

POLITECHNIKA RZESZOWSKA
KATEDRA INFORMATYKI I AUTOMATYKI
DR INŻ. MATEUSZ POMIANEK

ĆWICZENIE LABORATORYJNE - REACT NATIVE 1

TypeScript · Komponenty · Nawigacja · REST API

Programowanie Aplikacji Mobilnych - moduł RN

Projekt WeatherApp: aplikacja pogodowa z Open-Meteo API

Wymagania wstępne: znajomość podstaw JavaScript/TypeScript, zainstalowany Node.js i Android Studio

Spis treści

1. Czym jest React Native i jak działa?.....	1
2. Instalacja i konfiguracja środowiska.....	2
3. Struktura projektu React Native.....	4
4. TypeScript - podstawy języka w kontekście RN.....	5
5. Komponenty - fundamenty UI w React Native.....	6
6. Stan i efekty - useState, useEffect i Context API.....	8
7. Nawigacja - React Navigation i Expo Router.....	10
8. Pobieranie danych z REST API.....	12
9. Stylowanie - StyleSheet i nowoczesne podejścia.....	14
10. Projekt WeatherApp - architektura i struktura.....	15

1. Czym jest React Native i jak działa?

React Native to otwarty źródłowy framework firmy Meta (dawniej Facebook) do tworzenia aplikacji mobilnych przy użyciu JavaScript i TypeScript. W odróżnieniu od zwykłych aplikacji webowych opakowanych w WebView (np. Ionic), React Native kompiluje komponenty do PRAWDZIWYCH natywnych widżetów systemowych - przycisk w RN to prawdziwy Android Button, nie jego kopia rysowana na płotnie. W tym ćwiczeniu skupimy się na budowaniu aplikacji pogodowej WeatherApp, która pobiera dane z otwartego API Open-Meteo i wyświetla je w natywnym interfejsie.

Analogia do tłumacza na żywo

React Native działa jak tłumacz symultaniczny podczas międzynarodowej konferencji. Ty mówisz po angielsku (JavaScript), a tłumacz natychmiast przekazuje treść po polsku systemowi Android i po japońsku systemowi iOS. Słuchacze słyszą swój ojczysty język, czyli widzą prawdziwe natywne komponenty, nie imitacje.

1.1. Dwa mosty w architekturze React Native

React Native przeszedł gruntowną przebudowę architektury. Stara architektura (Bridge) i nowa (JSI + Fabric) różnią się fundamentalnie sposobem komunikacji między kodem JavaScript, a kodem natywnym.

Cecha	Stara architektura (bridge)	Nowa architektura (JSI + Fabric)
Komunikacja JS ↔ Native	Asynchroniczny, serializowany most JSON. Każde wywołanie to marshal/unmarshal danych.	Synchroniczne wywoływanie przez JSI (JavaScript Interface). Brak serializacji.
Renderowanie	Shadow Tree w JS, potem aktualizacja UI na wątku natywnym.	Fabric renderer: synchroniczne, współdzielone drzewo. Brak 'skoków' UI.
Wydajność	Bottle-neck: most JSON dla gestów i animacji.	Zbliżona do natywnej; animacje na 60/120 fps bez lagów.
Status (RN 0.76+)	Wyłączana domyślnie; dostępna jako opcja kompatybilności.	Domyślna od RN 0.76. Wymagana dla nowych projektów.

1.2. React Native, a inne cross-platformowe podejścia

Podejście	Jak to działa
React Native (JS/TS)	Most do natywnych widżetów. Prawdziwy UI platformy. Ogromna społeczność frontendowa. Język JS/TS znany web-deweloperom.
Flutter (Dart)	Własny silnik renderujący (Skia/Impeller). Identyczny wygląd na iOS i Android. Szybszy rendering, ale wygląd „nie platformowy” (nie natywny)
Expo (nad RN)	Warstwa nad React Native z gotowymi API (kamera, notyfikacje). Usuwa potrzebę Android Studio/Xcode dla prostych aplikacji. Rekomendowany na start.
Ionic / Capacitor	Aplikacja webowa w WebView. Używa HTML/CSS. Najłatwiejsze przejście z web, ale nienatywny wygląd i niższa wydajność.
Kotlin Multiplatform	Współdzieli logikę, UI osobny dla każdej platformy. Najlepsza integracja z natywnym systemem. Wymaga znajomości Kotlin i Swift.

Dlaczego JavaScript/TypeScript?

Miliony web-deweloperów znają JS. Meta chciała by mogli tworzyć aplikacje mobilne bez nauki nowego języka. TypeScript (nadzbiór JS z typami) rozwiązuje główną słabość JS, czyli brak typowania statycznego. W 2024 roku praktycznie każdy nowy projekt RN używa już TypeScript.

2. Instalacja i konfiguracja środowiska

React Native oferuje dwa sposoby pracy: Expo Go (szybki start bez Android Studio) i React Native CLI (pełna kontrola, wymagany Android Studio). W tym ćwiczeniu użyjemy React Native CLI z Expo jako narzędziem zarządzającym.

2.1. Wymagania wstępne

Wymaganie	Wersja minimalna / uwagi
Node.js	v18 LTS lub nowszy. Sprawdź: <code>node --version</code> . Użyj <code>nvm</code> do zarządzania wersjami.
npm lub Yarn	npm 9+ (z Node.js) lub Yarn 1.22+. W ćwiczeniu użyjemy npm.

Android Studio	Giraffe (2022.3.1) lub nowszy. Wymagany do Android SDK i emulatora AVD.
JDK	JDK 17 (Temurin/OpenJDK). Android Studio instaluje go automatycznie.
Android SDK	API level 33+ (Android 13). Dostępny przez Android Studio SDK Manager.
Git	2.x - do klonowania i zarządzania wersjami projektów.

2.2. Tworzenie projektu z Expo

Expo to framework i platforma dla React Native, która znacznie upraszcza początek pracy. Nie eliminuje Android Studio, ale zarządza dużą częścią konfiguracji natywnej automatycznie.

```
# Instalacja Expo CLI globalnie (jednorazowo)
```

```
npm install -g expo-cli
```

```
# Tworzenie nowego projektu z szablonem TypeScript:
```

```
npx create-expo-app@latest WeatherApp --template blank-typescript
```

```
# Przejdź do katalogu i uruchom:
```

```
cd WeatherApp
```

```
npx expo start
```

```
# Skróty klawiszowe w terminalu podczas expo start:
```

```
# a = Uruchom na Android emulatorze (wymagany AVD)
```

```
# i = Uruchom na iOS symulatorze (tylko macOS + Xcode)
```

```
# w = Uruchom w przeglądarce (wersja webowa)
```

```
# r = Reload (przeładuj JS bundle)
```

```
# d = Otwórz menu deweloperskie w emulatorze
```

Rozróżnienie pomiędzy Expo Go, a Development Build

Expo Go to aplikacja na telefon/emulator z preinstalowanymi bibliotekami. Wystarczy do większości ćwiczeń. Development Build to pełna wersja z Twoimi natywnymi modułami. Potrzebna jest, aby dołączyć biblioteki z kodem natywnym (np. react-native-camera).

2.3. Konfiguracja Android Studio i AVD

Na tym etapie przygotowujemy środowisko programistyczne umożliwiające tworzenie i testowanie aplikacji mobilnych. Android Studio stanowi zintegrowane środowisko (IDE), które zawiera kompilator, debugger oraz narzędzia do analizy wydajności. Kluczowym elementem jest także Android Virtual Device (AVD), czyli emulator urządzenia mobilnego, pozwalający uruchamiać aplikacje bez konieczności posiadania fizycznego sprzętu. Poprawna konfiguracja środowiska zapewnia zgodność z wybraną wersją systemu Android (API level) oraz umożliwia testowanie aplikacji w różnych warunkach sprzętowych i systemowych.

```
# Kroki konfiguracji Android Studio dla React Native:
```

```
1. Zainstaluj Android Studio (installer z developer.android.com/studio)
```

```
2. Podczas instalacji zaznacz: Android SDK, Android SDK Platform, Android Virtual Device
```

```
3. W SDK Manager (Tools > SDK Manager) zainstaluj:
```

```
- Android SDK Platform 34 (Android 14)
```

- Android SDK Build-Tools 34.0.0
- Android Emulator

```
# Zmienne srodowiskowe (dodaj do ~/.bashrc lub ~/.zshrc):
export ANDROID_HOME=$HOME/Android/Sdk # Linux
export ANDROID_HOME=$HOME/Library/Android/sdk # macOS
export PATH=$PATH:$ANDROID_HOME/emulator
export PATH=$PATH:$ANDROID_HOME/platform-tools
```

```
# Weryfikacja:
adb devices # Powinna wylistować emulator lub telefon
```

Fast Refresh w RN to odpowiednik Hot Reload we Flutterze

Zmiana w kodzie JS pojawia się w aplikacji w ciągu sekund bez utraty stanu. Działa automatycznie. Dla zmian w kodzie natywnym (java/kotlin) wymagany jest pełny restart (r w terminalu lub Reload w menu deweloperskim).

3. Struktura projektu React Native

Projekt stworzony przez create-expo-app ma z góry określoną strukturę katalogów. Zrozumienie roli każdego pliku jest kluczowe. Pomylenie ich to jeden z pierwszych błędów początkujących.

3.1. Mapa katalogów

Struktura katalogów projektu odzwierciedla logiczny podział aplikacji na warstwy oraz komponenty funkcjonalne. Pozwala to na utrzymanie czytelności kodu, łatwiejsze zarządzanie zależnościami oraz skalowanie projektu w miarę jego rozwoju. Typowy projekt zawiera katalogi odpowiedzialne za kod źródłowy, zasoby (np. layouty, grafiki), konfigurację oraz pliki build. Zrozumienie struktury katalogów jest kluczowe dla efektywnego poruszania się po projekcie i lokalizowania poszczególnych elementów aplikacji.

