</code> w Compose.
<code>find.byIcon(Icons.thermostat)</code>	Znajdź widжет <code>Icon</code> z daną ikoną.
<code>find.bySemanticsLabel('Temperatura')</code>	Znajdź po etykiecie semantycznej (jak <code>contentDescription</code> w Androidzie). Testuje dostępność.
<code>findsOneWidget</code>	Matcher: oczekuj dokładnie 1 widжет. Najczęstszy.
<code>findsNothing</code>	Matcher: oczekuj że widжет NIE istnieje. Dla stanów ukrytych.
<code>findsNWidgets(n)</code>	Matcher: oczekuj dokładnie n widжетów. Dla list.
<code>findsAtLeastNWidgets(1)</code>	Matcher: oczekuj co najmniej n widжетów.
<code>await tester.tap(finder)</code>	Akcja: tapnij znaleziony widжет. Jak <code>performClick()</code> w Compose Testing.
<code>await tester.enterText(finder, 'tekst')</code>	Akcja: wpisz tekst w <code>TextField</code> .
<code>await tester.pump()</code>	Obróć jeden klatka animacji / przetwórz pending callbacki. Użyj po każdej akcji.
<code>await tester.pumpAndSettle()</code>	Poczekaj aż wszystkie animacje i Futurey się skończą. Jak <code>waitUntilIdle()</code> w Compose.

6.3. Testowanie interakcji i asynchroniczności

Operacje asynchroniczne wymagają synchronizacji testu z cyklem renderowania (`pump`, `pumpAndSettle`). Pozwala to na oczekiwanie zakończenia animacji lub operacji sieciowych. Interakcje użytkownika są symulowane sekwencyjnie, a stan UI jest weryfikowany po każdej zmianie. Należy unikać zależności od czasu rzeczywistego.

search_screen_test.dart - testy interakcji	Wyjaśnienie
<code>testWidgets('search triggers weather load', (tester) async {</code>	Test interakcji: wpisz → submit → sprawdź efekt
<code> // Arrange: przygotuj fake provider Riverpoda</code>	
<code> await tester.pumpWidget(</code>	ProviderScope na szczycie - wymagane dla Riverpoda
<code> ProviderScope(</code>	
<code> overrides: [</code>	overrides = podmień providery na testowe wersje
<code> // Podmień serwis pogody na fake</code>	
<code> weatherServiceProvider.overrideWithValue(</code>	overrideWithValue() = podaj konkretną instancję
<code> FakeWeatherService(),</code>	FakeWeatherService = implementacja testowa bez HTTP
<code>),</code>	
<code>],</code>	
<code> child: const MaterialApp(home: HomeScreen()),</code>	
<code>),</code>	
<code>);</code>	
<code> // Act: znajdź TextField i wpisz miasto</code>	
<code> await tester.enterText(</code>	enterText = wpisz text w znalezionym TextField
<code> find.byKey(const Key('search-field')),</code>	Szukamy po Key (ustawiony w SearchBar Modifier.key)
<code> 'Gdańsk',</code>	Wpisywany tekst
<code>);</code>	
<code> await</code>	
<code> tester.testTextInput.receiveAction(TextInputAction.search);</code>	Symuluj wciśnięcie klawisza Search na klawiaturze
<code> // pumpAndSettle czeka na Future (ładowanie danych)</code>	
<code> await tester.pumpAndSettle();</code>	Czekaj aż FutureProvider zakończy i UI się zaktualizuje.
<code> // Assert: sprawdź że wynik się pojawił</code>	

search_screen_test.dart - testy interakcji	Wyjaśnienie
<code>expect(find.text('Gdańsk'), findsOneWidget);</code>	Nazwa miasta widoczna w wynikach
<code>expect(find.text('Ładowanie...'), findsNothing);</code>	Spinner już zniknął
<code>expect(find.byType(WeatherCard), findsOneWidget);</code>	Karta pogody widoczna
<code>});</code>	

6.4. Uruchamianie testów

Testy są wykonywane przez narzędzie CLI (`flutter test`). System automatycznie wykrywa pliki testowe i raportuje wyniki. Możliwe jest uruchamianie pojedynczych testów lub całych zestawów. Wyniki obejmują statystyki oraz szczegóły błędów, co wspiera proces debugowania.

flutter test; uruchamianie testów
<pre># Uruchom wszystkie testy flutter test # Uruchom konkretny plik: flutter test test/ui/widgets/weather_card_test.dart # Uruchom z verbose output (nazwa każdego testu) flutter test --reporter expanded # Uruchom z coverage (raport HTML) flutter test --coverage genhtml coverage/lcov.info -o coverage/html open coverage/html/index.html # macOS/Linux # Filtruj po nazwie testu (--name = regex) flutter test --name 'shows city name' # Obserwuj i uruchamiaj przy zmianach (jak --watch w Jest) flutter test --watch # wymaga flutter SDK >= 3.13</pre>

7. Mocktail - mockowanie zależności

W testach jednostkowych i widgetów często potrzebujemy zastąpić prawdziwe implementacje (serwis HTTP, baza danych) kontrolowanymi obiektami testowymi. Mocktail to biblioteka mockująca dla Dart - odpowiednik MockK z ekosystemu Kotlin.

7.1. Konfiguracja Mocktail

Mocktail umożliwia tworzenie mocków bez generowania kodu w czasie kompilacji. Wymaga rejestracji wartości domyślnych (`registerFallbackValue`) dla typów używanych w metodach mockowanych. Mock definiuje się poprzez dziedziczenie po `Mock` i implementację interfejsu zależności. Zachowanie jest konfigurowane przez `when(...).thenReturn(...)` lub `thenAnswer(...)`. Weryfikacja interakcji odbywa się przez `verify(...)`. Podejście to eliminuje boilerplate, lecz wymaga jawnego definiowania fallbacków dla typów złożonych.

pubspec.yaml - Mocktail
<code>dev_dependencies:</code>
<code>mocktail: ^1.0.4 # Mockowanie dla Dart (lepsze od mockito dla null safety)</code>
<code># Mocktail nie wymaga generowania kodu (w odróżnieniu od mockito).</code>
<code># Działa przez dziedziczenie i manual stubs.</code>

7.2. Tworzenie mocków i stubów

Mocki symulują zachowanie zależności poprzez kontrolowane odpowiedzi, natomiast stuby dostarczają statycznych danych. W Dart stosuje się biblioteki generujące klasy mockujące na podstawie interfejsów. Podejście to umożliwia izolację testowanej jednostki od zewnętrznych systemów (np. API), co zwiększa deterministyczność testów.

weather_provider_test.dart - Mocktail	Wyjaśnienie
<code>import 'package:mocktail/mocktail.dart';</code>	Import Mocktail
<code>// 1. Zdefiniuj klasę mocka - extends Mock + implementuj interfejs</code> <code>class MockWeatherService extends Mock implements</code> <code>WeatherService {}</code>	Jak class MockRepo extends MockK w MockK (Kotlin) Mock 'pusty' - wszystkie metody domyślnie rzucają wyjątek
<code>// 2. Zarejestruj fallback values dla typów niestandardowych</code> <code>setUpAll(() {</code> <code> // jeśli potrzebne</code> <code> registerFallbackValue(FakeWeatherData());</code> <code>});</code>	Wymagane gdy Mocktail nie zna typu argumentu setUpAll = raz przed wszystkimi testami (jak @BeforeAll w JUnit 5) Zarejestruj 'pusty' obiekt jako domyślny dla any()
<code>void main() {</code> <code> late MockWeatherService mockService;</code> <code> late WeatherNotifier notifier;</code>	late = inicjalizacja w setUp()
<code> setUp(() {</code> <code> mockService = MockWeatherService();</code> <code> notifier = WeatherNotifier(service: mockService);</code> <code> });</code>	setUp = przed każdym testem (jak @BeforeEach w JUnit 5) Nowy mock dla każdego testu - izolacja Wstrzyknij mock przez konstruktor
<code> test('fetch sets loading then data', () async {</code> <code> // STUB: zdefiniuj co mock zwróci</code> <code> when(() => mockService.fetchWeather('Kraków'))</code> <code> .thenAnswer((_) async => fakeWeatherData);</code>	test() = test jednostkowy (bez widżetu) when().thenAnswer() = jak coEvery {} returns w MockK when() + strzałka = identyfikacja wywoływanej metody thenAnswer() dla async. thenReturn() dla sync.
<code> // ACT: wywołaj metodę notifiersa</code> <code> await notifier.fetch('Kraków');</code>	
<code> // ASSERT: stan po operacji</code> <code> expect(notifier.state, isA<AsyncData<WeatherData>>());</code>	isA<T>() = sprawdź typ. AsyncData = sukces w Riverpod.
<code> // VERIFY: czy metoda została wywołana?</code> <code> verify(() =></code> <code> mockService.fetchWeather('Kraków')).called(1);</code> <code> verifyNever(() => mockService.fetchWeather('Gdańsk'));</code> <code> });</code> <code>}</code>	verify() = sprawdź interakcję (jak coVerify w MockK) called(1) = wywołana dokładnie 1 raz verifyNever = nigdy nie wywołana z tym argumentem

8. Architektura projektu - Feature-First i Clean Architecture

W Flutter Lab 1 stosowaliśmy prostą strukturę lib/ z katalogami data/, providers/, ui/. To podejście działa dla małych projektów. Przy rozroście aplikacji lepsza jest architektura Feature-First - każda funkcjonalność ma swój własny, samodzielny moduł.

8.1. Layer-first vs Feature-first

Struktura layer-first organizuje kod według warstw (UI, data, domain), co sprzyja separacji odpowiedzialności, lecz utrudnia lokalizację funkcjonalności. Feature-first grupuje kod według funkcji (np. „weather”), co zwiększa modularność i ułatwia skalowanie zespołów. W praktyce stosuje się podejścia hybrydowe.

Layer-first (Lab 1)	Feature-first (rekomendowany)
lib/data/, lib/providers/, lib/ui/ grupowanie po typie technicznym. Dobrze dla < 5 ekranów. Problemy: plik weather_provider.dart jest 'sąsiadem' cart_provider.dart mimo że to różne funkcje.	lib/features/weather/, lib/features/favorites/ - grupowanie po funkcji biznesowej. Każda feature jest samowystarczalna: ma własne data, providers, ui.
Zmiana jednej funkcji dotyka wielu katalogów (data/, providers/, ui/). Naukowiec, który czyta kod, musi skakać między katalogami.	Zmiana jednej funkcji odbywa się w jednym miejscu (features/weather/). Naukowiec widzi od razu co należy do 'pogody'.

8.2. Struktura Feature-First dla WeatherApp v2

Kod jest podzielony na moduły funkcjonalne (np. weather, favorites). Każdy moduł zawiera własne modele, serwisy, widgety i logikę stanu. Struktura ta minimalizuje zależności między modułami i umożliwia ich niezależny rozwój. Wspólne komponenty są wydzielone do katalogów współdzielonych (core, shared).

