POLITECHNIKA RZESZOWSKA
KATEDRA INFORMATYKI I AUTOMATYKI
DR INŻ. MATEUSZ POMIANEK

ĆWICZENIE LABORATORYJNE FLUTTER 1

Dart · Flutter · Widgety · 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. Widgety - fundamenty UI we Flutterze 7
6. Zarządzanie stanem - od setState do Provider 10
7. Nawigacja - Navigator 2.0 i GoRouter 12
8. Pobieranie danych z REST API 14
9. Motyw aplikacji - Material Design 3 16
10. Projekt WeatherApp - architektura i struktura 17
11. Zadania do wykonania 19
12. Kryteria oceniania 20
13. Dart vs Kotlin - tabela porównawcza 20
14. Najczęstsze błędy i ich rozwiązania 21

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 crossplatform

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 widgetów 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 widgetów 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 Kotlina: 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.

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

#	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

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)

# Skróty 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 widgetów. 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.

3.1. Mapa katalogów

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 widgetu
├── android/                    ← Kod natywny Android (XML, Kotlin) - rzadko edytujesz
│   ├── app/src/main/
│   │   ├── AndroidManifest.xml ← Uprawnienia: INTERNET tutaj!
│   │   └── kotlin/.../         ← MainActivity.kt - pusta, Flutter ją przejmuje
│   └── 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

kluczowe sekcje pubspec.yaml	Wyjaśnienie
`name: weather_app`	Nazwa paczki (małe litery, podkreślniki). Używana jako identyfikator.
`description: Aplikacja pogodowa`	Opis projektu - widoczny w pub.dev jeśli publikujesz.
`publish_to: 'none'`	Zapobiega przypadkowemu opublikowaniu na pub.dev.
`version: 1.0.0+1`	Wersja: major.minor.patch+buildNumber. +1 = versionCode na Androidzie.
`environment:`	
`  sdk: '>=3.0.0 <4.0.0'`	Minimalna wersja Dart SDK. Zawsze ustaw zakres, nie samą dolną granicę.
`dependencies:`	Paczki RUNTIME - używane w działającej aplikacji.
`  flutter:`	Obowiązkowa zależność na framework Flutter.
`    sdk: flutter`	sdk: flutter = paczka wbudowana, nie z pub.dev.
`  http: ^1.2.1`	Paczka http z pub.dev. ^ = dowolna kompatybilna wersja (1.x.x).
`dev_dependencies:`	Paczki DEVELOPERSKIE - tylko podczas kompilacji i testów.
`  flutter_test:`	Biblioteka do testów widgetów - wbudowana.
`    sdk: flutter`	

kluczowe sekcje pubspec.yaml	Wyjaśnienie
`flutter:`	Sekcja konfiguracji Flutter (assets, fonts).
`  uses-material-design: true`	Włącza ikony Material Design (wymagane dla Icons.xxx).
`  assets:`	Lista zasobów statycznych do dołączenia do APK.
`    - assets/images/`	Cały folder images/ (ze / na końcu = cały katalog).
`    - assets/data/cities.json`	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. W tej sekcji skupiamy się na elementach, które będą używane w projekcie WeatherApp.

4.1. Typy, zmienne i null safety

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

4.2. Funkcje i parametry

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

Dart - funkcje	Wyjaśnienie
`// Arrow function (=>) - jednoliniowe funkcje`	=> expr jest skrótem dla { return expr; }
`double toCelsius(double f) => (f - 32) * 5 / 9;`	Identyczne z: { return (f - 32) * 5 / 9; }
`// 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

Dart - klasy	Wyjaśnienie
`class WeatherData {`	Klasa Dart - domyślnie publiczna
`  final String city;  // pole final = readonly`	Brak private/public na polach - użyj _ 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 - niezmienność + 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

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 Kotlina
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. Widgety - fundamenty UI we Flutterze

Jeśli Jetpack Compose opiera się na funkcjach @Composable, to Flutter opiera się na klasach Widget. Wszystko we Flutterze jest widgetem: padding, kolumna, tekst, obraz, nawet aplikacja. To trochę jak klocki LEGO - z gotowych, małych elementów budujesz złożone UI.