WeatherApp/	
├─ app/	← GŁÓWNY KATALOG EKRAŃÓW (Expo Router)
│ └─ (tabs)/	← Grupowanie tras w zakładkach
│ └─ index.tsx	← Ekran główny (zakładka Home)
│ └─ _layout.tsx	← Konfiguracja zakładek
│ └─ detail/	
│ └─ [city].tsx	← Ekran szczegółów (parametr dynamiczny!)
│ └─ _layout.tsx	← Korzeń nawigacji (Stack/Tabs)
├─ components/	← Współdzielone komponenty UI
│ └─ WeatherCard.tsx	
│ └─ ForecastTile.tsx	
├─ hooks/	← Własne React Hooks
│ └─ useWeather.ts	
│ └─ useFavorites.ts	
├─ services/	← Logika API (fetch, axios)
│ └─ weatherService.ts	
├─ types/	← Definicje typów TypeScript
│ └─ weather.ts	
├─ constants/	← Stałe (kolory, URL API, klucze)
│ └─ config.ts	
├─ assets/	← Obrazy, fonty, pliki statyczne
├─ app.json	← Konfiguracja Expo (nazwa, ikona, splash)
├─ tsconfig.json	← Konfiguracja TypeScript
└─ package.json	← Zależności (odpowiednik pubspec.yaml)

└─ babel.config.js

← Konfiguracja transpilatora Babel

3.2. Plik package.json, czyli manifest projektu

Plik `package.json` pełni rolę manifestu projektu i zawiera podstawowe informacje o aplikacji oraz jej zależnościach. Definiuje m.in. nazwę projektu, wersję, skrypty uruchomieniowe oraz listę bibliotek wykorzystywanych w aplikacji. Dzięki temu narzędzia takie jak npm lub yarn mogą automatycznie instalować wymagane pakiety oraz zarządzać ich wersjami. Poprawna konfiguracja tego pliku jest istotna dla powtarzalności procesu budowania i uruchamiania projektu na różnych środowiskach.

Klucz w package.json	Wyjaśnienie
name	Nazwa projektu (małe litery, myślniki). Identyfikator npm.
version	Wersja semantyczna: major.minor.patch. Wyświetlana w ustawieniach aplikacji.
main	Punkt wejścia: 'expo-router/entry' dla Expo Router. Expo wczytuje ten plik jako korzeń.
dependencies	Zależności RUNTIME (react, react-native, expo)
devDependencies	Zależności DEWELOPERSKIE tylko do kompilacji i testów (typescript, @types/*, jest).
scripts	Skróty poleceń: 'start', 'android', 'ios', 'web', 'test'. Uruchamiane przez: npm run <nazwa>.
expo.sdk	Wersja Expo SDK (w app.json). Definiuje dostępne natywne API.

4. TypeScript, podstawy języka w kontekście RN

TypeScript to nadzbiór JavaScript z opcjonalnym typowaniem statycznym. W React Native jest de facto standardem od 2022 roku. Jeśli znasz JavaScript, TypeScript będzie znajomy, różnice są głównie addytywne (dodajesz typy, nie przepisujesz logiki).

4.1. Typy i interfejsy

Typy oraz interfejsy stanowią podstawowy mechanizm definiowania struktury danych oraz kontraktów w aplikacji. Typy określają kształt obiektów (np. jakie pola zawierają i jakiego są typu), natomiast interfejsy pozwalają definiować wymagania wobec klas lub struktur implementujących dane zachowanie. Ich stosowanie zwiększa czytelność kodu, ułatwia wykrywanie błędów na etapie kompilacji oraz wspiera rozwój aplikacji zgodnie z zasadami programowania statycznie typowanego.

Kod TypeScript	Wyjaśnienie
// Podstawowe typy	TypeScript dodaje typy do zmiennych; JS ich nie ma
const city: string = 'Warszawa';	Explicit type annotation. Kompilator pilnuje że city jest zawsze stringiem.
const temp: number = 22.5;	number obejmuje int i float (jak Dart). Brak osobnych typów.
const isRaining: boolean = false;	boolean (małe litery) - nie Boolean jak w Javie.

<code>let maybeCity: string null = null;</code>	Union type: string lub null. Odpowiednik String? w Dart i Kotlin.
<code>// Interfejs - kształt obiektu</code>	Interfejsy definiują strukturę obiektów (jak data class)
<code>interface WeatherData {</code>	interface = deklaracja kształtu. Brak implementacji.
<code> city: string;</code>	Pole wymagane - musi być przy tworzeniu obiektu.
<code> temperature: number;</code>	Odpowiednik: <code>val temperature: Double</code> w Kotlin.
<code> humidity: number;</code>	Pole wymagane.
<code> icon?: string;</code>	? = opcjonalne. Może być undefined (brak pola) lub string.
<code>}</code>	

4.2. Typy dla komponentów React Native

Modele danych reprezentują strukturę informacji przetwarzanych w aplikacji, np. obiekty użytkownika, wpisy w bazie danych czy odpowiedzi z API. Są one często mapowane bezpośrednio na dane przechowywane lokalnie lub pobierane z sieci. Poprawne zaprojektowanie modeli danych pozwala na jednoznaczne odwzorowanie logiki biznesowej oraz ułatwia integrację między różnymi warstwami aplikacji (np. warstwą sieciową i bazodanową).

Kod TypeScript	Wyjaśnienie
<code>// Props komponentu; zawsze interfejs</code>	Props to są właściwości przekazywane do komponentu (jak parametry konstruktora Widgetu Flutter)
<code>interface WeatherCardProps {</code>	Konwencja: NazwaKomponentuProps
<code> weather: WeatherData;</code>	Wymagany prop - brak = błąd TypeScript
<code> onPress?: () => void;</code>	Opcjonalny callback. <code>() => void</code> = funkcja bez argumentów i wartości zwracanej. Jak <code>VoidCallback</code> w Dart.
<code> style?: StyleProp<ViewStyle>;</code>	Prop stylu - pozwala rodzicowi nadpisać styl. <code>StyleProp<ViewStyle></code> to typ wbudowany w RN.
<code>}</code>	
<code>// Typy specyficzne dla RN:</code>	Importowane z <code>react-native</code> lub <code>@types/react-native</code>
<code>import { ViewStyle, TextStyle } from 'react-native';</code>	<code>ViewStyle</code> = styl kontenera, <code>TextStyle</code> = styl tekstu. Kompilator pilnuje poprawnych propów CSS-like.

4.3. Funkcje, async/await i generyki

Walidacja danych polega na sprawdzaniu poprawności i spójności danych wejściowych przed ich dalszym przetwarzaniem. Może obejmować kontrolę typów, zakresów wartości, formatu (np. adres e-mail) oraz reguł biznesowych. Wprowadzenie walidacji minimalizuje ryzyko błędów w działaniu aplikacji oraz zwiększa bezpieczeństwo, szczególnie w kontekście danych pochodzących od użytkownika lub z zewnętrznych źródeł.

```
// Funkcja z typami - arrow function (najczęstszy styl w RN)
const formatTemp = (celsius: number): string => {
  return `${celsius.toFixed(1)}°C`; // Template literals: ${wyrażenie}
};
```

```
// Async/await - identyczny wzorzec jak w Dart
const fetchWeather = async (city: string): Promise<WeatherData> => {
  const response = await fetch(`${API_URL}?city=${city}`);
  if (!response.ok) throw new Error(`HTTP ${response.status}`);
  return response.json() as WeatherData; // asercja 'zaufaj mi, to WeatherData'
};

// Generyki - funkcja wielokrotnego użyciu z typem parametrycznym
const getFirst = <T>(items: T[]): T | undefined => items[0];
const firstCity = getFirst<string>(['Warszawa', 'Krakow']); // string | undefined

// Typ unii z discriminant - elegancka obsługa stanu asynchronicznego
type AsyncState<T> =
  | { status: 'idle' }
  | { status: 'loading' }
  | { status: 'success'; data: T }
  | { status: 'error'; message: string };

// Użycie - TypeScript automatycznie zawiąże typ w switch/if:
function render(state: AsyncState<WeatherData>) {
  if (state.status === 'success') {
    return state.data.city; // TypeScript wie że state.data istnieje!
  }
}
```

5. Komponenty są fundamentem UI w React Native

Jeśli Flutter opiera się na klasach Widget, to React Native opiera się na funkcjach komponentach. Komponent to funkcja, która przyjmuje props i zwraca JSX - specjalny syntax łączący JavaScript z opisem UI. Wszystko w React Native jest komponentem: tekst, przycisk, kontener, cały ekran.

Analogia do klocków LEGO z przepisem

Komponent w React Native to jak klocek LEGO z instrukcją. Wiesz co dostaniesz (props determinują wygląd), możesz go łączyć z innymi klockami i używać wielokrotnie. Każdy klocek zna tylko swoje wnętrze, a nie wie co jest na zewnątrz. Aplikacja to taka budowla z klocków.

5.1. Podstawowe komponenty RN -> ściągą

Podstawowe komponenty React Native stanowią odpowiedniki najczęściej używanych elementów interfejsu znanych z innych frameworków mobilnych. View pełni rolę uniwersalnego kontenera, Text służy do wyświetlania tekstu, Image do prezentacji grafik, a TextInput do wprowadzania danych przez użytkownika. Szczególnie istotne są także komponenty związane z interakcją i wydajnością, takie jak Pressable, FlatList, ScrollView, SafeAreaView czy ActivityIndicator, ponieważ tworzą fundament większości ekranów aplikacji.

Komponent RN	Odpowiednik w Compose	Opis i uwagi
View	FrameLayout / Box / Column / Row	Kontener ogólnego przeznaczenia. Nie ma własnego wyglądu. Odpowiednik <div> w HTML.
Text	TextView / Text (Compose)	JEDYNY sposób wyświetlania tekstu. Każdy tekst MUSI być w <Text>.

Image	ImageView / Image (Compo-se)	Obrazy lokalne (require()) i zdalne (uri:). Zawsze ustawia rozmiar przez style.
TextInput	EditText / TextField (Compo-se)	Pole tekstowe. Kontrolowane przez useState. Prop value + onChangeText.
ScrollView	ScrollView / LazyColumn (bez lazy)	Przewijana lista. Renderuje WSZYSTKIE dzieci od razu. Użyj FlatList dla długich list.
FlatList	RecyclerView / LazyColumn	„Leniwa lista/kolumna” - renderuje tylko widoczne elementy. Zawsze używaj dla długich list.
TouchableOpacitiy	Button / clickable Modifier	Element klikalny ze zmniejszaniem nieprzezroczystości. Użyj Pressable dla nowych projektów.
Pressable	Modifier.clickable / Button	Nowszy, elastyczny element klikalny. Ma stan pressed do stylowań.
SafeAreaView	WindowInsets / systemBar-spPadding	Chroni treść przed notchem, wyciekami i systemowym paskiem nawigacji.
ActivityIndicator	CircularProgressIndicator	Spinner ładowania. Odpowiednik Flutterowego CircularProgressIndicator.
Modal	AlertDialog / Dialog	Pływające okno modalne z animacją.