Struktura Feature-First projektu WeatherApp v2	
lib/	
├─ main.dart	
├─ core/	← Współdzielone przez wszystkie features
│ └─ error/	
│ └─ exceptions.dart	← Własne klasy wyjątków (WeatherException itp.)
│ └─ failures.dart	← Sealed class Failure (NetworkFailure, CacheFailure)
│ └─ network/	
│ └─ api_client.dart	← Wspólny http.Client z interceptorami
│ └─ router/	
│ └─ app_router.dart	← GoRouter i cała nawigacja
│ └─ theme/	
│ └─ app_theme.dart	← ThemeData light i dark
├─ features/	
│ └─ weather/	← Feature: aktualna pogoda
│ └─ data/	
│ └─ models/	
│ └─ weather_dto.dart	← DTO: parsowanie JSON z API
│ └─ weather_data.dart	← Domain model + Hive @HiveType
│ └─ services/	
│ └─ weather_api_service.dart	← Wywołania HTTP
│ └─ repositories/	
│ └─ weather_repository.dart	← łączy API + cache Hive
│ └─ providers/	
│ └─ weather_provider.dart	← AsyncNotifierProvider
│ └─ weather_service_provider.dart	← Provider<WeatherRepository>
│ └─ ui/	
│ └─ screens/	
│ └─ detail_screen.dart	
│ └─ widgets/	
│ └─ weather_card.dart	
│ └─ forecast_tile.dart	
│ └─ favorites/	← Feature: ulubione miasta
│ └─ data/	
│ └─ repositories/	
│ └─ favorites_repository.dart	← Hive Box<String>
│ └─ providers/	
│ └─ favorites_provider.dart	← NotifierProvider
│ └─ ui/	
│ └─ screens/	
│ └─ home_screen.dart	
│ └─ widgets/	
│ └─ city_tile.dart	
└─ shared/	← Widgety używane w wielu features
└─ widgets/	
└─ loading_widget.dart	
└─ error_widget.dart	

8.3. Repository Pattern - warstwa między API a UI

Repository stanowi abstrakcję nad źródłami danych (API, lokalna baza). UI komunikuje się z repository zamiast bezpośrednio z serwisami. Warstwa ta agreguje dane, implementuje cache oraz logikę fallback. Umożliwia to łatwą zmianę źródła danych oraz upraszcza testowanie poprzez mockowanie repository.

weather_repository.dart; offline-first z Hive + API	Wyjaśnienie
<code>class WeatherRepository {</code>	Repository = orchestrator: API + cache. UI nie wie 'skąd' dane.
<code> final WeatherApiService _api;</code>	Wstrzyknięte przez konstruktor (dependency injection)
<code> final WeatherCacheService _cache;</code>	
<code>const WeatherRepository({</code>	named parameters - czytelne przy tworzeniu
<code> required WeatherApiService api,</code>	
<code> required WeatherCacheService cache,</code>	
<code>}) : _api = api, _cache = cache;</code>	: _api = api = inicjalizacja pola w liście inicjalizacyjnej
<code>Future<WeatherData> getWeather(String city) async {</code>	Główna metoda - cache-first
<code> // KROK 1: Sprawdź czy cache jest świeży (< 30 min)</code>	Offline-first: najpierw spróbuj lokalnie
<code> if (_cache.isFresh(city)) {</code>	
<code> return _cache.getCached(city)!; // z cache</code>	! bezpieczne po isFresh() - gwarantuje non-null
<code> }</code>	
<code> // KROK 2: Pobierz z API i zapisz do cache</code>	Sieć: tylko gdy cache nieaktualne
<code> try {</code>	
<code> final data = await _api.fetchWeather(city);</code>	
<code> await _cache.cacheWeather(data);</code>	Zapisz do Hive - następne wywołanie pobierze z cache
<code> return data;</code>	
<code> } catch (e) {</code>	
<code> // KROK 3: Błąd sieci - zwróć stary cache jeśli istnieje</code>	Graceful degradation: stare dane lepsze niż błąd
<code> final stale = _cache.getCached(city);</code>	getCached() = zwróć nieaktualne dane jeśli są
<code> if (stale != null) return stale;</code>	Zwróć nawet stare dane (lepsza UX niż crash)
<code> rethrow; // Brak cache - propaguj błąd do UI</code>	rethrow = rzuć ten sam wyjątek dalej (nie trać stack trace)
<code> }</code>	
<code>}</code>	
<code>}</code>	

9. Dostępność i internacjonalizacja

Profesjonalna aplikacja mobilna musi być dostępna dla użytkowników z niepełnosprawnościami oraz działać w wielu językach. Flutter zapewnia wbudowane mechanizmy dla obu tych wymagań.

9.1. Dostępność, Semantics widget

Flutter generuje drzewo semantyczne automatycznie dla standardowych widжетów (Text, ElevatedButton, TextField). Dla niestandardowych widжетów i gestów musimy je opisać ręcznie przez widжет Semantics.

Semantics; dostępność widgetu pogody	Wyjaśnienie
<code>// Karta temperatury z pełną semantyką TalkBack/VoiceOver</code>	
<code>Semantics(</code>	Semantics = opisuje widжет dla czytnika ekranu
<code> label: 'Temperatura w Warszawie: 22.5 stopnia Celsjusza',</code>	label = co czytnik ekranu przeczyta. Pełny opis, nie skrót.
<code> hint: 'Dwukrotnie stuknij, aby zobaczyć prognozę',</code>	hint = instrukcja dla użytkownika. Co się stanie po tapnięciu.
<code> button: true,</code>	button: true = czytnik ogłasza 'przycisk' i pozwala aktywować
<code> onTap: () => context.push('/detail/Warszawa'),</code>	onTap w Semantics = jak onPressed - dostępny dla assistive tech
<code> child: Container(</code>	Owiń widжет który nie ma sam semantyki
<code> child: Row(children: [</code>	
<code> // excludeSemantics: true = ukryj przed czytnikiem</code>	

Semantics; dostępność widżetu pogody	Wyjaśnienie
<pre>ExcludeSemantics(child: Icon(Icons.thermostat, size: 32),), Text('22.5°C', style: ...),]),),),</pre>	ExcludeSemantics = ikona dekoracyjna, czytnik pomija Ikona jest dekoracyjna - label Semantics już opisuje wszystko Text ma automatyczną semantykę - czytnik przeczyta '22.5°C'

9.2. Internacjonalizacja (i18n); flutter_localizations

Internacjonalizacja umożliwia dostosowanie aplikacji do różnych języków i regionów. Flutter wykorzystuje system lokalizacji oparty na plikach ARB oraz delegatach (LocalizationsDelegate). Pakiet flutter_localizations dostarcza podstawowe tłumaczenia komponentów systemowych. Mechanizm ten wspiera dynamiczne przełączanie języka oraz formatowanie danych zgodnie z lokalizacją użytkownika.

pubspec.yaml - i18n	YAML
<pre>dependencies: flutter_localizations: # wbudowana w Flutter SDK - bez wersji sdk: flutter intl: ^0.19.0 # Formatowanie liczb, dat, walut flutter: generate: true # Włącz generator plików .arb → .dart</pre>	

lib/l10n/app_pl.arb - polskie tłumaczenia	ARB
<pre>{ "@@locale": "pl", "appTitle": "Pogodynka", "searchHint": "Wpisz nazwę miasta...", "temperatureLabel": "Temperatura: {temp}°C", "@temperatureLabel": { "placeholders": { "temp": { "type": "double", "format": "decimalPattern" } } }, "humidity": "Wilgotność: {value}%", "@humidity": { "placeholders": { "value": { "type": "int" } } } } # lib/l10n/app_en.arb - angielskie tłumaczenia { "@@locale": "en", "appTitle": "Weather App", "searchHint": "Enter city name...", "temperatureLabel": "Temperature: {temp}°C", "humidity": "Humidity: {value}%" } # Generuj klasy Dart z plików .arb: flutter gen-l10n # Użycie w kodzie (po generacji): # Text(AppLocalizations.of(context)!.appTitle) # Text(AppLocalizations.of(context)!.temperatureLabel(22.5))</pre>	

main.dart - konfiguracja lokalizacji w MaterialApp	DART
--	------

```

MaterialApp.router(
  // Delegaci lokalizacji - MUSZĄ być ustawione
  localizationsDelegates: const [
    AppLocalizations.delegate,           // Własne tłumaczenia (generowane)
    GlobalMaterialLocalizations.delegate, // Material widgets (daty, przyciski OK/Cancel)
    GlobalWidgetsLocalizations.delegate,  // Podstawowe widżety (kierunek tekstu LTR/RTL)
    GlobalCupertinoLocalizations.delegate, // iOS-style widgets
  ],
  // Obsługiwane języki
  supportedLocales: const [
    Locale('pl', 'PL'), // Polski (Polska)
    Locale('en', 'US'), // Angielski (USA)
  ],
  // Domyślna lokalizacja gdy system nie pasuje do żadnej obsługiwanej
  locale: const Locale('pl', 'PL'),
  routerConfig: appRouter,
)

```

10. Flutter DevTools - diagnostyka i profilowanie

Flutter DevTools to zaawansowane narzędzie debugowania uruchamiane w przeglądarce. Dostarcza wizualizację drzewa widżetów na żywo, narzędzie do profilowania wydajności, inspektor sieci i debugger pamięci. Znajomość DevTools jest niezbędna przy optymalizacji aplikacji.

#	Uruchomienie Flutter DevTools
1	Uruchom aplikację w trybie debug: flutter run (lub przez Android Studio przycisk Debug).
2	W terminalu pojawi się URL DevTools, np.: 'An Observatory debugger and profiler on...'. Wpisz w przeglądarce lub kliknij link.
3	Alternatywnie: w Android Studio otwórz View → Tool Windows → Flutter Inspector. DevTools otworzy się w panelu bocznym.
4	Możesz też uruchomić DevTools niezależnie: dart devtools - otworzy się serwer, wklej URL aplikacji.

Panel DevTools	Do czego służy
Flutter Inspector	Drzewo widżetów na żywo. Zaznaczaj widżety w aplikacji i inspektor pokaże je w drzewie. Wyświetla rozmiary, padding, margin. Poszukaj 'Debug Paint' - pokaże granice widżetów.
Performance	Timeline renderowania - każda klatka, czas CPU i GPU. Szukaj czerwonych/żółtych klatek (> 16ms). Nagrywaj profile dla konkretnych akcji.
CPU Profiler	Nagrywanie call stack w czasie. Sprawdź co zajmuje czas CPU podczas ładowania danych lub renderowania listy.
Memory	Użycie pamięci w czasie. Wykrywanie wycieków pamięci - obserwuj, czy pamięć rośnie po wielokrotnych akcjach bez powrotu do baseline.
Network	Wszystkie żądania HTTP. Czas odpowiedzi, kody statusu, nagłówki, ciało odpowiedzi. Idealne do debugowania API calls bez Postmana.
Logging	Logi aplikacji z filtrami. Wyświetla print(), debugPrint() i wyjątki. Lepsze niż terminal gdy jest wiele logów.
App Size	Analiza rozmiaru APK/IPA. Pokaże co zajmuje miejsce (assets, paczki, kod). Uruchom przez: flutter build apk --analyze-size.