Analogia: drzewo widgetów jak drzewo genealogiczne

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

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

Każdy widget 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 widgetów 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 - widget bez stanu

WeatherCard.dart - StatelessWidget	Wyjaśnienie
`class WeatherCard extends StatelessWidget {`	extends StatelessWidget = ten widget nigdy nie zmienia stanu
`  final WeatherData weather;  // dane z zewnątrz`	Dane przekazywane przez konstruktor - immutable
`  final VoidCallback? onTap;  // callback opcjonalny`	VoidCallback = void Function() - callback bez argumentów
`  const WeatherCard({  // const constructor!`	const = Flutter może optymalizować: nie tworzy jeśli dane te same
`    super.key,  // przekaż key do rodzica`	Key pomaga Flutter śledzić widgety w listach - zawsze podawaj

WeatherCard.dart - StatelessWidget	Wyjaśnienie
`    required this.weather,`	
`    this.onTap,`	
`  });`	
`  @override`	@override = implementujesz metodę abstrakcyjną z StatelessWidget
`  Widget build(BuildContext context) {`	build() = Flutter woła ją kiedy widget ma się narysować
`    return Card(              // Card = Material widget z cieniem`	Card to gotowy widget z Material Design - cień, zaokrąglenia
`      child: InkWell(         // Reaguje na dotyk z ripple effect`	InkWell dodaje efekt fali (ripple) przy tapnięciu
`        onTap: onTap,`	Przekazujemy callback - null = wyłącz tapowanie
`        child: Padding(       // Padding wewnątrz karty`	Padding to WIDGET - nie property jak w Androidzie
`          padding: const EdgeInsets.all(16),`	EdgeInsets.all(16) = 16dp ze wszystkich stron
`          child: Column(      // Kolumna dzieci pionowo`	Column = LinearLayout pionowy
`            children: [`	children = lista dzieci widgetu
`              Text(weather.city,`	Text wyświetla string
`                style: Theme.of(context).textTheme.headlineMedium,`	Theme.of(context) = styl z tematu aplikacji
`              ),`	
`Text('${weather.temp.toStringAsFixed(1)}°C'),`	Interpolacja stringa w Dart: ${wyrażenie}
`            ],`	
`          ),`	
`        ),`	
`      ),`	
`    );`	
`  }`	
`}`	

5.2. StatefulWidget - widget 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).

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

Podkreślnik _ = 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
`class WeatherScreen extends StatefulWidget {`	Widget (immutable) - tylko konfiguracja

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

5.3. Podstawowe widgety układu; cheat sheet

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 = leniwia 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

setState() działa świetnie dla lokalnego stanu jednego widgetu. Ale gdy wiele widgetów potrzebuje tych samych danych (np. lista ulubionych miast wyświetlana na kilku ekranach), potrzebujemy globalnego zarządzania stanem. W tym ćwiczeniu użyjemy Provider - najprostszego i oficjalnie zalecanego rozwiązania dla początkujących.

6.1. Problem z setState() na dużą skalę

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

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

weather_provider.dart - ChangeNotifier	Wyjaśnienie
`class WeatherProvider extends ChangeNotifier {`	ChangeNotifier = klasa z mechanizmem powiadamiania
`  WeatherData? _weather;  // _ = prywatne`	Prywatne pole - zmień tylko przez metody publiczne
`  bool _isLoading = false;`	
`  String? _error;`	
`  // Gettery publiczne - widgety czytają przez nie`	Udostępniamy tylko do odczytu (immutable z zewnątrz)
`  WeatherData? get weather => _weather;`	get weather = właściwość computed (jak val w Kotlinie)
`  bool get isLoading => _isLoading;`	
`  String? get error => _error;`	
`  // Metoda zmieniająca stan`	
`  Future<void> loadWeather(String city) async {`	Publiczna metoda - widgety mogą ją wywołać
`    _isLoading = true;`	Zmień prywatne pole...
`    notifyListeners();  // POWIADOM widgety!`	notifyListeners() = odpowiednik setState() dla Provider
`    try {`	
`      _weather = await WeatherService().fetch(city);`	await = asynchroniczne pobieranie
`      _error = null;`	
`    } catch (e) {`	
`      _error = e.toString();`	
`    } finally {`	
`      _isLoading = false;`	
`      notifyListeners();  // Powiadom po zakończeniu`	Flutter przerysuje wszystkie nasłuchujące widgety
`    }`	
`  }`	
`}`	

6.4. Rejestracja i odczyt Provider

Provider - rejestracja i odczyt (trzy metody)

```
// 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 widgecie 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 nasłuchiwania (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!
```

context.watch() tylko w build() - 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

Flutter ma dwa systemy nawigacji. Stary Navigator 1.0 (push/pop) jest prosty dla małych aplikacji. GoRouter to oficjalnie zalecane rozwiązanie dla aplikacji produkcyjnych - oparty na URL-ach, wspiera deep linking i działa tak samo na wszystkich platformach.