5.2. Komponent funkcyjny WeatherCard

Komponent WeatherCard jest przykładem typowego komponentu funkcyjnego w React Native, który przyjmuje dane przez props i na ich podstawie renderuje interfejs użytkownika. W prezentowanym wzorcu wykorzystano interfejs TypeScript do zdefiniowania struktury właściwości, destrukuryzację argumentów oraz komponent Pressable, który umożliwia obsługę kliknięcia i zmianę stylu w stanie naciśnięcia. Taki komponent dobrze ilustruje połączenie JSX, typowania statycznego oraz lokalnego stylowania przez StyleSheet.create.

components/WeatherCard.tsx

tsx

```
import React from 'react';
import { View, Text, StyleSheet, Pressable } from 'react-native';
import { WeatherData } from '../types/weather'; // nasz interfejs

// Definicja Props - ZAWSZE interfejs dla komponentów
interface WeatherCardProps {
  weather: WeatherData; // wymagany prop
  onPress?: () => void; // opcjonalny callback
}

// Komponent funkcyjny - arrow function z destrukuryzacją props
export const WeatherCard: React.FC<WeatherCardProps> = ({ weather, onPress }) => {
  // JSX - JavaScript + XML-like syntax do opisu UI
  return (
    <Pressable // klikalny kontener
      onPress={onPress} // {wywołanie JS} w JSX = klamry
      style={{ pressed }} => [ // styl zależy od stanu pressed
        styles.card,
        pressed && styles.pressed // && = warunkowy styl
      ]
    >
    <Text style={styles.city}>{weather.city}</Text>
    <Text style={styles.temp}>
      {`${weather.temperature.toFixed(1)}°C`} {/ template literal */}
    </Text>
    <Text style={styles.humidity}>
      Wilgotność: {weather.humidity}%
    </Text>
  )
}
```

```

    </Pressable>
  );
};

// StyleSheet.create - optymalizuje style (przetwarza raz, nie przy kazdym renderze)
const styles = StyleSheet.create({
  card: {
    backgroundColor: '#fff',
    borderRadius: 16,
    padding: 16,
    marginVertical: 8,
    elevation: 4, // Cień Android
    shadowColor: '#000', // Cień iOS
    shadowOffset: { width: 0, height: 2 },
    shadowOpacity: 0.1,
    shadowRadius: 8,
  },
  pressed: { opacity: 0.85 }, // Efekt naciśnięcia
  city: { fontSize: 22, fontWeight: '700', color: '#1a1a2e' },
  temp: { fontSize: 48, fontWeight: '300', color: '#0164FF' },
  humidity: { fontSize: 14, color: '#666' },
});

```

JSX, a Dart widgets

W Dart piszesz: `Text('Warszawa', style: TextStyle(fontSize: 22))`. W React Native piszesz: `<Text style={{fontSize: 22}}>Warszawa</Text>`. Różnice są składniowe, ale koncepcja identyczna - tworzysz hierarchie elementów UI z właściwościami. JSX kompiluje się do `React.createElement('Text', {style:..., 'Warszawa'})` - wywołań JS.

5.3. FlatList, czyli lekkie listy

`FlatList` jest standardowym komponentem do wydajnego wyświetlania długich list danych w React Native. W odróżnieniu od `ScrollView` nie renderuje od razu wszystkich elementów, lecz tylko te, które są aktualnie potrzebne, dzięki czemu lepiej skaluje się dla większych zbiorów danych. W praktyce wykorzystuje się go wraz z funkcją `renderItem`, mechanizmem `keyExtractor`, komponentami pustej listy i nagłówka oraz opcjami optymalizacyjnymi i odświeżania zawartości.

Komponent z FlatList - odpowiednik `ListView.builder` w Dart

```

import { FlatList, ListRenderItem } from 'react-native';

// Typ dla renderItem - TypeScript pomaga uniknąć błędów
const renderWeatherCard: ListRenderItem<WeatherData> = ({ item, index }) => (
  <WeatherCard
    weather={item}
    onPress={() => router.push(`/detail/${item.city}`)}
  />
);

<FlatList<WeatherData>
  data={weatherList} // tablica danych do wyświetlenia
  renderItem={renderWeatherCard} // funkcja tworząca element dla każdego item
  keyExtractor={({item}) => item.city} // unikalny klucz (jak key w Compose)
  // Stany specjalne:
  ListEmptyComponent={<Text>Brak danych</Text>}
  ListHeaderComponent={<SearchBar />} // Nagłówek listy (przewija się razem)
  // Odświeżanie (Pull-to-Refresh):
  refreshing={isLoading}
  onRefresh={loadData}
  // Optymalizacje:
  initialNumToRender={10} // Ile elementów renderować przy starcie
  windowSize={5} // Ile 'okien' ekranu w pamięci
/>

```

6. Stan i efekty - useState, useEffect i Context API

React Native używa systemu hooków do zarządzania stanem i efektami ubocznymi. Hooki to specjalne funkcje rozpoczynające się od 'use', które 'podpinają' komponent do mechanizmów Reacta. Są odpowiednikiem setState() + StatefulWidget z Fluttera, ale elegantsze.

6.1. useState, czyli lokalny stan komponentu

Hook useState służy do przechowywania lokalnego stanu komponentu i jego aktualizacji w sposób reaktywny. Pozwala kontrolować wartości takie jak tekst wpisywany przez użytkownika, liczniki, flagi logiczne czy dane załadowane asynchronicznie. W kontekście TypeScript szczególnie ważne jest jawne określanie typów dla stanów bardziej złożonych, na przykład takich, które mogą mieć wartość null.

Kod React Native	Wyjaśnienie
<code>import { useState } from 'react';</code>	Importuj hook useState z pakietu react
<code>const [count, setCount] = useState(0);</code>	Destrukuryzacja: [wartosc, setter] = useState(wartosc-Pocatkowa). Jak para state + setState() w Dart.
<code>const [city, setCity] = useState<string>('');</code>	Jawny typ generyczny. Bez niego TS wnioskuje typ z wartosci pocatkowej.
<code>const [weather, setWeather] =</code>	Stan nullable - przed załadowaniem danych jest null.
<code>useState<WeatherData null>(null);</code>	WeatherData null = typ unii. Odpowiednik WeatherData? W Dart/Kotlin.
<code>setCity('Gdansk');</code>	Aktualizuj stan. React automatycznie rerenderuje komponent. Jak setState() => ... w Dart.
<code>setCount(prev => prev + 1);</code>	Funkcjonalna aktualizacja - używaj gdy nowy stan zależy od poprzedniego! Bezpieczniejsze.

6.2. Efekty uboczne useEffect

Hook useEffect służy do realizacji efektów ubocznych, czyli operacji wykonywanych poza samym renderowaniem komponentu, takich jak pobieranie danych, subskrypcje czy porządkowanie zasobów. Może działać jednokrotnie po zamontowaniu komponentu, reagować na zmiany konkretnych zależności albo zwracać funkcję sprząającą uruchamianą przy odmontowaniu. W praktyce jest podstawowym narzędziem do łączenia interfejsu z logiką asynchroniczną i cyklem życia komponentu. Jest to odpowiednik initState() + dispose() z Fluttera, ale bardziej elastyczny (Flutter gorszy, buuu).

```
import { useEffect } from 'react';
```

```
// Przypadek 1: Wykonaj RAZ przy montowaniu komponentu (inicjalizacja)
useEffect(() => {
  loadWeatherData(); // wywołaj async funkcje
}, []); // [] = PUSTA tablica zależności = wykonaj raz
// Odpowiednik: void initState() w Flutter
```

```
// Przypadek 2: Wykonaj przy KAŻDEJ zmianie city
useEffect(() => {
  if (city) fetchWeather(city);
}, [city]); // [city] = uruchom efekt gdy city się zmieni
// Odpowiednik: didUpdateWidget lub watch() w Compose
```

```
// Przypadek 3: Cleanup (sprzątanie po sobie)
```

```
useEffect(() => {
  const subscription = locationService.subscribe(onLocation);
  // Funkcja cleanup - wywoływana PRZED następnym efektem lub odmontowaniem:
  return () => subscription.unsubscribe(); // Odpowiednik: dispose() w Dart
}, []);
```

// Przypadek 4: Pobieranie danych - pełny wzorzec

```
const [weather, setWeather] = useState<WeatherData | null>(null);
const [loading, setLoading] = useState(false);
const [error, setError] = useState<string | null>(null);
```

```
useEffect(() => {
  let cancelled = false; // zapobieganie race condition
  const load = async () => {
    setLoading(true);
    try {
      const data = await weatherService.fetch(city);
      if (!cancelled) setWeather(data);
    } catch (e) {
      if (!cancelled) setError(String(e));
    } finally {
      if (!cancelled) setLoading(false);
    }
  };
  load();
  return () => { cancelled = true; }; // anuluj jeżeli komponent odmontowany
}, [city]);
```

6.3. Context API - globalny stan bez Redux

Gdy wiele komponentów potrzebuje tych samych danych (np. lista ulubionych miast), użyj Context API. To wbudowany w React mechanizm globalnego stanu - odpowiednik Provider z Fluttera lub prostego wzorca ViewModel w Compose.

// context/FavoritesContext.tsx - definicja kontekstu

```
import React, { createContext, useContext, useState } from 'react';
```

// Typ kontekstu - co będziemy udostępniać

```
interface FavoritesContextType {
  favorites: string[];
  addFavorite: (city: string) => void;
  removeFavorite: (city: string) => void;
}
```

// Tworzenie kontekstu z wartością domyślną (undefined rzuca błąd jeżeli brak Providera)

```
const FavoritesContext = createContext<FavoritesContextType | undefined>(undefined);
```

// Provider - opakowuje dzieci i udostępnia im kontekst

```
export const FavoritesProvider: React.FC<{ children: React.ReactNode }> = ({ children }) => {
  const [favorites, setFavorites] = useState<string[]>(['Warszawa', 'Kraków']);
```

```
  const addFavorite = (city: string) => {
    if (!favorites.includes(city)) setFavorites(prev => [...prev, city]);
  };

```

```
  const removeFavorite = (city: string) => {
    setFavorites(prev => prev.filter(c => c !== city));
  };

```

```
  return (
    <FavoritesContext.Provider value={{ favorites, addFavorite, removeFavorite }}>
      {children} /* wszystkie zagnieżdżone komponenty mają dostęp do kontekstu */
    </FavoritesContext.Provider>
  );
};
```

// Custom hook - wygodny dostęp z walidacją

```
export const useFavorites = (): FavoritesContextType => {
```

```

const ctx = useContext(FavoritesContext);
if (!ctx) throw new Error('useFavorites musi być użyty w FavoritesProvider!');
return ctx;
};

// Użycie w komponencie - bez prop drilling!
const HomeScreen = () => {
  const { favorites, addFavorite } = useFavorites(); // bezpośredni dostęp
  // ...
};

```

Kiedy Context, Zustand, a kiedy Redux?

Context API wystarczy do prostego stanu globalnego. Dla złożonych aplikacji ze skomplikowanymi zależnościami między stanami warto rozważyć Zustand (prostszy od Redux) lub TanStack Query (do zarządzania cache danych z API). W tym ćwiczeniu Context API jest wystarczający.