Repaint Rainbow; znajdź zbędne przebudowania widжетów

W Flutter Inspector włącz 'Highlight Repaints' (przełącznik u góry). Widgety które się przebudowują będą migać kolorowymi ramkami. Częste miganie = zbędny rebuild - optymalizuj przez:

1. const konstruktory na widжетach które się nie zmieniają.
2. Zmiana watch() na select() w Riverpod - subskrybuj tylko część stanu.
3. Wydziel często zmieniający się widget do osobnej klasy (mniejszy zakres rebuilda).

11. Zadania do wykonania

Zadania są kontynuacją projektu WeatherApp z Flutter Lab 1. Kod z Lab 1 służy jako punkt startowy. Migrujemy go do nowej architektury i rozbudowujemy o nowe funkcje.

Zadanie 1 - Migracja na Riverpod i Hive

- 1.1 Dodaj paczki do pubspec.yaml: flutter_riverpod, hive, hive_flutter, hive_generator, build_runner.
- 1.2 Owiń aplikację w ProviderScope w main.dart. Zainicjalizuj Hive przez Hive.initFlutter() przed runApp().
- 1.3 Dodaj adnotacje @HiveType i @HiveField do klasy WeatherData. Uruchom build_runner, zarejestruj wygenerowany adapter w main.dart, otwórz Box<WeatherData>('weatherCache').
- 1.4 Zmigruj WeatherProvider z ChangeNotifier (Provider) na AsyncNotifierProvider (Riverpod).
Zmigruj FavoritesProvider na NotifierProvider<FavoritesNotifier, List<String>>.
- 1.5 Zmień HomeScreen i DetailScreen: zastąp Consumer<T> przez ConsumerWidget (WidgetRef ref).

WERYFIKACJA: Aplikacja działa identycznie jak po Lab 1. Brak importów package:provider.

Zadanie 2 - Offline-first z Hive i Repository Pattern

- 2.1 Wdrożenie WeatherCacheService: metody cacheWeather(), getCached(), isFresh() (30 min TTL), remove().
- 2.2 Wdrożenie WeatherRepository łączącego API + cache: cache-first, fallback na stary cache przy błędzie sieci. Provider<WeatherRepository> zarejestrowany w Riverpodzie.
- 2.3 Zmodyfikuj WeatherNotifier aby używał WeatherRepository (ref.read(repositoryProvider)) zamiast bezpośrednio serwisu.
- 2.4 Przetestuj offline-first: załaduj pogodę → wyłącz internet → uruchom ponownie aplikację → dane nadal widoczne.
- 2.5 Wyświetl w DetailScreen znacznik czasu ostatniej aktualizacji danych (weather.updatedAt, sformatowany przez intl).

WERYFIKACJA: Screenshot z danymi widocznymi przy wyłączonym locie samolotowym. Timestamp widoczny w UI.

Zadanie 3 - Animacje

- 3.1. AnimatedContainer na karcie miasta w HomeScreen: tapnięcie rozszerza kartę (h: 80→180dp) z efektem animowanego koloru tła. Czas: 350ms, Curves.easeInOut.
- 3.2. AnimatedSwitcher dla ikony pogody w WeatherCard: zmiana weatherCode animuje ikonę (ScaleTransition + FadeTransition). Pamiętaj o ValueKey(weatherCode).
- 3.3. Hero między HomeScreen a DetailScreen: ikona pogody (64×64 na liście, 128×128 w szczegółach). Tag: 'weather-icon- $\$$ city'. Sprawdź płynne przejście.
- 3.4. Animacja wejścia listy w HomeScreen: ListView z animowanym pojawianiem się kart (SlideTransition + FadeTransition). Każda karta przesuwa się od prawej ze staggered delay (karta_n × 50ms opóźnienia).

WERYFIKACJA: Demo animacji na AVD przy 60fps (flutter run --release dla lepszego pomiaru).

Zadanie 4 - Testy widжетów i jednostkowe

- 4.1. Napisz min. 4 testy jednostkowe (flutter test, bez widжетów): WeatherData.fromJson() dla poprawnych danych, pustego JSON, JSON z nullem w wymaganym polu, przeliczenie temperatury z K na °C.

- 4.2. Napisz min. 3 testy widgetu WeatherCard (testWidgets): wyświetlanie nazwy miasta, temperatury z jednostką, ikony pogody dla weatherCode == 0 (słonecznie).
- 4.3. Napisz test interakcji SearchBar: wpisz 'Gdańsk' → wciśnij Search → sprawdź, że FakeWeatherService.fetchWeather() wywołane z 'Gdańsk'.
- Użyj Mocktail do mockowania WeatherService i ProviderScope(overrides:[]) do wstrzyknięcia mocka.
- 4.4. Uruchom flutter test -coverage. Uzyskaj pokrycie > 60% dla pliku weather_card.dart.
- WERYFIKACJA: flutter test uruchamia się bez błędów. Raport coverage w coverage/html/index.html.

12. Kryteria oceniania

Zadanie	Punkty	Co weryfikuje prowadzący
Zad. 1: Riverpod + Hive	20 pkt	Brak package:provider. ProviderScope. Hive inicjalizacja + adapter. AsyncNotifierProvider i NotifierProvider działają.
Zad. 2: Offline-first	25 pkt	Dane widoczne przy wyłączonym internecie. Timestamp w UI. Repository Pattern (API + cache oddzielone).
Zad. 3: Animacje	30 pkt	AnimatedContainer (expand). AnimatedSwitcher z ValueKey. Hero między ekranami. Staggered list entry.
Zad. 4: Testy	25 pkt	Min. 7 testów (4 jednostkowe + 3 widgetów). Mocktail mock. Coverage > 60% dla WeatherCard.
RAZEM	100 pkt	

12.1. Skala ocen

Ocena	Punkty / Wymagania
5.0	90-100 pkt - Wszystkie zadania. Animacje płynne w release. Coverage > 70%. Mocktail z verify().
4.5	80-89 pkt - Zadania 1-3 kompletne. Min. 5 testów. Hero działa. Offline-first.
4.0	70-79 pkt - Zadania 1-2 + AnimatedContainer i AnimatedSwitcher. Min. 3 testy widgetu.
3.5	60-69 pkt - Zadania 1-2. Riverpod i Hive działają. Brak animacji. Min. 4 testy jednostkowe.
3.0	50-59 pkt - Zadanie 1: migracja na Riverpod działa. Dane z Hive persystują.
2.0	0-49 pkt - Projekt nie kompiluje lub Riverpod nie działa.

13. Najczęstsze błędy i ich rozwiązania

Błąd / Objaw	Przyczyna i rozwiązanie
MissingProviderScopeException / ProviderNotFoundException	Brak ProviderScope w drzewie widgetów. Owiń całą aplikację: runApp(ProviderScope(child: MyApp())).
HiveError: 'Box not found. Did you forget to call Hive.openBox?'	Box nie został otwarty przed użyciem. Hive.openBox() MUSI być wywołane w main() przed runApp(). Nie otwieraj Box w konstruktorze klasy serwisu.
Bad state: already registered typeId 0	Dwa adaptory mają ten sam typeId. Każda klasa @HiveType musi mieć unikalny typeId (0, 1, 2...). Sprawdź wszystkie @HiveType w projekcie.
AnimatedSwitcher nie animuje przy zmianie child	Brak unikalnego Key w child. Dodaj ValueKey(unikalnaWartość) do każdego child AnimatedSwitcher.
Hero: 'There are multiple heroes that share the same tag'	Dwa widgety Hero z tym samym tagiem na tym samym ekranie. Użyj unikalnych tagów: 'weather-icon-\$cityName' zamiast 'weather-icon'.
AnimationController nie zwalnia pamięci (wyciek)	Brak _controller.dispose() w metodzie dispose() State. Zawsze: @override void dispose() { _controller.dispose(); super.dispose(); }
ref.watch() throws ProviderException poza build()	ref.watch() wywołane w initState(), onPressed() lub async callback. Zamień na ref.read() dla akcji poza build().

Błąd / Objaw	Przyczyna i rozwiązanie
Mocktail: MissingStubError - no stub for method	Metoda mocka wywołana bez wcześniejszego when(). Dodaj when(() => mock.method()).thenAnswer((_) async => result) przed wywołaniem kodu testowanego.
flutter test - Cannot find widget with type X	Widget nie jest wyrenderowany w testowym drzewie. Sprawdź czy pumpWidget() zawiera MaterialApp jako korzeń i czy widget jest strukturze children.
Hive: 'Cannot write to a closed box'	Box zamknięty przed zapisem lub operacja po dispose(). Upewnij się że Box jest otwarty przez cały czas życia aplikacji.

Debugging Riverpod - ProviderObserver

Riverpod umożliwia obserwowanie wszystkich zmian stanu przez ProviderObserver. Dodaj do ProviderScope(observers: [LoggingObserver()],) gdzie:

```
class LoggingObserver extends ProviderObserver {
  @override
  void didUpdateProvider(provider, prev, next, container) {
    debugPrint('[Riverpod] ${provider.name}: $prev → $next');
  }
}
```

Flutter 2 - Riverpod, Animacje, Hive i Testy Widgetów

Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki

MobileHub

Następne ćwiczenie: Flutter 3 - Publikacja aplikacji, CI/CD z GitHub Actions, Flavors

ĆWICZENIE LABORATORYJNE - FLUTTER 3

Publikacja · CI/CD · Flavors

Programowanie Aplikacji Mobilnych - moduł Flutter

Projekt: WeatherApp v3 - rozszerzenie projektu z Flutter Lab 2

Wymagania wstępne: ukończony Flutter Lab 2 (Riverpod, Hive, animacje, testy widżetów)

Spis treści

1. Czym jest Flavor i dlaczego każda aplikacja go potrzebuje?	50
2. Konfiguracja Flavor-ów w Flutter	51
3. Podpisywanie aplikacji, czyli Keystore i upload key	54
4. Budowanie artefaktów; APK vs App Bundle	56
5. Od konta dewelopera do publikacji na Google Play Store	57
6. Automatyzacja potoku, CI/CD z GitHub Actions	58
7. Automatyzacja publikacji z Fastlane	61
8. Zarządzanie sekretami i zmiennymi środowiskowymi	63
9. Monitorowanie przez Firebase Crashlytics i Analytics	64

1. Czym jest Flavor i dlaczego każda aplikacja go potrzebuje?

Wyobraź sobie, że pracujesz nad WeatherApp. Masz dwie wersje: jedna łączy się z prawdziwym API pogodowym (produkcja), druga z serwerem testowym (development). No i łatwo tu o problem. Jak jednocześnie mieć obie na telefonie, nie mylić kluczy API i nie wypychać kodu testowego do użytkowników? Odpowiedzią są Flavors (znane również jako Product Flavors w świecie Androida lub Build Configurations w iOS/Xcode). To mechanizm budowania wielu wariantów tej samej aplikacji z jednej bazy kodu.