7.1. Navigator 1.0 - push i pop

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

Podstawowa nawigacja Navigator	Wyjaśnienie
`// result = String? - co przekazał SearchScreen przez pop(result)`	
`if (result != null) useResult(result);`	

7.2. GoRouter - nawigacja dla większych aplikacji

Instalacja GoRouter

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

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

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.

8.1. Przegląd Open-Meteo API

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: &hourly=temperature_2m - dane co godzinę na 7 dni. Duża odpowiedź JSON.
Prognoza dzienna	Parametr: &daily=temperature_2m_max,temperature_2m_min - dane dzienne na 7 dni.

8.2. Uprawnienie INTERNET, czyli krok którego nie wolno pominąć

android/app/src/main/AndroidManifest.xml

```
<!-- 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>
```

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

weather_service.dart - pełna implementacja	Wyjaśnienie
`import 'dart:convert';  // jsonDecode()`	dart:convert = wbudowana biblioteka JSON (bez import z pub.dev!)
`import 'package:http/http.dart' as http;`	as http = alias - unikamy konfliktu nazw
`class WeatherService {`	
`  static const _baseGeo =`	static const = stała na poziomie klasy (jak companion object)
`    'https://geocoding-api.open-meteo.com/v1/search';`	
`  static const _baseWeather =`	
`    'https://api.open-meteo.com/v1/forecast';`	

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

8.4 FutureBuilder: wyświetlanie asynchronicznych danych

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

FutureBuilder - użycie	Wyjaśnienie
`FutureBuilder<WeatherData>(`	Generic typ = co Future zwróci. Tu WeatherData.
`  future: _weatherFuture,   // Future przypisany raz!`	WAŻNE: przypisz Future w initState(), nie w build()!
`  builder: (context, snapshot) {`	snapshot = AsyncSnapshot<WeatherData>
`    // Stan 1: czekanie na dane (loading)`	

FutureBuilder - użycie	Wyjaśnienie
`if (snapshot.connectionState == ConnectionState.waiting) {`	waiting = Future jest w toku
`  return const Center(child: CircularProgressIndicator());`	
`}`	
`// Stan 2: błąd`	
`if (snapshot.hasError) {`	hasError = Future rzucił wyjątek
`  return Center(child: Text('Błąd: ${snapshot.error}'));`	snapshot.error = złapany wyjątek
`}`	
`// 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`	hasData gwarantuje non-null - ! nie spowoduje crashu
`return WeatherCard(weather: weather);`	
`},`	
`)`	

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

```
// 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
}
```

Zmienna _future jest 'late final' - inicjalizowana raz w initState(), potem tylko czytana.

9. Motyw aplikacji - Material Design 3

Flutter domyślnie implementuje Material Design 3 (Material You). 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

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)

    // Typografia - opcjonalna customizacja
```

```
    textTheme: TextTheme(
      displayLarge: GoogleFonts.notoSansTextTheme().displayLarge,
      headlineMedium: const TextStyle(fontWeight: FontWeight.bold),
    ),

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

  // Dark mode - osobny motyw
  darkTheme: ThemeData(
    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 hardcoded wartości

Kolory z Theme.of(context)

```
// ŹLE - hardcoded kolor:
Container(color: const Color(0xFF027DFD), ...)   // Złamie 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

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

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

main.dart

```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 SZCZYCIE 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>.
Dwuetapowo: 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)
`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 widgetach.
`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()

Kotlin	Dart - odpowiednik
`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 uprawnienia 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 widgetu (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 widget 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. Zawiń Column w SingleChildScrollView lub użyj Expanded/Flexible na dzieciach rozciągliwych.
`pubspec.yaml - Unrecognized keys`	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 widgetów 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