7. React Navigation i Expo Router

React Native nie ma wbudowanej nawigacji - musisz dodać bibliotekę. Dwa najpopularniejsze rozwiązania to React Navigation (elastyczny, sprawdzony) i Expo Router (oparty na plikach, jak Next.js). W tym ćwiczeniu użyjemy Expo Router, który jest domyślny w projektach Expo i bardziej intuicyjny dla nowych deweloperów.

Analogia systemu adresów do fizycznej mapy

React Navigation to taka fizyczna mapa. Sam rysujesz drogi i definiujesz połączenia między miejscami. Expo Router to GPS z adresami URL: ścieżka pliku `app/detail/[city].tsx` automatycznie staje się URL `/detail/Warszawa`. Nie musisz ręcznie definiować tras, bo struktura katalogów jest już nawigacją.

7.1. Expo Router - nawigacja oparta na plikach

Expo Router realizuje nawigację na podstawie struktury katalogów i nazw plików w folderze `app/`. Oznacza to, że ścieżki nawigacyjne są budowane automatycznie na podstawie układu projektu, a pliki dynamiczne, takie jak `[city].tsx`, pozwalają definiować trasy z parametrami. Takie podejście upraszcza konfigurację, porządkuje architekturę aplikacji i ułatwia początkującym zrozumienie zależności między ekranami.

Plik	Trasa URL	Opis
<code>app/index.tsx</code>	<code>/</code>	Ekran główny aplikacji.
<code>app/detail/[city].tsx</code>	<code>/detail/Warszawa</code>	<code>[city]</code> = parametr dynamiczny. Odczytaj: <code>const { city } = useLocalSearchParams()</code>
<code>app/(tabs)/_layout.tsx</code>	Konfiguracja zakładek	Nawias = grupowanie bez wpływu na URL. Tabs layout.
<code>app/_layout.tsx</code>	Korzeń nawigacji	Stack/Tabs dla całej aplikacji. Zawsze obecny.
<code>app/+not-found.tsx</code>	Nieznana trasa	Ekran 404 dla nieznanych tras.

`app/_layout.tsx` - korzeń nawigacji aplikacji

tsx

```
import { Stack } from 'expo-router';
```

```
export default function RootLayout() {
  return (
    <Stack>
      <Stack.Screen
        name='index'
        options={{ title: 'WeatherApp', headerShown: true }}
      />
      <Stack.Screen
        name='detail/[city]'
        options={({ route }) => ({ title: route.params?.city as string })}
      />
    </Stack>
  );
}
```

// Nawigacja programatyczna - z dowolnego komponentu:

```
import { router, useLocalSearchParams } from 'expo-router';
```

// Przejsie do ekranu szczegółów:

```
router.push(`/detail/${cityName}`); // dodaj na stos (Back działa)
router.replace(`/detail/${cityName}`); // zastąp bieżący ekran (brak Back)
router.back(); // wroc do poprzedniego ekranu
```

// Odczyt parametru w [city].tsx:

```
const { city } = useLocalSearchParams<{ city: string }>();
// city = 'Warszawa' (z URL /detail/Warszawa)
```

7.2. Tab Navigator i zakładki

Nawigacja zakładkowa umożliwia szybkie przełączanie się między głównymi obszarami aplikacji, np. ekranem głównym i listą ulubionych. W Expo Router konfiguracja zakładek odbywa się najczęściej w pliku `app/(tabs)/_layout.tsx`, gdzie definiuje się nazwy ekranów, tytuły oraz ikony. Jest to wygodny sposób organizacji aplikacji, gdy użytkownik regularnie wraca do kilku kluczowych widoków.

app/(tabs)/_layout.tsx - konfiguracja zakładek dolnych

tsx

```
import { Tabs } from 'expo-router';
import { Ionicons } from '@expo/vector-icons'; // paczka ikon Expo

export default function TabLayout() {
  return (
    <Tabs
      screenOptions={{
        tabBarActiveTintColor: '#0164FF',
        tabBarStyle: { backgroundColor: '#fff' },
      }}
    >
      <Tabs.Screen
        name='index'
        options={{
          title: 'Pogoda',
          tabBarIcon: ({ color }) => <Ionicons name='partly-sunny' size={24} color={color} />,
        }}
      />
      <Tabs.Screen
        name='favorites'

```

```

    options={{
      title: 'Ulubione',
      tabBarIcon: ({ color }) => <Icons name='heart' size={24} color={color} />,
    }}
  />
</Tabs>
);
}

```

8. Pobieranie danych z REST API

WeatherApp będzie pobierać dane z Open-Meteo - darmowego, publicznego API pogodowego bez klucza API. Użyjemy natywnego `fetch()` dostępnego w React Native oraz omówimy popularną alternatywę `axios`.

8.1. Serwis API z `fetch()` i TypeScript

Serwis API porządkuje logikę komunikacji z zewnętrznym źródłem danych, oddzielając ją od komponentów interfejsu. W przedstawionym rozwiązaniu wykorzystano natywne `fetch()`, interfejsy TypeScript opisujące odpowiedzi API oraz funkcje pomocnicze do budowania zapytań i mapowania surowych danych JSON na model domenowy aplikacji. Takie podejście poprawia czytelność kodu, ułatwia obsługę błędów i wspiera ponowne użycie logiki sieciowej.

services/weatherService.ts

ts

```

const GEO_URL = 'https://geocoding-api.open-meteo.com/v1/search';
const WEATHER_URL = 'https://api.open-meteo.com/v1/forecast';

// Typy odpowiedzi API (surowy JSON od API)
interface GeoResponse {
  results?: Array<{ latitude: number; longitude: number; name: string }>;
}

interface WeatherResponse {
  current: {
    temperature_2m: number;
    relative_humidity_2m: number;
    wind_speed_10m: number;
    weather_code: number;
  };
}

// Funkcja pomocnicza - geolokalizacja nazwy miasta
async function getCoords(city: string): Promise<{ lat: number; lon: number }> {
  const params = new URLSearchParams({ // bezpieczne budowanie URL
    name: city, count: '1', language: 'pl',
  });
  const res = await fetch(`${GEO_URL}?${params}`);
  if (!res.ok) throw new Error(`Geolokalizacja: HTTP ${res.status}`);

  const data: GeoResponse = await res.json();
  if (!data.results?.length) throw new Error(`Miasto '${city}' nie znalezione`);

  const { latitude: lat, longitude: lon } = data.results[0];
  return { lat, lon };
}

// Główna funkcja - pobierz pogodę dla nazwy miasta
export async function fetchWeather(city: string): Promise<WeatherData> {
  const { lat, lon } = await getCoords(city); // krok 1: geolokalizacja

  const params = new URLSearchParams({
    latitude: String(lat),
    longitude: String(lon),
  });

```

```

    current: 'temperature_2m,relative_humidity_2m,wind_speed_10m,weather_code',
    timezone: 'Europe/Warsaw',
  });

  const res = await fetch(`${WEATHER_URL}?${params}`);
  if (!res.ok) throw new Error(`Pogoda: HTTP ${res.status}`);

  const data: WeatherResponse = await res.json();
  const c = data.current;

  // Mapuj surowy JSON na nasz model WeatherData
  return {
    city,
    temperature: c.temperature_2m,
    humidity: c.relative_humidity_2m,
    windSpeed: c.wind_speed_10m,
    weatherCode: c.weather_code,
    updatedAt: new Date(),
  };
}

```

8.2. Custom Hook; useWeather

Zamiast pisać `useEffect+useState` w każdym komponencie z osobna, wydzielamy logikę do Custom Hook. To wzorzec ponownego użytku logiki stanowej - odpowiednik `PersonalnaKlasaNotifera` w Riverpodzie.

hooks/useWeather.ts

ts

```

import { useState, useCallback } from 'react';
import { fetchWeather } from '../services/weatherService';
import { WeatherData } from '../types/weather';

// Typ stanu asynchronicznego
type WeatherState =
  | { status: 'idle' }
  | { status: 'loading' }
  | { status: 'success'; data: WeatherData }
  | { status: 'error'; message: string };

// Custom Hook - zawsze zaczyna się od 'use!'
export function useWeather() {
  const [state, setState] = useState<WeatherState>({ status: 'idle' });

  // useCallback - memoizuje funkcje (nie tworzy nowej przy każdym renderze)
  // Odpowiednik lazy val lub remember {} w Compose
  const fetch = useCallback(async (city: string) => {
    setState({ status: 'loading' });
    try {
      const data = await fetchWeather(city);
      setState({ status: 'success', data });
    } catch (e) {
      setState({ status: 'error', message: String(e) });
    }
  }, []); // [] = funkcja nie zależy od zewnętrznych zmiennych

  return { state, fetch };
}

// Użycie w komponencie - czyste i czytelne:
const HomeScreen = () => {
  const { state, fetch } = useWeather();

  if (state.status === 'loading') return <ActivityIndicator />;
  if (state.status === 'error') return <Text>Błąd: {state.message}</Text>;
  if (state.status === 'success') return <WeatherCard weather={state.data} />;
  return <Text>Wpisz miasto, aby sprawdzić pogodę.</Text>;
};

```

fetch() vs axios

Natywny `fetch()` jest wbudowany i wystarczy do prostych przypadków. `Axios` (npm install axios) oferuje automatyczną serializację JSON, interceptory (np. do doładowywania tokenów), ładniejszą obsługę błędów HTTP, timeout. W projektach produkcyjnych częściej używa się `axios`.

9. StyleSheet i nowoczesne podejścia do stylowania

React Native używa podzbioru CSS do stylowania, ale z kilkoma istotnymi różnicami. Nie ma kaskadowości (cascade) - style nie dziedziczą się automatycznie między komponentami. Każdy komponent musi być samodzielnie ostylowany.

9.1. Standardowe podejście dla StyleSheet.create

`StyleSheet.create` jest podstawowym i zalecanym mechanizmem definiowania stylów w React Native. Stylowanie przypomina CSS, ale różni się składnią i semantyką, m.in. stosowaniem `camelCase`, brakiem jednostek typu `px`, ograniczonym zakresem shorthandów i odmienną obsługą cieni na Androidzie oraz iOS. Korzystanie z tego podejścia porządkuje kod wizualny i ogranicza liczbę stylów tworzonych bezpośrednio w trakcie renderowania.