Analogia jako kuchnia restauracyjna

Wyobraź sobie restaurację, która serwuje ten sam przepis na zupę w trzech wersjach: wegetariańskiej, mięsnej i bezglutenowej. Składniki są podobne, ale kilka kluczowych się różni. Szef kuchni nie pisze trzech odrębnych przepisów, ma jeden bazowy i adnotuje różnice. Flavors w Flutterze działają identycznie: jeden kod bazowy, kilka zestawów konfiguracji.

1.1. Trzy środowiska aplikacji mobilnej

Profesjonalna aplikacja mobilna zwykle ma trzy środowiska. Typowy podział obejmuje środowiska: development, staging oraz production. Development służy do lokalnego testowania i eksperymentów, staging odwzorowuje środowisko produkcyjne (np. testowe API), natomiast production jest wersją końcową dla użytkownika. Każde środowisko może różnić się konfiguracją endpointów, kluczy API, logowania oraz poziomu debugowania.

Flavor / środowisko	Cel	Cechy charakterystyczne	Pakiet (Android)
development (dev)	Aktywny development, testy funkcji	Ładny banner DEBUG, szczegółowe logi, mock lub testowe API, HMR/Hot Reload	<i>pl.edu.pam.weather.dev</i>
staging	Testy QA, testy akceptacyjne klienta	Produkcyjny wygląd, testowe API, dane testowe, instalacja obok prod	<i>pl.edu.pam.weather.staging</i>
production (prod)	Wersja dla użytkowników w Play Store	Brak logów debug-owych, prawdziwe API, zaciemniony kod (obfuscation)	<i>pl.edu.pam.weather</i>

Dlaczego różne identyfikatory pakietów?

Na urządzeniu Android mogą współistnieć dwie aplikacje o tym samym nazewnictwie tylko wtedy, gdy mają różne package names. Używając `pl.edu.pam.weather.dev` i `pl.edu.pam.weather` możesz jednocześnie testować obie wersje na jednym telefonie.

1.2. Co Flavor zmienia, a co pozostaje takie samo?

Flavor wprowadza warianty builda, które różnią się konfiguracją aplikacji (np. nazwa, identyfikator pakietu, API endpoint). Kod źródłowy pozostaje w większości wspólny, natomiast zmienne środowiskowe i zasoby mogą być specyficzne dla danego wariantu. Mechanizm ten pozwala utrzymać jedną bazę kodu przy jednoczesnym wsparciu wielu konfiguracji runtime.

Co zmienia Flavor	Co pozostaje IDENTYCZNE
Identyfikator pakietu (package name / bundle ID)	Cały kod Dart (logika, widżety, UI)
Nazwa aplikacji wyświetlana pod ikoną	Zależności w pubspec.yaml
Ikoną aplikacji i splash screen	Struktura katalogów projektu
Klucz API, URL backendu, klucze Firebase	Testy jednostkowe i widgetów
Podpisywanie (keystore / upload key)	Logika biznesowa, modele danych
Poziom logowań, sentry DSN, analytics key	Paczki / pluginy Flutter

2. Konfiguracja Flavor-ów w Flutter

Konfiguracja Flavorów we Flutterze dzieli się na dwie części: stronę Dart (punkt wejścia aplikacji, klasa konfiguracji) i stronę natywną (Android Gradle, iOS Xcode schemes). Zaczniemy od strony Dart, bo ona kontroluje logikę, a nie tylko budowanie.

2.1. Klasa konfiguracji AppConfig

Zamiast hardkodować URL API i klucze w kodzie, tworzymy klasę `AppConfig`, która przechowuje wszystkie wartości zależne od środowiska. Jest inicjalizowana PRZED uruchomieniem aplikacji. `AppConfig` enkapsuluje parametry środowiskowe, takie jak adres API, flagi feature'ów czy tryb debugowania. Instancja konfiguracji jest inicjalizowana na starcie aplikacji i udostępniana globalnie (np. przez provider). Centralizacja konfiguracji upraszcza zarządzanie środowiskami i redukuje rozproszenie stałych w kodzie.

Wzorzec

Singleton z inicjalizacją przed `runApp()`. `AppConfig` jest inicjalizowany raz w `main()`, zanim Flutter uruchomi drzewo widżetów. Wszystkie serwisy pobierają konfigurację z `AppConfig.instance`.

lib/core/config/app_config.dart	DART
<pre> enum AppFlavor { dev, staging, prod } class AppConfig { // Singleton - jedna instancja na caly czas zycia aplikacji static AppConfig? _instance; static AppConfig get instance { assert(_instance != null, 'AppConfig nie zostal zainicjalizowany!'); return _instance!; } // Pola konfiguracji final AppFlavor flavor; final String baseUrl; final String appName; final bool enableAnalytics; final bool showDebugBanner; final String sentryDsn; AppConfig._({ required this.flavor, required this.baseUrl, required this.appName, required this.enableAnalytics, required this.showDebugBanner, this.sentryDsn = '', }); // Gettery pomocnicze bool get isDev => flavor == AppFlavor.dev; bool get isStaging => flavor == AppFlavor.staging; bool get isProd => flavor == AppFlavor.prod; // Fabryki dla kazdego flavoru static void initDev() { _instance = AppConfig._(flavor: AppFlavor.dev, baseUrl: 'https://api-dev.openmeteo.test', appName: '[DEV] WeatherApp', enableAnalytics: false, // brak analityk w dev showDebugBanner: true, // czerwony banner DEBUG); } static void initStaging() { _instance = AppConfig._(flavor: AppFlavor.staging, baseUrl: 'https://api-staging.openmeteo.test', appName: '[STG] WeatherApp', enableAnalytics: false, showDebugBanner: false,); } static void initProd() { _instance = AppConfig._(flavor: AppFlavor.prod, baseUrl: 'https://api.open-meteo.com', // prawdziwe API appName: 'WeatherApp', enableAnalytics: true, // analytics wlaczone showDebugBanner: false, sentryDsn: 'https://xxx@sentry.io/yyy', // monitoring błędów); } } </pre>	

2.2. Osobne punkty wejścia: - main_*.dart

Kazdy flavor ma swój plik main_*.dart. To jest punkt wejścia uruchamiany przez flutter run --flavor. Oddzielne pliki wejściowe (np. main_dev.dart, main_prod.dart) umożliwiają inicjalizację aplikacji z różnymi konfiguracjami. Różnica między plikami jest minimalna - wywołanie innej metody init(). Każdy punkt wejścia ustawia odpowiedni flavor oraz przekazuje właściwą instancję AppConfig. Mechanizm ten pozwala kontrolować zachowanie aplikacji już na etapie startu procesu.

lib/main_dev.dart - punkt wejścia dla flavoru dev	DART
<pre> import 'package:flutter/material.dart'; import 'core/config/app_config.dart'; import 'app.dart'; // wspólna klasa MyApp void main() { AppConfig.initDev(); // ZAWSZE przed runApp() } </pre>	

```

    runApp(const MyApp());
  }

// lib/main_staging.dart
void main() { AppConfig.initStaging(); runApp(const MyApp()); }

// lib/main_prod.dart (lub main.dart dla kompatybilności)
void main() { AppConfig.initProd(); runApp(const MyApp()); }

// lib/app.dart - wspólna klasa MyApp (nie zmienia się między flavorami!)
class MyApp extends ConsumerWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    return MaterialApp.router(
      title: AppConfig.instance.appName,
      debugShowCheckedModeBanner: AppConfig.instance.showDebugBanner,
      theme: AppTheme.lightTheme,
      darkTheme: AppTheme.darkTheme,
      themeMode: ThemeMode.system,
      routerConfig: appRouter,
    );
  }
}

```

2.3. Konfiguracja Android - Gradle

Strona Dart to tylko połowa pracy. Android musi wiedzieć o flavorach, aby nadać im różne package names, ikony i nazwy. Gradle definiuje flavor poprzez `productFlavors` w pliku `build.gradle` (`android/app/build.gradle`). Każdy flavor może mieć własny `applicationId`, `suffix`, konfigurację builda oraz zasoby. System builda generuje warianty kombinujące flavor i typ builda (`debug/release`). Konfiguracja ta jest kluczowa dla poprawnej dystrybucji wielu wersji aplikacji.

android/app/build.gradle	gradle
<pre> android { // ... defaultConfig ... flavorDimensions += 'environment' // Dart side nazewnictwo wymiaru productFlavors { // Definicja 3 flavorow dev { dimension 'environment' applicationIdSuffix '.dev' // pl.edu.pam.weather.dev versionNameSuffix '-dev' resValue 'string', 'app_name', 'DEV WeatherApp' // Nazwa pod ikoną } staging { dimension 'environment' applicationIdSuffix '.staging' // pl.edu.pam.weather.staging versionNameSuffix '-staging' resValue 'string', 'app_name', 'STG WeatherApp' } prod { dimension 'environment' // Brak suffixu - bazowy package name: pl.edu.pam.weather resValue 'string', 'app_name', 'WeatherApp' } } buildTypes { release { signingConfig signingConfigs.release // Omówione w sekcji 3 minifyEnabled true // ProGuard / R8 - zaciemnianie kodu shrinkResources true // Usunięcie nieużywanych zasobów proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro' } } } </pre>	

```
}
}
}
```

2.4. Uruchamianie z flavorem

Uruchomienie aplikacji z konkretnym flavorem wymaga wskazania wariantu builda (np. przez CLI lub konfigurację IDE). Parametry builda determinują, który punkt wejścia oraz konfiguracja zostaną użyte. Proces ten umożliwia równoległe testowanie różnych środowisk bez zmiany kodu.

Uruchamianie konkretnego flavor-u

```
# Uruchamianie konkretnego flavoru (w terminalu):
flutter run --flavor dev -t lib/main_dev.dart
flutter run --flavor staging -t lib/main_staging.dart
flutter run --flavor prod -t lib/main_prod.dart

# Budowanie APK z flavorami:
flutter build apk --flavor dev -t lib/main_dev.dart --debug
flutter build apk --flavor prod -t lib/main_prod.dart --release

# Budowanie App Bundle (do Play Store):
flutter build appbundle --flavor prod -t lib/main_prod.dart --release

# Skrócony zapis (konfiguracja w pubspec.yaml z flutter_flavorizr):
flutter run -t lib/main_dev.dart --flavor dev # alias 'dev'
```

pro tip

Możesz dodać skrypty w pubspec.yaml (sekcja scripts) lub użyć paczki flutter_flavorizr, która automatycznie konfiguruje Gradle, Xcode i ikony dla każdego flavoru. Oszczędza to godziny ręcznej konfiguracji.

2.5. Różne ikony dla flavorów

Podstawowym sygnałem wizualnym flavoru są ikony. Ikony aplikacji mogą być zróżnicowane per flavor poprzez oddzielne zasoby graficzne. Ułatwia to identyfikację środowiska (aplikacja dev powinna być łatwa do odróżnienia od prod). Konfiguracja odbywa się na poziomie zasobów platformowych (Android/iOS), gdzie każdy flavor wskazuje własny zestaw ikon. Typowo stosuje się banner kolorowy (czerwony dla dev, pomarańczowy dla staging, bez baneru dla prod).