CSS (web)	React Native StyleSheet	Uwagi
<code>background-color: #fff</code>	<code>backgroundColor: '#fff'</code>	<code>camelCase</code> zamiast kebab-case. Standard RN.
<code>font-size: 16px</code>	<code>fontSize: 16</code>	Brak jednostek. Liczby = dp (density-independent pixels).
<code>margin: 8px 16px</code>	<code>marginVertical: 8, marginHorizontal: 16</code>	Brak shorthand. Osobne props dla każdego kierunku.
<code>display: flex; flex-direction: column</code>	(domyślne!)	View jest domyślnie flex column. Nie trzeba dodawać.
<code>border-radius: 8px</code>	<code>borderRadius: 8</code>	Identyczne (bez px).
<code>box-shadow: 0 2px 8px rgba(0,0,0,0.1)</code>	<code>elevation: 4</code> (Android) + <code>shadow*</code> (iOS)	Dwa oddzielne systemy cieni (patrz przykłady poniżej)
<code>position: absolute; top: 0</code>	<code>position: 'absolute', top: 0</code>	Absolutne pozycjonowanie działa tak samo jak w CSS.

9.2. Flexbox w React Native

React Native używa Flexbox jako jedynego systemu układu. Domyślna os jest pionowa (`flexDirection: 'column'`), co różni się od CSS gdzie domyślnie jest pozioma. Pamiętaj o tej różnicy!

Układy Flexbox w React Native

```
const styles = StyleSheet.create({
  // Kolumna (domyślne!) - elementy jeden pod drugim
  column: {
    flexDirection: 'column', // DOMYŚLNE w RN (w CSS domyślne to 'row')
    alignItems: 'center',    // Wyroównanie w poprzek osi (poziome dla column)
    justifyContent: 'space-between', // Rozmieszczenie wzdłuż osi (pionowe dla column)
  }
});
```

```

},

// Rząd - elementy obok siebie
row: {
  flexDirection: 'row',
  alignItems: 'center',    // Wyroównanie pionowe dla row
},

// Rozciągnięcie - zajmij dostępne miejsce (jak Expanded w Dart)
expand: { flex: 1 },

// Absolutne pozycjonowanie - badge DEV w rogu karty
devBadge: {
  position: 'absolute',
  top: 8, right: 8,
  backgroundColor: '#FF3B30',
  paddingHorizontal: 6, paddingVertical: 2,
  borderRadius: 4,
},
});

// Dynamiczne style - styl zależy od propów lub stanu:
const cardStyle = [
  styles.card,
  isHighlighted && styles.highlighted, // warunkowo dodaj styl
  { backgroundColor: theme.surface }, // obiekt stylu inline
];
<View style={cardStyle} />

```

9.3. Responsiveness, czyli Dimensions i useWindowDimensions

Responsywność w React Native polega na dostosowywaniu rozmiaru i układu komponentów do dostępnej przestrzeni ekranu. `Dimensions.get('window')` pozwala jednorazowo odczytać wymiary okna, natomiast `useWindowDimensions()` reaguje dynamicznie na ich zmianę, np. po obróceniu urządzenia. Dzięki temu można tworzyć interfejsy, które zachowują poprawny wygląd zarówno w orientacji pionowej, jak i poziomej.

```

import { Dimensions, useWindowDimensions } from 'react-native';

// Statyczna (nie aktualizuje się przy obrocie ekranu):
const { width, height } = Dimensions.get('window');

// Dynamiczna (aktualizuje się automatycznie) - używaj w komponentach:
const MyComponent = () => {
  const { width, height } = useWindowDimensions(); // hook!
  const isLandscape = width > height;

  return (
    <View style={{ width: width * 0.9 }}> {/* 90% szerokości ekranu */}
      <WeatherCard style={{ height: isLandscape ? 100 : 200 }} />
    </View>
  );
};

```

10. Architektura i struktura projektu WeatherApp

Połączmy wszystkie poznane elementy w spójną aplikację pogodową. WeatherApp pozwoli wyszukać miasto, wyświetli aktualną pogodę i prognozę na 7 dni, z możliwością zapisania ulubionych miast.

10.1. Pełna struktura katalogów projektu

Pełna struktura katalogów projektu porządkuje aplikację według odpowiedzialności poszczególnych plików i folderów. W przykładowym projekcie wydzielono m.in. katalog ekranów, komponentów, kontekstu, hooków, serwisów, typów i stałych. Taki układ ułatwia rozwój aplikacji, skraca czas odnajdywania kodu i wspiera utrzymanie czytelnej architektury.

```

WeatherApp/
├── app/
│   ├── _layout.tsx      ← FavoritesProvider + Stack nawigacji
│   ├── (tabs)/
│   │   ├── _layout.tsx  ← Konfiguracja zakładek (Home + Ulubione)
│   │   ├── index.tsx    ← Główny ekran: wyszukiwarka + lista
│   │   └── favorites.tsx ← Ekran ulubionych miast
│   └── detail/
│       └── [city].tsx    ← Ekran szczegółów pogody
├── components/
│   ├── WeatherCard.tsx  ← Karta aktualnej pogody
│   ├── ForecastTile.tsx ← Wiersz prognozy dziennej
│   ├── SearchBar.tsx    ← Pole wyszukiwania z przyciskiem
│   └── ErrorView.tsx    ← Widok błędu z przyciskiem Retry
├── context/
│   └── FavoritesContext.tsx ← Context + Provider + useFavorites hook
├── hooks/
│   ├── useWeather.ts    ← Logika pobierania pogody (aktualnej)
│   └── useForecast.ts    ← Logika pobierania prognozy 7 dni
├── services/
│   └── weatherService.ts ← Wywołania HTTP do Open-Meteo API
├── types/
│   └── weather.ts        ← Interfejsy: WeatherData, ForecastDay
├── constants/
│   └── config.ts         ← API_URL, COLORS, ICONS_MAP (kod→ikona)

```

10.2. Punkt startowy całej aplikacji, czyli app/_layout.tsx

Plik `app/_layout.tsx` pełni rolę głównego punktu wejścia do struktury ekranów i nawigacji w aplikacji opartej na Expo Router. To właśnie tam zwykle umieszcza się konfigurację stosu ekranów oraz globalne opakowanie aplikacji w dostawców stanu, motywu lub innych usług. W praktyce jest to centralne miejsce, w którym scala się strukturę aplikacji z mechanizmami współdzielonymi przez wszystkie widoki.

// app/_layout.tsx - korzeń całej aplikacji

tsx

```

import { Stack } from 'expo-router';
import { FavoritesProvider } from '../context/FavoritesContext';

// RootLayout musi być eksportem domyślnym
export default function RootLayout() {
  return (
    // Provider na SZCZYZIE drzewa - dostępny dla wszystkich ekranów
    <FavoritesProvider>
    <Stack>
      <Stack.Screen
        name='(tabs)' // Cała grupa zakładek jako jeden ekran
        options={{ headerShown: false }} // Zakładki mają własny nagłówek
      />
    </Stack>
  );
}

```

```

<Stack.Screen
  name='detail/[city]'
  options={({ route }) => ({
    title: (route.params as { city: string }).city,
    headerBackTitle: 'Wróć',
  })}
/>
</Stack>
</FavoritesProvider>
);
}

```

11. Zadania do wykonania

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

Zadanie 1. Środowisko i projekt bazowy

1. Zainstaluj Node.js (v18 LTS). Sprawdź: `node --version`. Skonfiguruj Android Studio z SDK API 33+ i utwórz AVD (Pixel 7, API 34).
2. Utwórz projekt: `npx create-expo-app@latest WeatherApp --template blank-typescript`. Uruchom na AVD: `npx expo start` → a.
3. Zapoznaj się ze strukturą projektu. Zmodyfikuj `app/index.tsx`: zmień tekst powitalny na 'WeatherApp by [twoje-imie]'. Sprawdź Fast Refresh.
4. Dodaj zależności: `npm install axios @expo/vector-icons`. Zainstaluj typy: `npm install -D @types/react-native`.
5. Utwórz plik `types/weather.ts` z interfejsem `WeatherData` (`city`, `temperature`, `humidity`, `windSpeed`, `weatherCode`, `updatedAt`: `Date`).

WERYFIKACJA: Screenshot AVD z tekstem powitalnym. `node --version` i `npx expo --version` bez błędów. Plik `types/weather.ts` z interfejsem.

Zadanie 2. Serwis API i custom hook

1. Utwórz `services/weatherService.ts` z funkcją `fetchWeather(city: string): Promise<WeatherData>`. Dwu-etapowo: geolokalizacja Open-Meteo → dane pogodowe.
2. Przetestuj serwis w `app/index.tsx` tymczasowo: wywołaj `fetchWeather('Warszawa')` w `useEffect` i wylóguj wynik przez `console.log`. Sprawdź logi w terminalu (Metro bundler).
3. Utwórz `hooks/useWeather.ts` z Custom Hookiem zwracającym `{ state, fetch }`. Stan jest unia discriminowana: `idle | loading | success | error`.
4. Dodaj obsługę błędów: rzuć `WeatherError(message)` dla `HTTP != 200` i pustych wyników geolokalizacji.

WERYFIKACJA: Dane pogodowe widoczne w konsoli Metro. Przetestuj z nieistniejącym miastem - aplikacja pokazuje komunikat błędu (nie crashuje).

Zadanie 3. Komponenty i pełne UI

1. Zaimplementuj `FavoritesContext` z listą ulubionych miast, metodami `addFavorite()` i `removeFavorite()`. Owijaj aplikację `FavoritesProvider` w `app/_layout.tsx`.
2. Skonfiguruj Expo Router: zakładki (tabs) z dwoma ekranami - Główny i Ulubione. Skonfiguruj Stack dla ekranu Detail.
3. Zbuduj `HomeScreen`: `SearchBar` (`TextInput` + przycisk), wyświetlanie stanów `loading/error/idle/success`, `FlatList` z przyciskami do dodawania ulubionych.
4. Zbuduj `WeatherCard`: wyświetla miasto, temperaturę (duży tekst), wilgotność, prędkość wiatru, ikona pogody na podstawie `weatherCode` (`Ionicons`).
5. Zbuduj `DetailScreen`: wyświetla wszystkie dane dla wybranego miasta + prognoza na 7 dni (osobne wywołanie API: `&daily=temperature_2m_max,temperature_2m_min`). Wyświetl 7 wierszy `ForecastTitle`.

WERYFIKACJA: Demo na AVD - wyszukanie Warszawy, widok szczegółów z prognoza 7 dni, nawigacja Back, dodawanie do ulubionych, ekran Ulubione z lista.