Podejście	Opis / kroki
Reczna podmiana ikon w android/app/src/{flavor}/res/	Skopiuuj android/app/src/main/res/ do android/app/src/dev/res/ i podmien ikony. Gradle automatycznie wybierze odpowiedni katalog src/{flavor}/ przy buildowaniu.
flutter_launcher_icons + flutter_flavorizr	Paczki automatyzujące. Generują ikony dla wszystkich flavorów z jednego pliku konfiguracyjnego w pubspec.yaml. Rekomendowane dla nowych projektów.
adaptive icons (Android 8+)	Oddzielny foreground i background. Pozwala na overlay banera DEV/STG na ikonie przez zmianę samego foreground. Wygląda profesjonalnie.

3. Podpisywanie aplikacji, czyli Keystore i upload key

Każda aplikacja opublikowana na Google Play musi być podpisana cyfrowo. Podpis kryptograficzny zapewnia, że aktualizacje aplikacji pochodzą od tego samego dewelopera. Jest to mechanizm bezpieczeństwa chroniący użytkowników przed podmienionymi APK.

Analogia do pieczęci notarialnej

Podpis cyfrowy aplikacji działa jak pieczęć notarialna na dokumencie. Każdy może zobaczyć dokument (APK), ale nie może dać mu pieczęci notarialnej bez klucza notariusza. Google Play sprawdza pieczęć przy każdej aktualizacji - jeśli jest inna niż przy pierwszym przesłaniu, aktualizacja jest odrzucana.

3.1. Dwa klucze w Google Play App Signing

Od 2021 roku Google Play wymaga korzystania z modelu podpisywania aplikacji „Play App Signing”. W tym modelu funkcjonują dwa klucze: upload key (klucz uploadu) i app signing key (klucz aplikacji). Klucz aplikacji jest przechowywany przez Google i używany do podpisywania finalnych artefaktów. Klucz uploadu służy deweloperowi do przysyłania nowych wersji. Separacja zwiększa bezpieczeństwo i umożliwia rotację klucza uploadu bez utraty kontroli nad aplikacją.

Klucz	Kto przechowuje	Do czego służy	Co się stanie jeśli zgubisz?
Upload key (klucz przesłania)	Ty (deweloper) / CI/CD	Podpisujesz APK/AAB przed uploadem do Play Console	Można zresetować w Play Console (kontakt z supportem)
App Signing key (klucz aplikacji)	GOOGLE (bezpieczna infrastruktura)	Google podpisuje APK przed dostarczeniem użytkownikowi	Nie można zgubić, bo to Google trzyma, nie Ty

Dlaczego dwa klucze?

To podejście separation of concerns: ty możesz zgubić upload key (można zresetować), ale Google nigdy nie zgubi app signing key. Aplikacja dla użytkowników jest zawsze podpisana kluczem app signing. Ty podpisujesz 'kopertę' do Google, a Google przepakowuje apkę z własnym kluczem.

3.2. Generowanie upload keystore

Keystore jest generowany lokalnie przy użyciu narzędzi takich jak `keytool`. Zawiera klucz prywatny używany do podpisywania buildów przed uploadem. Proces wymaga określenia aliasu, hasła oraz okresu ważności. Bezpieczne przechowywanie keystore jest krytyczne, ponieważ jego utrata utrudnia aktualizację aplikacji.

Generowanie keystore, jednorazowe, wykonaj raz na projekt! (chyba, że zapomnisz hasła...)

```
keytool -genkey -v \
  -keystore upload-keystore.jks \           # nazwa pliku keystore
  -keyalg RSA \                             # algorytm RSA (wymagany przez Play)
  -keysize 2048 \                           # rozmiar klucza (minimum 2048 bitów)
  -validity 10000 \                         # ważność: ~27 lat (wystarczy na życie aplikacji)
  -alias upload                             # alias klucza w keystore

# Narzędzie zapyta o:
# - Hasło do keystore (KEY_STORE_PASSWORD) - ZAPAMIĘTAJ!
# - Hasło do klucza (KEY_PASSWORD) - może być to samo
# - Twoje dane (imie, firma, lokalizacja) - opcjonalne dla upload key

# BEZPIECZENSTWO: NIE commituj upload-keystore.jks do repozytorium!
echo 'upload-keystore.jks' >> .gitignore
echo 'key.properties' >> .gitignore
```

3.3. Konfiguracja key.properties

Plik `key.properties` przechowuje dane dostępowe do keystore (ścieżka, alias, hasło). Jest wykorzystywany przez Gradle podczas procesu builda. Hasła i ścieżka do keystore trafia do pliku `key.properties`. Plik ten powinien być wyłączony z repozytorium (np. `.gitignore`) ze względów bezpieczeństwa. Separacja konfiguracji umożliwia łatwe zarządzanie podpisywaniem w różnych środowiskach. Jest wstrzykiwany przez CI/CD lub trzymany lokalnie.

android/key.properties - NIE commituj tego do Git-a!

```
storePassword=twoje_haslo_keystora
keyPassword=twoje_haslo_klucza
keyAlias=upload
storeFile=../upload-keystore.jks # Sciezka wzgledna od android/app/
```

android/app/build.gradle - wczytaj key.properties i używaj do podpisywania

```
def keystoreProperties = new Properties()
def keystorePropertiesFile = rootProject.file('key.properties')
if (keystorePropertiesFile.exists()) {
    keystoreProperties.load(new FileInputStream(keystorePropertiesFile))
}

android {
    signingConfigs {
        release {
            keyAlias keystoreProperties['keyAlias']
            keyPassword keystoreProperties['keyPassword']
            storeFile keystoreProperties['storeFile'] ?
                file(keystoreProperties['storeFile']) : null
            storePassword keystoreProperties['storePassword']
        }
    }
    buildTypes {
        release {
            signingConfig signingConfigs.release // Uzyj powyzszej konfiguracji
            // ... minifyEnabled, proguardFiles ...
        }
    }
}
```

Kto ma słabą pamięć, ten robi dwa razy

Jeśli zapomnisz dodać signingConfig do buildType release, flutter build appbundle wygeneruje AAB podpisany kluczem debug. Play Console odmowi uploadu. Sprawdź zawsze czy key.properties jest dostępny.

4. Budowanie artefaktów; APK vs App Bundle

Flutter może zbudować dwa rodzaje artefaktów: APK (Android Package) i AAB (Android App Bundle). Różnią się przeznaczeniem, rozmiarem i sposobem dystrybucji.

Cecha	APK (.apk)	App Bundle (.aab)
Przeznaczenie	Bezpośrednia instalacja, dystrybucja poza Play Store, testy	Przesłanie do Google Play Store (od 2021)
Rozmiar na dysku	Zawiera wszystkie architektury	Zawiera tylko metadane; Play generuje optymalne APK per urządzenie
Rozmiar u użytkownika	Pełny rozmiar, bo pobiera się wszystko	Do 40% mniejszy - użytkownik pobiera tylko to, czego potrzebuje
Weryfikacja podpisu	Możesz zainstalować lokalnie, bez sklepu Play (ale tylko do 2027)	Google weryfikuje i podpisuje app signing key
Komenda	flutter build apk --release	flutter build appbundle --release

4.1. Budowanie release z flavor-em

Build release z flavorem generuje zoptymalizowany artefakt (APK/AAB) dla konkretnej konfiguracji. Proces obejmuje kompilację AOT, minifikację kodu oraz podpisanie aplikacji. Wariant builda jest determinowany przez kombinację flavor + release. Artefakt ten jest przeznaczony do dystrybucji.

Pełne polecenie budowania App Bundle dla floru `prod`

```
flutter build appbundle \
  --flavor prod \                # który flavor
  -t lib/main_prod.dart \        # punkt wejścia
  --release \                     # tryb release (optymalizacje)
  --obfuscate \                  # zaciemnianie kodu Dart
  --split-debug-info=build/debug-info/ \ # symboliki do kraw stosow
  --tree-shake-icons             # usunięcie nieużywanych ikon

# Wynik: build/app/outputs/bundle/prodRelease/app-prod-release.aab

# Budowanie split APK (per ABI) - do testów:
flutter build apk --split-per-abi --flavor prod -t lib/main_prod.dart --release
# Tworzy osobne APK dla arm64-v8a, armeabi-v7a, x86_64
```

0 --obfuscate i --split-debug-info

--obfuscate zaciemnia nazwy klas i funkcji Dart w binarnym kodzie. Utrudnia to reverse engineering, ale sprawia, że stack trace w logach są nieczytelne (MyClass.myMethod staje się a.b.c). --split-debug-info zapisuje mapowanie, które pozwala odkryć oryginalny stack trace. ZAWSZE używaj obu razem i archiwizuj debug-info dla każdej wersji - będziesz ich potrzebować do analizy crash-y w Crashlytics.

4.2. Weryfikacja artefaktu przed uploadem

Przed publikacją konieczna jest walidacja artefaktu: poprawność podpisu, wersjonowanie, działanie aplikacji oraz brak debugowych komponentów. Narzędzia takie jak `bundletool` umożliwiają inspekcję pakietu. Testy manualne i automatyczne powinny być wykonane na buildzie release.

Sprawdź czy AAB jest poprawnie podpisany

```
bundletool validate --bundle=app-prod-release.aab

# Sprawdź podpis APK:
apksigner verify --print-certs app-release.apk

# Testuj AAB lokalnie (bundletool generuje lokalne APK z AAB):
bundletool build-apks --bundle=app-prod-release.aab \
  --output=test-install.apks \
  --ks=upload-keystore.jks \
  --ks-pass=pass:twoje_haslo \
  --ks-key-alias=upload \
  --key-pass=pass:twoje_haslo

bundletool install-apks --apks=test-install.apks # Instalacja na podłączonym urządzeniu
```

5. Od konta dewelopera do publikacji na Google Play Store

Proces publikacji aplikacji w Play Store składa się z kilku etapów. Warto je znać, zanim zaczniesz pisać kod CI/CD. Automatyzacja ma sens tylko wtedy, gdy rozumiesz co jest automatyzowane.

5.1. Cykl życia wersji w Play Store

Publikacja aplikacji przebiega etapowo: upload artefaktu, przetwarzanie, testy (wewnętrzne, zamknięte, otwarte) oraz rollout produkcyjny. System wspiera stopniowe wdrażanie (staged rollout), co umożliwia kontrolę ryzyka. Każda wersja jest identyfikowana przez numer wersji i kod builda, co determinuje możliwość aktualizacji.

Etap	Co się dzieje	Dostępność	Czas
Internal testing	Szybka dystrybucja do max 100 testerów (e-mail whitelist). Brak recenzji Google. Idealny do testów CI/CD.	Tylko whitelist	Natychmiastowa

Closed testing (alfa)	Dystrybucja do grup testerów Google. Testerzy muszą dołączyć przez link. Brak publicznej recenzji.	Grupy testerów	Kilka minut
Open testing (beta)	Publiczny test - każdy może pobrać wersje beta. Widoczne w Play Store z oznaczeniem Beta.	Publiczna	Do 3 dni (pierwsza)
Production	Pełna publikacja. Można robić staged rollout (np. 10% użytkowników najpierw).	Wszyscy użytkownicy	1-3 dni (recenzja)

Wskazówka; użyj Internal Testing Track

Upload AAB, dodaj swój e-mail jako testera. Nie wymaga pełnej recenzji Google i jest darmowy. Będziesz mógł zainstalować aplikacje na telefonie w ciągu minut.