Zadanie 4. Ulepszenia i funkcje dodatkowe

1. Zaimplementuj persystencję ulubionych miast przez AsyncStorage (npm install @react-native-async-storage/async-storage). Ulubione mają przetrwać restart aplikacji.
2. Dodaj ciemny motyw: użyj useColorScheme() z react-native aby automatycznie przełączać kolory między jasnym a ciemnym systemem. Wszystkie kolory przez obiekt Colors (nie hardkodowane).
3. Zaimplementuj animacje przez Animated API: FlatList z animowanym wejściem kart (FadeIn z opóźnieniem $\text{index} * 80\text{ms}$). Karta WeatherCard zwiększa się przy naciśnięciu (scale 0.97).
4. Dodaj Pull-to-Refresh (refreshing + onRefresh na FlatList) odświeżający wszystkie ulubione miasta.

WERYFIKACJA: Ulubione przetrwają restart. Ciemny motyw przełącza się automatycznie. Animacje wejścia widoczne na liście. Pull-to-Refresh odświeża dane.

12. Kryteria oceniania

Zadanie	Punkty	Co weryfikuje prowadzący
Zad. 1: Środowisko	20 pkt	Działający AVD, projekt skompilowany, typy WeatherData, zależności dodane.
Zad. 2: API i Hook	25 pkt	fetchWeather() zwraca dane. Custom Hook z unią stanów. Obsługa błędu.
Zad. 3: UI	35 pkt	SearchBar, WeatherCard, DetailScreen z prognoza 7 dni, nawigacja, stany UI.
Zad. 4: Polisz	20 pkt	AsyncStorage, ciemny motyw, animacje wejścia, Pull-to-Refresh.
RAZEM	100 pkt	

Oce- na	Punkty / wymagania
5.0	90-100 pkt - Wszystkie zadania. Ciemny motyw, AsyncStorage, animacje. Brak hardkodowanych 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 + HomeScreen i DetailScreen (bez prognozy).
3.0	50-59 pkt - Zadania 1-2. API działa, dane wyświetlane.
2.0	0-49 pkt - Projekt nie kompiluje lub API nie zwraca danych.

13. Tabela porównawcza React Native vs Flutter

Jeśli znasz Flutter z poprzednich ćwiczeń, ta tabela pozwoli szybko znaleźć odpowiedniki. Różnice oczywiście są widoczne, ale same koncepcje są zaskakująco podobne.

Flutter / Dart	odpowiednik React Native / TypeScript
Widget (klasa)	Komponent (funkcja z JSX)
StatelessWidget	Komponent funkcyjny bez useState
StatefulWidget + State	Komponent z useState / useReducer
build(BuildContext ctx)	return (...JSX...) z funkcji komponentu
setState(() { _x = y; })	setX(y) - setter z useState
initState()	useEffect(() => { ... }, []) - pusta tablica
dispose()	return () => cleanup() z useEffect

Provider / ChangeNotifier	Context API + <code>useContext()</code>
Riverpod / AsyncNotifier	Zustand / TanStack Query (zewnętrzne)
Column / Row	View z <code>flexDirection: 'column' / 'row'</code>
<code>ListView.builder</code>	FlatList z <code>renderItem</code>
<code>Text(style: TextStyle(...))</code>	<code><Text style={styles.text}></code>
<code>ElevatedButton(onPressed: fn)</code>	<code><Pressable onPress={fn}></code>
<code>Image.network(url)</code>	<code><Image source={{ uri: url }} /></code>
GoRouter / Navigator	Expo Router / React Navigation
<code>context.push('/detail/\$city')</code>	<code>router.push(`/detail/\${city}`)</code>
<code>pubspec.yaml</code>	<code>package.json</code>
<code>flutter pub get</code>	<code>npm install</code>
<code>flutter run</code>	<code>npx expo start</code>
Hot Reload (r)	Fast Refresh (automatyczny)
SharedPreferences	AsyncStorage (@react-native-async-storage)
AnimationController + Tween	<code>Animated.Value</code> + <code>Animated.timing()</code>
AnimatedContainer	<code>Animated.View</code> ze stylem z <code>Animated.Value</code>
Hero widget	Shared Element Transition (react-navigation-shared-element)
<code>flutter test</code>	<code>npx jest</code> (lub <code>npm test</code>)
<code>EdgeInsets.all(16)</code>	<code>padding: 16</code> (StyleSheet)
<code>Colors.blue[500]</code>	<code>'#1565C0'</code> lub <code>theme.colors.primary</code>

14. Najczęstsze błędy i ich rozwiązania

Sekcja poświęcona najczęstszym błędom ma duże znaczenie dydaktyczne, ponieważ pokazuje problemy pojawiające się w realnej pracy z React Native. Obejmuje ona m.in. błędy związane z renderowaniem tekstu poza komponentem `Text`, odwołaniami do niezainicjalizowanych danych, problemami sieciowymi, niepoprawnym użyciem `useContext`, pętlami w `useEffect` czy błędami w imporcie modułów. Zrozumienie tych przypadków przyspiesza diagnozowanie usterek i uczy dobrych praktyk implementacyjnych.

Błąd / objaw	Przyczyna i rozwiązanie
Invariant Violation: Text strings must be rendered within a <code><Text></code> component	Próba wyświetlenia tekstu poza <code><Text></code> . Często przypadkowy string w JSX: <code>{ isLoading && 'Loading' }</code> - gdy <code>isLoading=false</code> , <code>false</code> jest ignorowane, ale 0 nie! Zamień na: <code>{ isLoading ? <Text>Loading</Text> : null }</code> .
Network request failed / TypeError: Network request failed	Brak połączenia sieciowego lub błąd CORS. Na Android emulatorze użyj 10.0.2.2 zamiast localhost. Sprawdź czy emulator ma dostęp do internetu: ustawienia emulatora > sieć.
Cannot read property 'X' of undefined / null	Użycie wartości zmiennych zanim dane zostaną załadowane. Dodaj warunek: <code>if (!weather) return <Loading /></code> . Lub użyj optional chaining: <code>weather?.city</code> .
<code>useContext</code> zwraca undefined	Komponent używający <code>useContext</code> jest poza Providerem. Sprawdź czy <code>FavoritesProvider</code> opakowuje aplikację w <code>_layout.tsx</code> , NIE w konkretnym ekranie.
FlatList - 'VirtualizedList' warning: You have a large	<code>ScrollView</code> opakowuje <code>FlatList</code> . Nie należy tego robić. Użyj <code>ListHeaderComponent</code> do dodania nagłówka wewnątrz <code>FlatList</code> zamiast owijac <code>FlatList</code> w

list	ScrollView.
useEffect działa w petli (infinite loop)	Tablica zależności zawiera obiekt lub tablice tworzone przy każdym renderze. Obiekty w JS porównywane są przez referencje. Użyj useMemo lub useCallback dla obiektów w zależności, lub wyodrębnij wartość prymitywną: [city.id] zamiast [city].
Expo: Unable to resolve module 'X'	Brakujący pakiet. Wykonaj: npm install X. Jeśli pakiet jest zainstalowany, sprawdź czy import jest poprawny (wielkość liter!). Restart Metro bundler: npx expo start --clear.
Keyboard pushes content up (blokuje UI)	Brak KeyboardAvoidingView. Owin formularz: <KeyboardAvoidingView behavior={Platform.OS === 'ios' ? 'padding' : 'height'}>. Na Androidzie można też ustawić android:windowSoftInputMode='adjustResize' w AndroidManifest.xml.
Image nie wyświetla się (biały prostokąt)	Brak wymiarów na Image. W RN Image MUSI mieć zdefiniowany width i height w style, inaczej ma rozmiar 0. Dodaj style={{ width: 200, height: 150 }} lub użyj resizeMode='contain'.
Cannot update a component from inside the function body of a different component	setState wywoływany podczas renderowania innego komponentu. Przenieś wywołanie do useEffect lub handlera zdarzenia.

14.1. Narzędzia diagnostyczne dla React Native

Narzędzia diagnostyczne w React Native wspierają analizę błędów, wydajności i poprawności konfiguracji projektu. Do najważniejszych należą Fast Refresh, menu deweloperskie Expo, React Native Debugger, Flipper, logi adb logcat, statyczne sprawdzanie typów przez tsc --noEmit oraz czyszczenie cache bundlera Metro. Ich regularne użycie pozwala szybciej lokalizować problemy oraz świadomie kontrolować jakość rozwijanej aplikacji.

```
# Fast Refresh to automatyczne odświeżanie zmian JS
# Shake emulator lub „d” w terminalu = menu deweloperskie

# React Native Debugger to taka przeglądarka z DevTools dla RN:
npm install -g react-native-debugger
# Lub użyj Flipper (Meta) - bardziej zaawansowane debugowanie

# Expo DevTools - diagnostyka przez przeglądarkę
npx expo start # Otwórz http://localhost:8081/_expo/

# Logi na emulatorze Android (Logcat)
adb logcat *:S ReactNative:V ReactNativeJS:V

# TypeScript - sprawdzenie typów bez budowania
npx tsc --noEmit

# Analiza bundla - co zajmuje miejsce
npx expo export && npx react-native-bundle-visualizer

# Reset cache Metro bundler (gdy coś dziwnego się dzieje)
npx expo start --clear
```

Instrukcja React Native 1 - TypeScript, Komponenty, Nawigacja i REST API

Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki

MobileHub

Następne ćwiczenie: React Native 2 - Zustand, Animacje, AsyncStorage i Testy

POLITECHNIKA RZESZOWSKA
KATEDRA INFORMATYKI I AUTOMATYKI
DR INŻ. MATEUSZ POMIANEK**ĆWICZENIE LABORATORYJNE REACT NATIVE 2****TypeScript · Komponenty · Nawigacja · REST API***Programowanie Aplikacji Mobilnych · moduł RN***Projekt WeatherApp v2: rozszerzenie projektu z Lab 1***Wymagania wstępne: ukończony RN 1 (TypeScript, komponenty, nawigacja, Context API, REST API)***Spis treści**

1. Od Context API do Zustand - dlaczego warto przejść?.....	1
2. Zustand - zarządzanie stanem globalnym.....	2
3. TanStack Query - zarządzanie stanem serwera.....	5
4. AsyncStorage i MMKV - lokalna persystencja danych.....	7
5. Animacje niejawne - Animated API.....	8
6. Animacje jawne - React Native Reanimated 3.....	10
7. Testy jednostkowe - Jest i React Native Testing Library.....	12
8. Mockowanie zależności - jest.mock() i MSW.....	15
9. Architektura projektu - Feature-First.....	16
10. Dostępność i internacjonalizacja.....	18
11. React Native DevTools - diagnostyka i profilowanie.....	20

1. Od Context API do Zustand - dlaczego warto przejść?

W React Native Lab 1 zarządzaliśmy stanem globalnym przez Context API. Context jest świetny na start, ale przy rozroście aplikacji ujawnia ograniczenia: każda zmiana stanu rerenderuje WSZYSTKICH konsumentów kontekstu, niezależnie od tego, czy zmieniony fragment ma dla nich znaczenie. Zustand rozwiązuje ten problem elegancko i jest jednym z najpopularniejszych rozwiązań stanu globalnego w ekosystemie React w 2024 roku.