5.2. Konfiguracja aplikacji w Play Console

Przed pierwszym uploadem AAB musisz skonfigurować aplikację w Play Console. Wymagane elementy:

1. Konto dewelopera Google Play - odpłatne jednorazowo 25 USD (trwały dostęp).
2. Strona główna aplikacji: nazwa, opis krótki (80 zn.) i długi (4000 zn.), grafika (ikona 512x512, feature graphic 1024x500).
3. Polityka prywatności - URL do strony z polityką. WYMAGANA jeśli aplikacja zbiera jakiegokolwiek dane (nawet analityczne).
4. Klasyfikacja zawartości - kreatorem ocen IARC (pytania o treść).
5. Zarządzanie danymi - deklaracja jakie dane zbierasz (Data Safety Form).
6. Zapis w Google Play App Signing - potwierdź że Google będzie zarządzał kluczem podpisującym.

5.3. Automatyzacja przez „version code” i „version name”

Każdy upload do Play Store musi mieć wyższy versionCode niż poprzedni. W CI/CD można automatycznie generować versionCode na podstawie numeru buildu.

Automatyczny versionCode w GitHub Actions (będzie omówione w sekcji 6)

```
# versionCode = numer buildu GitHub (GITHUB_RUN_NUMBER)

# pubspec.yaml - wersja semantyczna:
version: 1.2.0+47 # format: versionName+versionCode
#           ^^^ versionName (widoczny dla użytkownika: 1.2.0)
#           ^^ versionCode (monotoniczny licznik, np. 47)

# Nadpisanie przy budowaniu (bez edycji pubspec.yaml):
flutter build appbundle \
  --build-name=1.2.0 \           # versionName
  --build-number=${GITHUB_RUN_NUMBER} \ # versionCode z CI
  --flavor prod -t lib/main_prod.dart -release
```

6. Automatyzacja potoku, CI/CD z GitHub Actions

CI/CD to skrót od Continuous Integration / Continuous Deployment. CI = automatyczne testowanie każdego commita. CD = automatyczne budowanie i publikowanie aplikacji. GitHub Actions to platforma CI/CD wbudowana w GitHub - darmowa dla repozytoriów publicznych i do 2000 minut miesięcznie dla prywatnych.

Analogia jako linia produkcyjna w fabryce

Wyobraź sobie fabrykę samochodów. Kiedy pracownik wprowadza zmiany do projektu (commit), linia produkcyjna automatycznie: sprawdza quality control (testy), składa auto (build), maluje je (podpisuje), wysyła do showroomu (Play Store). Nikt nie musi ręcznie klikać 'wyślij' - wszystko dzieje się automatycznie. GitHub Actions jest ta linia produkcyjna.

6.1. Koncepcja pipeline'u

Pipeline (potok) w GitHub Actions składa się z: triggera (co uruchamia), jobów (grup kroków) i kroków (pojedynczych poleceń). Każdy job biegnie na świeżym serwerze (runner).

Pojęcie	Wyjaśnienie / przykład
Workflow (.yml)	Plik YAML definiujący cały pipeline. Trafia do .github/workflows/. Jeden projekt może mieć wiele workflowów.
Trigger (on:)	Zdarzenie uruchamiające workflow: push na gałąź, pull_request, harmonogram (cron), ręcznie (workflow_dispatch).
Job	Grupa „kroków” biegnących na jednym runnerze. Joby mogą biec równolegle lub sekwencyjnie (needs:).
Step	Pojedynczy krok: użycie gotowej akcji (uses:) lub polecenie shell (run:). Wykonywany w kolejności.
Runner	Wirtualna maszyna (Ubuntu, macOS, Windows). ubuntu-latest dla Androida, macos-latest dla iOS.
Secret	Zaszyfrowana zmienna (hasło, klucz API). Definiowana w Settings > Secrets. Nie widoczna w logach.
Artifact	Plik/katalog zapisany po zakończeniu jobu. Można pobrać lub przekazać do następnego jobu.

6.2. Pełny workflow: test + build + deploy

Zautomatyzowany pipeline obejmuje trzy etapy: walidację, kompilację oraz dystrybucję. Najpierw uruchamiane są testy jednostkowe i widżetowe, co zapewnia poprawność logiczną i regresyjną. Następnie generowany jest artefakt builda (APK/AAB) w trybie release, z uwzględnieniem konkretnego flavoru. Ostatni etap to deploy – przesłanie artefaktu do systemu dystrybucji (np. Play Store). Workflow powinien być deterministyczny i wykonywany w środowisku CI, co eliminuje zależność od lokalnej konfiguracji.

.github/workflows/ci-cd.yml - pełny pipeline CI/CD dla WeatherApp	YAML
<pre> name: WeatherApp CI/CD on: push: branches: [main, develop] # Uruchom na push do main i develop pull_request: branches: [main] # Uruchom na PR do main workflow_dispatch: # Możliwość ręcznego uruchomienia z UI env: FLUTTER_VERSION: '3.22.0' # Globalne zmienne środowiskowe JAVA_VERSION: '17' # Przypnij wersje Fluttera! jobs: # === JOB 1: Testy jednostkowe i widżetów === test: name: Run Tests runs-on: ubuntu-latest steps: - name: Checkout kodu uses: actions/checkout@v4 # Pobierz repozytorium - name: Konfiguracja Java uses: actions/setup-java@v4 with: { distribution: temurin, java-version: '\${{ env.JAVA_VERSION }}' } - name: Instalacja Flutter SDK uses: subosito/flutter-action@v2 with: flutter-version: '\${{ env.FLUTTER_VERSION }}' channel: stable cache: true # Cache SDK - szybsze kolejne uruchomienia - name: Pobranie zależności run: flutter pub get - name: Weryfikacja formatowania run: dart format --output=none --set-exit-if-changed . </pre>	

```

- name: Analiza statyczna (linter)
  run: flutter analyze

- name: Uruchom testy z pokryciem
  run: flutter test --coverage --reporter github

- name: Upload coverage do Codecov
  uses: codecov/codecov-action@v4
  with:
    files: coverage/lcov.info

# === JOB 2: Budowanie i publikacja (tylko main branch) ===
build-and-deploy:
  name: Build & Deploy to Play Store
  runs-on: ubuntu-latest
  needs: test
  if: github.ref == 'refs/heads/main'          # Uruchom TYLKO jeśli test przeszedł!
                                              # Tylko dla main

  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-java@v4
      with: { distribution: temurin, java-version: '${{ env.JAVA_VERSION }}' }
    - uses: subosito/flutter-action@v2
      with: { flutter-version: '${{ env.FLUTTER_VERSION }}', cache: true }

    # - Odtworzenie keystore z secrets -
    - name: Dekoduj keystore z Base64
      run: |
        echo '${{ secrets.KEYSTORE_BASE64 }}' | base64 --decode > android/upload-keystore.jks
        echo 'storePassword=${{ secrets.KEY_STORE_PASSWORD }}' > android/key.properties
        echo 'keyPassword=${{ secrets.KEY_PASSWORD }}' >> android/key.properties
        echo 'keyAlias=${{ secrets.KEY_ALIAS }}' >> android/key.properties
        echo 'storeFile=./upload-keystore.jks' >> android/key.properties

    - run: flutter pub get

    # - Budowanie App Bundle -
    - name: Build App Bundle (prod)
      run: |
        flutter build appbundle \
          --flavor prod \
          -t lib/main_prod.dart \
          --release \
          --obfuscate \
          --split-debug-info=build/debug-info/ \
          --build-number=${{ github.run_number }} # Auto-inkrementowany versionCode

    # - Zapis artefaktów -
    - name: Uploaduj AAB jako artefakt
      uses: actions/upload-artifact@v4
      with:
        name: app-bundle-${{ github.run_number }}
        path: build/app/outputs/bundle/prodRelease/app-prod-release.aab

    # - Publikacja do Play Store Internal Testing -
    - name: Publikacja w Play Store
      uses: r0adkll/upload-google-play@v1
      with:
        serviceAccountJsonPlainText: ${{ secrets.SERVICE_ACCOUNT_JSON }}
        packageName: pl.edu.pam.weather
        releaseFiles: build/app/outputs/bundle/prodRelease/app-prod-release.aab
        track: internal # internal / alpha / beta / production
        status: completed # completed = od razu dostępny

```

6.3. Konfiguracja sekretów w GitHub

Sekrety to zaszyfrowane zmienne środowiskowe (GitHub Actions Secrets) dostępne tylko w kontekście GitHub Actions. Definiujesz je w Settings > Secrets and variables > Actions. Dostęp do nich jest ograniczony do pipeline'ów CI. Wykorzystywane są do przechowywania danych wrażliwych, takich jak klucze API, dane do podpisywania czy tokeny dostępu. Sekrety nie są dostępne w kodzie źródłowym i nie mogą być logowane w jawnej postaci. Ich użycie odbywa się poprzez wstrzykiwanie do środowiska wykonawczego workflow.

Nazwa sekretu	Wartość / jak uzyskać
KEYSTORE_BASE64	Keystore zakodowany w Base64: base64 -i upload-keystore.jks pbcopy (macOS) / base64 upload-keystore.jks (Linux)
KEY_STORE_PASSWORD	Hasło do keystore (wpisane przy keytool -genkey)
KEY_PASSWORD	Hasło do klucza w keystore (może być to samo co KEY_STORE_PASSWORD)
KEY_ALIAS	Alias klucza: upload (lub inny wybrany przy tworzeniu keystore)
SERVICE_ACCOUNT_JSON	JSON z kluczem konta serwisowego Google Cloud z dostępem do Play Developer API. Patrz Play Console > Setup > API access.

Encoding keystore do Base64. Ale dlaczego Base64 zapytacie?

Keystore to plik binarny, nie można go wkleić bezpośrednio do „sekrety” tekstowego. Rozwiązanie: zakoduj go do Base64 (tekst), wklej jako sekret, a w CI dekoduj z powrotem do pliku binarnego (krok 'Dekoduj keystore z Base64' powyżej).