Analogia: biuletyn informacyjny vs. newsletter tematyczny

Context API to biuletyn firmowy: każdy pracownik dostaje cały newsletter przy każdej zmianie, nawet jeśli interesuje go tylko kolumna sportowa. Zustand to newsletter tematyczny z subskrypcjami: każdy subskrybuje tylko te działy, które go interesują. Zmiana pogody nie powiadamia czytelników sportu.

1.1. Problemy Context API w dużych aplikacjach

Context API jest wygodnym rozwiązaniem dla prostego stanu globalnego, ale w większych projektach zaczyna ujawniać ograniczenia architektoniczne i wydajnościowe. Każda zmiana wartości kontekstu może powodować renderowanie wielu konsumentów, nawet jeśli korzystają tylko z niewielkiego fragmentu danych, a rozbudowana struktura providerów szybko prowadzi do tzw. „provider hell”. Z tego powodu w bardziej złożonych aplikacjach często zastępuje się je lżejszymi bibliotekami do zarządzania stanem.

Problem z Context API	Jak Zustand to rozwiązuje
Każda zmiana dowolnego pola rerenderuje WSZYSTKICH konsumentów kontekstu.	Zustand pozwala subskrybować się tylko na wybrane pole: <code>useWeatherStore(s => s.city)</code> . Zmiana <code>humidity</code> nie rerenderuje komponentu który czyta tylko <code>city</code> .
Przy wielu kontekstach drzewo Providerów staje się 'Provider Hell': <code><A><C><D>...</code>	Zustand nie wymaga Providerów. Store to globalny singleton - dostępny z dowolnego miejsca bez owijania aplikacji.
Brak wbudowanego wsparcia dla danych asynchronicznych (loading/error).	Zustand łączy się natywnie z TanStack Query do zarządzania stanem serwera. Każdy ma swoją rolę.
Trudne testowanie - trzeba owijać komponenty w Providery z odpowiednimi wartościami.	Zustand store można zresetować między testami jednym wywołaniem. Proste i izolowane.

1.2. Konfiguracja Zustand

Zustand jest biblioteką stanu globalnego, która wyróżnia się bardzo prostą konfiguracją i niewielkim narzutem. W podstawowym wariancie wymaga jedynie instalacji pakietu i nie wymusza owijania całej aplikacji dodatkowymi providerami. Dzięki temu dobrze nadaje się jako pierwszy krok po Context API w projektach, które zaczynają rosnąć pod względem liczby ekranów i współdzielonych danych.

```
# Instalacja (jeden pakiet, zero konfiguracji)
npm install zustand

# Opcjonalnie - immer dla niemutowalnych aktualizacji (jak w Redux Toolkit):
npm install immer

# Zustand NIE wymaga żadnych zmian w _layout.tsx ani owijania aplikacji!
# Store jest dostępny natychmiast po zaimportowaniu.
```

2. Zarządzanie stanem globalnym w Zustand

Zarządzanie stanem globalnym w Zustand polega na definiowaniu centralnych store'ów, które przechowują zarówno dane, jak i akcje modyfikujące ten stan. Rozwiązanie to upraszcza przepływ danych między ekranami i eliminuje konieczność ręcznego przekazywania właściwości przez wiele poziomów komponentów. W praktyce Zustand dobrze sprawdza się dla danych klienckich, takich jak preferencje, lista ulubionych czy stan interfejsu współdzielony między ekranami.

2.1. Tworzenie store - podstawowy wzorzec

Podstawowy store w Zustand definiuje strukturę stanu oraz funkcje odpowiedzialne za jego aktualizację. Typowy wzorzec obejmuje interfejs opisujący dane i akcje, wywołanie funkcji `create()` oraz niemutowalne aktualizacje realizowane przez `set()`. Takie podejście porządkuje logikę aplikacji i sprawia, że operacje na stanie są łatwe do odczytania i testowania.

Kod Zustand	Wyjaśnienie
// stores/favoritesStore.ts	Jeden plik = jeden store. Konwencja: useFavorites + Store.
import { create } from 'zustand';	<code>create()</code> to jedyna funkcja z pakietu <code>zustand</code> - tworzy hook store.
interface FavoritesState {	Interfejs opisuje CAŁY store: stan + akcje razem.
favorites: string[];	Stan: tablica nazw ulubionych miast.
addCity: (city: string) => void;	Akcja: metoda zmieniająca stan. Typ jak metoda interfejsu.
removeCity: (city: string) => void;	Akcja usuwająca miasto z listy.
clear: () => void; }	Akcja zerująca cały stan - przydatna w testach.
export const useFavoritesStore = create<FavoritesState>()(set => ({	<code>create<Typ>()(fn)</code> - podwójne nawiasy: zewnętrzne to middleware (<code>persist</code>), wewnętrzne to fabryka.
favorites: ['Warszawa', 'Krakow'],	Stan początkowy - inferencja typów z interfejsu.
addCity: (city) => set(state => ({	<code>set()</code> - JEDYNA droga do aktualizacji stanu. Zawsze tworzy nowy obiekt (niemutowalność).
favorites: state.favorites.includes(city)	Guard: nie dodawaj duplikatów.
? state.favorites	Brak zmiany = Zustand NIE rerenderuje.
: [...state.favorites, city]	Spread operator - nowa tablica z nowym miastem.
})),	
removeCity: (city) => set(state => ({	Filtrowanie zamiast mutacji.
favorites: state.favorites.filter(c => c !== city) })),	<code>filter()</code> zwraca nową tablicę - Zustand wykryje zmianę.
clear: () => set({ favorites: [] },));	Skrót: jeśli nie potrzebujesz poprzedniego stanu, podaj obiekt bezpośrednio.

2.2. Selektory, czyli subskrypcja tylko na to co potrzebne

Selektory są jedną z najważniejszych zalet Zustand, ponieważ pozwalają komponentowi subskrybować wyłącznie konkretny fragment store'a. Dzięki temu zmiana jednego pola nie wymusza niepotrzebnego renderowania wszystkich komponentów korzystających z całego obiektu stanu. Jest to kluczowy mechanizm optymalizacyjny, szczególnie istotny w większych aplikacjach mobilnych.

Użycie w komponencie - trzy sposoby, różna wydajność:

```
// SPOSOB 1: Selektor - NAJLEPSZY (rerender tylko gdy favorites się zmieni)
const favorites = useFavoritesStore(state => state.favorites);
```

```
const addCity = useFavoritesStore(state => state.addCity);

// SPOSOB 2: Destrukturyzacja - ZLY (rerender przy kazdej zmianie store!)
const { favorites, addCity } = useFavoritesStore(); // anti-pattern!

// SPOSOB 3: useShallow - gdy potrzebujesz wielu pol bez rerenderow
import { useShallow } from 'zustand/react/shallow';
const { favorites, addCity, removeCity } = useFavoritesStore(
  useShallow(state => ({ favorites: state.favorites, addCity: state.addCity, removeCity: state.removeCity }))
);

// Wywołanie akcji (identyczne jak setState w Context):
<Pressable onPress={() => addCity('Gdansk')}>
  <Text>Dodaj do ulubionych</Text>
</Pressable>
```

2.3. Middleware persist - automatyczna persystencja

Zustand ma wbudowany middleware persist, który automatycznie zapisuje i wczytuje stan z AsyncStorage. Odpowiednik ręcznie obsługiwanego Hive.box w Dart/Flutter, ale bez generowania kodu.

```
import { create } from 'zustand';
import { persist, createJSONStorage } from 'zustand/middleware';
import AsyncStorage from '@react-native-async-storage/async-storage';

export const useFavoritesStore = create<FavoritesState>() ( // 1. generyk stanu
  persist( // 2. middleware persist
    (set) => ({ // 3. fabryka stanu
      favorites: [],
      addCity: (city) => set(s => ({ favorites: s.favorites.includes(city) ? s.favorites : [...s.favorites, city] })),
      removeCity: (city) => set(s => ({ favorites: s.favorites.filter(c => c !== city) })),
      clear: () => set({ favorites: [] }),
    }),
    {
      name: 'favorites-storage', // klucz w AsyncStorage
      storage: createJSONStorage(() => AsyncStorage), // adapter dla RN
      partialize: (state) => ({ favorites: state.favorites }), // zapisz TYLKO dane, nie akcje
    }
  )
);

// To wszystko! Zustand automatycznie:
// - Odczytuje stan z AsyncStorage przy uruchomieniu aplikacji
// - Zapisuje do AsyncStorage przy kazdej zmianie favorites
// - Brak useEffect, brak recznie pisanego loadData()/saveData()
```

Optymalizacja przez persist z MMKV

AsyncStorage jest asynchroniczne i stosunkowo wolne. Dla lepszej wydajności użyj MMKV (react-native-mmkv) - synchroniczne, 10x szybsze od AsyncStorage. Wymiana to zmiana jednej linii: storage: createJSONStorage(() => new MMKV()). MMKV omówione w sekcji 4.

2.4. Wzorzec: store asynchroniczny ze stanami loading/error

Asynchroniczny store w Zustand rozszerza klasyczne zarządzanie stanem o obsługę ładowania danych i błędów. Oprócz właściwych danych przechowuje on zwykle informacje o aktywnych operacjach, komunikatach błędów i funkcjach odpowiedzialnych za pobieranie zasobów. Taki wzorzec ułatwia budowę bardziej odpornych ekranów oraz czytelną obsługę stanów przejściowych w interfejsie użytkownika.

```
// stores/weatherStore.ts - store z danymi asynchronicznymi
import { create } from 'zustand';
import { fetchWeather } from '../services/weatherService';
import { WeatherData } from '../types/weather';

interface WeatherState {
  // Stan danych
  weatherMap: Record<string, WeatherData>; // miasto -> dane pogodowe
  // Stan ładowania
  loadingCities: Set<string>; // ktore miasta sa aktualnie ladowane
  errors: Record<string, string>; // miasto -> komunikat bledu
  // Akcje
  loadWeather: (city: string) => Promise<void>;
  clearError: (city: string) => void;
}

export const useWeatherStore = create<WeatherState>()((set, get) => ({
  weatherMap: {},
  loadingCities: new Set(),
  errors: {},

  loadWeather: async (city) => {
    // Guard: nie laduj jesli juz w trakcie ladowania
    if (get().loadingCities.has(city)) return;

    set(s => ({ // Ustaw loading = true
      loadingCities: new Set([...s.loadingCities, city]),
      errors: { ...s.errors, [city]: undefined }, // Skasuj stary blad
    }));

    try {
      const data = await fetchWeather(city);
      set(s => ({
        weatherMap: { ...s.weatherMap, [city]: data },
        loadingCities: new Set([...s.loadingCities].filter(c => c !== city)),
      }));
    } catch (e) {
      set(s => ({
        errors: { ...s.errors, [city]: String(e) },
        loadingCities: new Set([...s.loadingCities].filter(c => c !== city)),
      }));
    }
  },

  clearError: (city) => set(s => ({ errors: { ...s.errors, [city]: undefined } })),
});

// Selektory pomocnicze (poza hookiem - bez rerenderow przy odczycie):
export const selectWeather = (city: string) => (s: WeatherState) => s.weatherMap[city];
export const selectLoading = (city: string) => (s: WeatherState) => s.loadingCities.has(city);
```

```
export const selectError = (city: string) => (s: WeatherState) => s.errors[city];

// Użycie w komponencie:
const weather = useWeatherStore(selectWeather('Warszawa')); // WeatherData | undefined
const loading = useWeatherStore(selectLoading('Warszawa')); // boolean
```

3. Zarządzanie stanem serwera przez TanStack Query

TanStack Query (dawniej React Query) to biblioteka specjalizująca się w zarządzaniu 'stanem serwera' - danymi pobranymi z API. Rozróżnia między stanem klienta (ulubione miasta - Zustand) a stanem serwera (aktualna pogoda - TanStack Query). To rozróżnienie jest kluczowe w profesjonalnych aplikacjach mobilnych.