6.4. Matryca buildu, czyli wiele flavorów równolegle

GitHub Actions pozwala zdefiniować matryce (matrix), czyli uruchomienie tego samego jobu z różnym zestawem parametrów. Matryca buildu umożliwia równoległe wykonywanie wielu wariantów pipeline'u, np. dla różnych flavorów i platform. Przydatne gdy chcemy budować dev, staging i prod równolegle. Każda kombinacja (np. dev/staging/prod) jest traktowana jako niezależne zadanie. Mechanizm ten skraca czas pipeline'u oraz zapewnia spójność buildów między środowiskami. Konfiguracja opiera się na deklaratywnym zdefiniowaniu osi macierzy (np. flavor, system operacyjny).

```
jobs:
  build:
    strategy:
      matrix:
        flavor: [dev, staging, prod]      # 3 job-y równolegle!
        include:
          - flavor: dev
            main: lib/main_dev.dart
            track: internal
          - flavor: staging
            main: lib/main_staging.dart
            track: alpha
          - flavor: prod
            main: lib/main_prod.dart
            track: internal      # Prod też najpierw internal
    runs-on: ubuntu-latest
    steps:
      - run: |
          flutter build appbundle \
            --flavor ${matrix.flavor} \
            -t ${matrix.main} \
            --release
```

7. Automatyzacja publikacji z Fastlane

Fastlane to narzędzie open-source do automatyzacji publikacji aplikacji mobilnych. Może działać samodzielnie lub w połączeniu z GitHub Actions. Jego główną przewagą to gotowe 'lane' (ścieżki) dla typowych operacji: testowanie, budowanie, podpisywanie, upload do sklepu.

Analogia jako skrypt vs. narzędzie

Kolację można przygotować sobie ręcznie albo użyć robota kuchennego (czy czegokolwiek automatycznego, ot założymy burżujskiego termomixa). Skrypty bash/PowerShell to gotowanie ręcznie. Działa, ale trzeba

wszystkiego pilnować. Fastlane to taki termomix. Masz gotowe, sprawdzone przepisy (actions) i tylko wywołujesz je po nazwie. Mniej błędów, mniej kodu, lepsza czytelność.

7.1. Konfiguracja Fastlane

Fastlane automatyzuje procesy związane z budowaniem i dystrybucją aplikacji. Konfiguracja opiera się na definicji tzw. „lane’ów”, które opisują sekwencje operacji (np. build, podpisanie, upload). Narzędzie integruje się z systemami dystrybucji i pozwala na zarządzanie wersjami, metadanymi oraz screenshotami. Automatyzacja redukuje błędy manualne i standaryzuje proces release.

Instalacja Fastlane (wymagane Ruby)

```
gem install fastlane

# Inicjalizacja w katalogu projektu Flutter:
cd twój_projekt_flutter/
fastlane init
# Fastlane zapyta o platformę: wybierz 'android'
# Fastlane tworzy katalog fastlane/ z plikami Fastfile i Appfile

# fastlane/Appfile - identyfikacja aplikacji
json_key_file("fastlane/service-account.json") # Klucz Google Play API
package_name("pl.edu.pam.weather")             # Package name prod

# fastlane/Fastfile - definicja lanes (ściezek automatyzacji)
default_platform(:android)

platform :android do

  # LANE: Uruchom testy
  lane :test do
    sh('flutter test --coverage --reporter github')
  end

  # LANE: Budowanie i upload do Internal Testing
  lane :deploy_internal do
    sh('flutter build appbundle --flavor prod -t lib/main_prod.dart --release')
    upload_to_play_store(
      track: 'internal',
      aab: 'build/app/outputs/bundle/prodRelease/app-prod-release.aab',
      skip_upload_metadata: true, # Nie aktualizuj opisu w Play Store
      skip_upload_images: true,  # Nie aktualizuj grafik
    )
  end

  # LANE: Promuj Internal -> Beta
  lane :promote_to_beta do
    upload_to_play_store_internal_app_sharing(
      # Lub prosta promocja:
    )
    # Albo używając supply (starsza akcja):
    supply(track_promote_to: 'beta', track: 'internal')
  end

  # LANE: Pełna publikacja (dev -> staging -> prod)
  lane :full_deploy do
    test
    deploy_internal
    slack(message: 'Nowy build wyleciał do Internal Testing!')
  end
end

# Uruchamianie lanes:
fastlane test          # Tylko testy
fastlane deploy_internal # Build + upload
fastlane full_deploy    # Testy + build + upload + motyfikacja
```

Integracja Fastlane w GitHub Actions

Zamiast ręcznie pisać kroki flutter build i upload, wywołaj fastlane deploy_internal w kroku run:. To czystszy kod CI i jest łatwiejszy do testowania lokalnie (uruchomisz te same kroki co CI).

8. Zarządzanie sekretami i zmiennymi środowiskowymi

Klucze API, hasła do baz danych, klucze Firebase, (itd.) to dane, które nie mogą trafić do repozytorium. Istnieje dlatego kilka warstw zarządzania sekretami w projekcie Flutter.

8.1. Zasady bezpiecznego zarządzania kluczami

Klucze kryptograficzne i API powinny być izolowane od kodu źródłowego oraz przechowywane w bezpiecznych magazynach (np. secrets w CI). Należy stosować zasadę minimalnych uprawnień oraz rotację kluczy. Dostęp do nich powinien być audytowalny. Klucze nie powinny być hardkodowane ani przechowywane w repozytorium. W środowisku lokalnym zaleca się użycie plików konfiguracyjnych ignorowanych przez system kontroli wersji.

- Nigdy nie commituj kluczy do Git. Używaj .gitignore dla plików z kluczami.
- Nie „hardkoduj” kluczy w kodzie Dart. Nawet zaciemniony kod można zdeobfuskować (odciemnić).
- Używaj osobnych kluczy dla każdego środowiska (dev/staging/prod). Kompromitacja klucza dev nie dotyka prod.
- Rotuj klucze regularnie. Jeśli klucz wyciekł, unieważnij go i wygeneruj nowy.
- Ogranicz uprawnienia kluczy do minimum. Klucz API powinien mieć dostęp tylko do tego co potrzebuje.

8.2. Dart define, czyli wstrzykiwanie kluczy na etapie budowania

Flutter umożliwia przekazanie zmiennych środowiskowych przez `--dart-define`. Wartości są wbudowywane w kod binarny w momencie kompilacji. Nie są w plikach tekstowych w repozytorium. Wartości te są dostępne w kodzie poprzez stałe kompilacyjne (`const String.fromEnvironment`). Pozwala to na dynamiczną konfigurację bez modyfikacji kodu źródłowego. Dane są osadzone w artefakcie builda, co eliminuje potrzebę runtime’owego ładowania konfiguracji.

Wstrzyknięcie klucza API przez `--dart-define`

```
flutter run --dart-define=API_KEY=abc123 --dart-define=MAP_KEY=xyz456
```

W kodzie Dart odczyt przez `String.fromEnvironment`

```
const apiKey = String.fromEnvironment('API_KEY', defaultValue: '');  
// 'API_KEY' musi dokładnie odpowiadać nazwie w --dart-define!  
// defaultValue jest używane gdy budujemy bez --dart-define (np. unit testy)
```

W GitHub Actions jest przekazanie sekretów przez `--dart-define`

```
run: |  
  flutter build appbundle \  
    --dart-define=API_KEY=${{ secrets.PROD_API_KEY }} \  
    --dart-define=SENTRY_DSN=${{ secrets.SENTRY_DSN }} \  
    --flavor prod -t lib/main_prod.dart --release  
  
// odczyt dart-define z typami lib/core/config/app_config.dart  
class AppSecrets {  
  // const jest wartością znaną w compile time!  
  static const String apiKey = String.fromEnvironment('API_KEY');  
  static const String sentryDsn = String.fromEnvironment('SENTRY_DSN');  
  
  // Walidacja już w runtime, bo będzie crash-fast jeśli klucza brakuje w prod  
  static void validate() {  
    assert(apiKey.isNotEmpty, 'API_KEY nie został ustawiony! Użyj --dart-define.');
```

8.3. Plik z kluczami; --dart-define-from-file

Przy wielu kluczach wpisywanie ich w wierszu poleceń staje się niepraktyczne. Flutter obsługuje wczytywanie z pliku JSON. Rozszerzenie `--dart-define-from-file` umożliwia wczytanie zestawu parametrów z pliku (np. JSON). Ułatwia to zarządzanie większą liczbą zmiennych konfiguracyjnych oraz ich wersjonowanie poza kodem aplikacji. Plik taki może być generowany w pipeline CI na podstawie sekretów, co zapewnia separację konfiguracji od implementacji.

.env.dev.json to plik z kluczami dev (NIE commituj tego do Git-a!)

```
{
  "API_KEY": "dev-key-abc",
  "SENTRY_DSN": "",
  "ANALYTICS_ENABLED": "false"
}
```

Użycie

```
flutter run --dart-define-from-file=.env.dev.json -t lib/main_dev.dart --flavor dev
```

Upewnij się że pliki .env* są ignorowane (przez .gitignore)

```
*.env.json
.env.*
*.env
```

9. Monitorowanie przez Firebase Crashlytics i Analytics

Aplikacja w produkcji to aplikacja, która będzie się crash-ować. Nie dlatego że jest źle napisana, ale dlatego że użytkowników są miliony, urządzenia są różne, sieć jest zawodna. Firebase Crashlytics pozwala zbierać raporty o awariach i analizować je bez dostępu do urządzenia użytkownika.

9.1. Konfiguracja Firebase

Integracja z Firebase wymaga dodania plików konfiguracyjnych specyficznych dla platformy oraz inicjalizacji SDK w aplikacji. System dostarcza usługi backendowe, takie jak analityka, crash reporting czy push notifications. Konfiguracja obejmuje powiązanie aplikacji z projektem Firebase oraz ustawienie odpowiednich identyfikatorów i kluczy.

Instalacja FlutterFire CLI (narzędzie do konfiguracji Firebase)

```
dart pub global activate flutterfire_cli
```

Konfiguracja projektu Firebase (wymaga konta Firebase i wcześniejszego zalogowania)

```
flutterfire configure \
  --project=weather-app-prod \
  --platforms=android
```

Efekt to `lib/firebase_options.dart` z konfiguracją dla każdej platformy

pubspec.yaml # zależności Firebase

```
dependencies:
  firebase_core: ^3.3.0
  firebase_crashlytics: ^4.1.0 # Raporty crash-y
  firebase_analytics: ^11.2.0 # Analityka użycia
```

main.dart

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();

  await Firebase.initializeApp()
```

```

    options: DefaultFirebaseOptions.currentPlatform, // z firebase_options.dart
  );

  // Crashlytics: przechwytuje błędy niezłapane przez Flutter
  FlutterError.onError =
    FirebaseCrashlytics.instance.recordFlutterFatalError;

  // Przechwytuje błędy asynchroniczne spoza UI
  PlatformDispatcher.instance.onError = (error, stack) {
    FirebaseCrashlytics.instance.recordError(error, stack, fatal: true);
    return true;
  };

  AppConfig.initProd(); // lub initDev()
  runApp(const ProviderScope(child: MyApp()));
}

```

9.2. Logi niestandardowe i atrybuty

Systemy analityczne i monitorujące umożliwiają rejestrowanie zdarzeń oraz atrybutów kontekstowych. Logi niestandardowe pozwalają śledzić zachowanie użytkownika oraz diagnozować problemy. Atrybuty (np. wersja aplikacji, typ urządzenia) wzbogacają dane analityczne i umożliwiają segmentację. Kluczowe jest ograniczenie ilości danych wrażliwych oraz zachowanie zgodności z regulacjami prywatności.

Dodawanie kluczowych atrybutów użytkownika do raportów crash-ów

```

// Pomaga zrozumieć kontekst awarii (który użytkownik, jaki plan)
FirebaseCrashlytics.instance.setUserIdentifier(userId);
FirebaseCrashlytics.instance.setCustomKey('flavor', AppConfig.instance.flavor.name);
FirebaseCrashlytics.instance.setCustomKey('app_version', packageInfo.version);

// Manualne logowanie błędu (nie-fatalnego)
try {
  await weatherService.fetchWeather(city);
} catch (e, stack) {
  // Zaloguj w Crashlytics bez crash-owania aplikacji:
  FirebaseCrashlytics.instance.recordError(e, stack, fatal: false);
  // Pokaz błąd użytkownikowi i kontynuuj
}

```

Warto separować Crashlytics między flavor-ami

Stwórz ODDZIELNE projekty Firebase dla dev/staging/prod. Crash-e z testów nie mieszają się z tymi produkcyjnymi. Konfiguruj przez flutterfire configure --project=weather-app-dev dla dev i --project=weather-app-prod dla prod(ukcji), zapisując różne lib/firebase_options_dev.dart i lib/firebase_options_prod.dart.

10. Zadania do wykonania

Zadania bazują na projekcie **WeatherApp v2** z Flutter Lab 2. Każde zadanie kończy się weryfikacją. Wykonaj ją zanim przejdziesz dalej.

Zadanie 1. Konfiguracja Flavorów

1. Utwórz klasę AppConfig z trzema fabrykami: initDev(), initStaging(), initProd(). Pola: baseUrl, appName, enableAnalytics, showDebugBanner. Typ wyliczeniowy AppFlavor.
2. Utwórz trzy punkty wejścia: main_dev.dart, main_staging.dart, main_prod.dart. Każdy wywołuje odpowiednią fabrykę AppConfig przed runApp().
3. Skonfiguruj Android Gradle (android/app/build.gradle): trzy productFlavors (dev, staging, prod) z różnym applicationIdSuffix i versionNameSuffix.
4. Przetestuj ze możesz uruchomić wszystkie trzy flavor-y jednocześnie na emulatorze (trzy ikony na ekranie głównym). Użyj różnych nazw aplikacji (DEV/STG/bez prefixu).
5. W WeatherCard wyświetl aktualne środowisko jako mały badge (np. [DEV] w lewym górnym rogu karty), widoczny tylko dla flavorów dev i staging.

WERYFIKACJA: Screenshot AVD z trzema ikonami WeatherApp na ekranie głównym z różnymi nazwami. flutter run --flavor dev i --flavor prod uruchamiają aplikacje bez błędów.

Zadanie 2. Podpisywanie i budowanie

6. Wygeneruj upload keystore poleceniem keytool. Zapisz w katalogu projektu (nie w android/app/!). Dodaj upload-keystore.jks i android/key.properties do .gitignore.
7. Skonfiguruj android/key.properties z hasłem, aliasem i ścieżką. Skonfiguruj android/app/build.gradle: signingConfigs.release z wczytywaniem key.properties.
8. Zbuduj podpisany App Bundle dla flavoru prod: flutter build appbundle --flavor prod -t lib/main_prod.dart --release.
9. Zweryfikuj podpis za pomocą apksigner lub bundletool. Sprawdź rozmiar wygenerowanego .aab.
10. Opcjonalnie: załóż konto Google Play (25 USD) i uploaduj AAB do Internal Testing Track.

WERYFIKACJA: Plik app-prod-release.aab istnieje w build/app/outputs/bundle/prodRelease/. Komenda apksigner verify nie zgłosiła błędów. .gitignore zawiera wpisy dla keystore.

Zadanie 3. Pipeline CI/CD z GitHub Actions

11. Stwórz repozytorium GitHub (publiczne lub prywatne) i wypchaj projekt WeatherApp.
12. Stwórz workflow .github/workflows/ci.yml uruchamiany na push do dowolnej galezi i na pull_request do main. Workflow powinien: pobrać kod, zainstalować Flutter, uruchomić flutter analyze i flutter test.
13. Dodaj do repozytorium sekrety: KEYSTORE_BASE64 (zakodowany keystore), KEY_STORE_PASSWORD, KEY_PASSWORD, KEY_ALIAS. Rozszerz workflow o job build-release uruchamiany tylko na main, zależy od test i buduje podpisany AAB.
14. Workflow powinien zapisywać zbudowany AAB jako GitHub Artifact dostępny do pobrania po zakończeniu buildu (actions/upload-artifact).
15. Opcjonalnie: dodaj krok publikacji do Play Store Internal Testing przez r0adkll/upload-google-play i sekret SERVICE_ACCOUNT_JSON.

WERYFIKACJA: Screenshot z GitHub Actions pokazujący zielony workflow z dwoma jobami: test (wszystkie kroki zielone) i build-release (AAB widoczny jako artefakt do pobrania).

Zadanie 4. Sekrety i monitorowanie

16. Zrefaktoryzuj AppConfig tak, aby URL API był wstrzykiwany przez --dart-define=API_BASE_URL. Utwórz plik .env.dev.json z kluczem API_BASE_URL ustawionym na adres Open-Meteo. Dodaj .env.dev.json do .gitignore.
17. Zaktualizuj workflow GitHub Actions, aby w kroku build przekazywać --dart-define=API_BASE_URL=\${{ secrets.PROD_API_URL }}. Dodaj sekret PROD_API_URL do repozytorium GitHub.
18. Zintegruj Firebase Crashlytics (opcjonalnie bez konta Firebase - wystarczy poprawna konfiguracja w kodzie z komentarzem wyjaśniającym kroki inicjalizacji). Pokaż w kodzie: inicjalizację Crashlytics, przechwycenie FlutterError.onError, manualne logowanie błędu w bloku catch.
19. Napisz dokument README.md (w repozytorium GitHub) opisujący: jak uruchomić każdy flavor lokalnie, jak działa pipeline CI/CD, jak dodać nowy sekret do GitHub Actions.

WERYFIKACJA: README.md w repozytorium. Workflow GitHub Actions przekazuje API_BASE_URL z sekretu. Kod pokazuje inicjalizację Crashlytics.

11. Kryteria oceniania

11.1. Punktacja zadań

Zadanie	Punkty	Co weryfikuje prowadzący
Zad. 1: Flawory	20 pkt	Trzy flawory działają równocześnie na emulatorze. AppConfig z różnym baseUrl. Badge środowiska w UI.
Zad. 2: Podpisywanie	25 pkt	Poprawnie wygenerowany keystore. Podpisany AAB. .gitignore chroni klucze. Zweryfikowany podpis.
Zad. 3: CI/CD	35 pkt	Dwa job-y (test + build). Zielony pipeline na GitHub. AAB dostępny jako artefakt. Sekrety nie wyświetlają się w logach.
Zad. 4: Sekrety i monitoring	20 pkt	dart-define w kodzie i CI. README z dokumentacją. Crashlytics w kodzie.
RAZEM	100 pkt	

11.2. Skala ocen

Ocena	Punkty / wymagania
5.0	90-100 pkt - Wszystkie zadania. CI/CD z automatycznym uploadem do Play Store. README. dart-define w CI.
4.5	80-89 pkt - Zadania 1-3 kompletne. Artefakt AAB w GitHub Actions. Sekrety poprawnie skonfigurowane.
4.0	70-79 pkt - Zadania 1-2 + podstawowy CI (tylko job test). Podpisany AAB zbudowany lokalnie.
3.5	60-69 pkt - Zadania 1-2. Trzy flawory działają. Podpisany AAB. Brak CI/CD.
3.0	50-59 pkt - Zadanie 1: Trzy flawory działają. AppConfig zaimplementowany.
2.0	0-49 pkt - Projekt nie kompiluje lub flawory nie działają.

12. Najczęstsze błędy i ich rozwiązania

Błąd / objaw	Przyczyna i rozwiązanie
Gradle error: Flavor 'dev' not found	flutter run --flavor dev ale brak definicji w build.gradle. Sprawdź czy productFlavors { dev { ... } } istnieje w android/app/build.gradle. Flavorname musi się dokładnie zgadzać (case-sensitive).
MissingPluginException po dodaniu Firebase	Zmiana w build.gradle lub AndroidManifest.xml wymaga pełnego restartu (nie Hot Reload/Restart). Wykonaj: flutter clean && flutter pub get && flutter run.
Upload AAB odrzucony: 'APK was not signed'	buildType release nie ma signingConfig release lub key.properties jest niedostępny. Sprawdź czy plik key.properties istnieje i czy ścieżka w storeFile jest poprawna.
String.fromEnvironment() zwraca pusty string	Brak --dart-define=NAZWA=WARTOSC przy uruchamianiu. W Android Studio skonfiguruj w Run > Edit Configurations > Additional run args: --dart-define=API_KEY=wartość.
GitHub Actions: Permission denied przy uploading to Play	Konto serwisowe nie ma uprawnień do Play Developer API. W Play Console: Setup > API access > Link Service Account. Przyznaj rolę 'Release Manager' lub 'Admin'.
versionCode nie jest większy niż poprzedni	Probujesz uploadować AAB z tym samym lub niższym build number co poprzedni upload. Użyj --build-number=\${{ github.run_number }} w CI lub ręcznie inkrementuj w pubspec.yaml.
Trzy ikony o tej samej nazwie na emulatorze	Brak resValue 'string', 'app_name', '...' w definicjach flavorów lub AndroidManifest.xml używa hardkodowanej nazwy zamiast @string/app_name.
CI/CD: Keystore nieznalesiony przy budowaniu	Plik keystore jest tworzony w kroku 'Dekoduj keystore' ale ścieżka w key.properties jest błędna. Sprawdź że echo zapisuje do android/key.properties a keystore do katalogu root projektu.
AppConfig.instance rzuca AssertionError	AppConfig.initDev() (lub inna fabryka) nie została wywołana przed dostępem do instance. Sprawdź kolejność w main(): najpierw WidgetsFlutterBinding.ensureInitialized(), potem AppConfig.initXxx(), a na końcu runApp().

Przydatne komendy diagnostyczne

```
# Sprawdź jakie flavy sa skonfigurowane w Gradle:
cd android && ./gradlew tasks | grep -i flavor
# Lista dostępnych urządzeń (dla --device-id):
flutter devices
# Verbose logi budowania (gdy Gradle się sypie):
flutter build appbundle --verbose 2>&1 | head -100
# Sprawdź zawartość keystore:
keytool -list -v -keystore upload-keystore.jks
# Weryfikacja podpisu APK:
apksigner verify --verbose --print-certs app-release.apk
# Symulacja CI lokalnie (podobnie do GitHub Actions):
act push # Używa narzędzia 'act'
(https://github.com/nektos/act)
# Logi GitHub Actions - pełna historia:
gh run view --log # Używa GitHub CLI
```

Instrukcja Laboratoryjna Flutter 3 - Publikacja aplikacji, CI/CD i Flavors

Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki

MobileHub

Następne ćwiczenie: Flutter 4 - Testy end-to-end, Flutter Web i Desktop, Isolates