Analogia kelnera vs. szefa kuchni

Zustand to kelner: zarządza tym, co jest na sali (stan klienta: koszyk, preferencje, UI). TanStack Query to szef kuchni: wie kiedy danie jest gotowe, jak długo wytrzyma w ciepłe i kiedy trzeba ugotować od nowa (cache, stale time, refetch). Kelner nie pyta szefa kuchni co minute czy jedzenie jest gotowe - dostaje powiadomienie.

3.1. Konfiguracja i podstawowy useQuery

TanStack Query służy do zarządzania stanem danych pobieranych z serwera i cache'owania ich po stronie klienta. Konfiguracja polega na utworzeniu QueryClient, opakowaniu aplikacji QueryClientProvider oraz definiowaniu zapytań za pomocą hooków takich jak useQuery. W efekcie komponent otrzymuje gotowy zestaw informacji o stanie zapytania, np. dane, błąd, ładowanie i możliwość ręcznego odświeżenia.

```
# Instalacja:
npm install @tanstack/react-query

// app/_layout.tsx - konfiguracja QueryClient
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 1000 * 60 * 15, // dane 'świeże' przez 15 minut
      gcTime: 1000 * 60 * 30, // cache trzymany 30 minut po ostatnim użyciu
      retry: 2, // ponow 2 razy przy błędzie sieci
    },
  },
});

export default function RootLayout() {
  return (
    <QueryClientProvider client={queryClient}>
      <FavoritesProvider>
        <Stack>{/* ... */}</Stack>
      </FavoritesProvider>
    </QueryClientProvider>
  );
}
```

```

}

// hooks/useWeatherQuery.ts - zapytanie do API
import { useQuery } from '@tanstack/react-query';
import { fetchWeather } from '../services/weatherService';

export function useWeatherQuery(city: string) {
  return useQuery({
    queryKey: ['weather', city], // Unikalny klucz cache. MUSI zawierac parametry!
    queryFn: () => fetchWeather(city), // Funkcja pobierajaca dane
    enabled: city.length > 0, // Nie pobieraj gdy city jest puste
    staleTime: 1000 * 60 * 10, // Dane swieże przez 10 minut (nadpisuje default)
  });
}

// Użycie w komponencie - wszystkie stany wbudowane
const { data, isLoading, isError, error, refetch } = useWeatherQuery('Warszawa');
//   ^^^^^ ^^^^^^^^^^^^^ ^^^^^^^^^ ^^^^^ ^^^^^^^^^
//   Dane Ładowanie Błąd Obiekt Ręcznie odśwież

```

3.2. Kiedy Zustand, kiedy TanStack Query?

Zustand i TanStack Query rozwiązują różne problemy, dlatego nie należy traktować ich jako bezpośrednich zamienników. Zustand najlepiej nadaje się do stanu klienta, czyli danych lokalnych i trwałych, niezależnych od API, natomiast TanStack Query obsługuje stan serwera, czyli dane pobierane z backendu, posiadające czas życia, cache i mechanizmy odświeżania. Rozróżnienie tych ról prowadzi do bardziej przejrzystej i profesjonalnej architektury aplikacji.

Typ stanu	Narzędzie	Przykład dla WeatherApp
Stan klienta - trwały, niezależny od sieci	Zustand (+ persist)	Lista ulubionych miast, motyw (jasny/ciemny), preferencje języka
Stan serwera - pobierany z API, ma TTL	TanStack Query	Aktualna pogoda, prognoza 7 dni, lista miast z geolokalizacji
Stan UI - lokalny, efemeryczny	useState w komponencie	Rozwinięty/zwinięty akordeon, tekst w polu wyszukiwania, wybrany tab
Stan formularza	React Hook Form lub useState	Formularz dodawania miasta z walidacją

4. AsyncStorage i MMKV - lokalna persystencja danych

Każda aplikacja mobilna potrzebuje przechowywać dane lokalnie: ustawienia, dane użytkownika, cache. React Native oferuje kilka rozwiązań, od prostych klucz-wartość po relacyjne bazy danych.

Rozwiązanie	Typ	Kiedy używać	Zady i wady
AsyncStorage	Klucz-wartość, async	Proste dane użytkownika, ustawienia, cache tokenów (< 6 MB).	Wolne (async I/O). Nie szyfrować. Wbudowany w Expo.
MMKV	Klucz-wartość, sync	Wszystko co AsyncStorage, ale szybciej. Zalecany w nowych projektach.	10x szybszy. Synchroniczny API. Wymaga native build.

Expo SecureStore	Klucz-wartość, szyfrowany	Tokeny JWT, hasła, dane wrażliwe. Używa Keychain (iOS) / Keystore (Android).	Bezpieczny. Limit 2048 bajtów na wartość.
SQLite (expo-sqlite)	Relacyjna baza	Złożenie zapytania, relacje, duże dane strukturalne, offline sync.	Pełny SQL. Bardziej boilerplate. API async od SDK 50.
WatermelonDB	Relacyjna + reactive	Duże zestawy danych z synchronizacją z backendem. Lazy loading.	Leniwa ewaluacja. Skomplikowana konfiguracja.

4.1. Bezpośrednie użycie AsyncStorage

AsyncStorage jest prostym mechanizmem lokalnej persystencji danych w modelu klucz–wartość. Umożliwia zapisywanie prostych informacji, takich jak ustawienia, flagi czy zserializowane obiekty JSON, a także ich późniejszy odczyt i usuwanie. Jego bezpośrednie użycie dobrze ilustruje podstawy lokalnego przechowywania danych, choć w większych projektach często bywa abstrahowane przez bibliotekę stanu lub osobny serwis.

```
import AsyncStorage from '@react-native-async-storage/async-storage';

// ZAPIS - zawsze string. Obiekty serializuj przez JSON:
await AsyncStorage.setItem('theme', 'dark');
await AsyncStorage.setItem('user', JSON.stringify({ name: 'Jan', age: 25 }));

// ODCZYT - zwraca string | null:
const theme = await AsyncStorage.getItem('theme'); // 'dark' lub null
const raw = await AsyncStorage.getItem('user');
const user = raw ? JSON.parse(raw) : null; // deserializacja

// USUWANIE:
await AsyncStorage.removeItem('theme');

// GRUPOWE operacje - batch (szybszy niż wiele pojedynczych):
await AsyncStorage.multiSet([['key1', 'val1'], ['key2', 'val2']]);
const values = await AsyncStorage.multiGet(['key1', 'key2']);
// values = [['key1', 'val1'], ['key2', 'val2']]

// WSZYSTKIE klucze:
const keys = await AsyncStorage.getAllKeys();

// CZYSZCZENIE (uwaga: usuwa wszystkie dane aplikacji!):
await AsyncStorage.clear();
```

4.2. MMKV, czyli szybsza alternatywa

MMKV (Memory Mapped Key-Value) to biblioteka od Tencent, oryginalnie napisana dla WeChat. Jest synchroniczna (brak async/await), co upraszcza kod i eliminuje race conditions.

Instalacja MMKV

```
npm install react-native-mmkv
npx expo prebuild # Wymagany native build (nie działa w Expo Go!)

// Tworzenie instancji (singleton):
import { MMKV } from 'react-native-mmkv';
export const storage = new MMKV({ id: 'weather-app-storage' }); // id = izolacja
```

```
// SYNCHRONICZNE API - brak await!
storage.set('theme', 'dark'); // zapis string
storage.set('fontSize', 16); // zapis number
storage.set('notifications', true); // zapis boolean
storage.set('user', JSON.stringify({})); // obiekty: JSON

const theme = storage.getString('theme'); // string | undefined
const size = storage.getNumber('fontSize'); // number | undefined
const notif = storage.getBoolean('notifications'); // boolean | undefined

storage.delete('theme'); // usuwanie
storage.clearAll(); // czyszczenie

// Integracja z Zustand persist (zamiana AsyncStorage na MMKV):
import { StateStorage } from 'zustand/middleware';

const mmkvStorage: StateStorage = {
  getItem: (name) => storage.getString(name) ?? null,
  setItem: (name, value) => storage.set(name, value),
  removeItem: (name) => storage.delete(name),
};

// W persist middleware:
storage: createJSONStorage(() => mmkvStorage),
```

5. Animacje niejawne - Animated API

React Native ma wbudowane Animated API do tworzenia animacji. Działa podobnie do AnimatedController z Fluttera - tworzysz wartości animowane (Animated.Value) i przyłączasz je do właściwości komponentów. Animacje wykonywane są natywnie (poza wątkiem JS) co gwarantuje 60fps nawet przy obciążonej logice.

Animacje niejawne vs jawne

W React Native rozróżniamy Animated API (odpowiednik AnimatedContainer/AnimatedOpacity z Fluttera - prostsze) oraz Reanimated 3 (odpowiednik AnimationController, ale bardziej zaawansowany, pełna kontrola, worklet na wątku UI). Zaczynij od Animated API dla prostych animacji, przejdź do Reanimated gdy potrzebujesz gestów lub skomplikowanych sekwencji.

5.1. Animated.Value i podstawowe animacje

Animated.Value stanowi podstawową jednostkę animacji w klasycznym Animated API React Native. Reprezentuje wartość zmieniającą się w czasie i może być powiązana z właściwościami widoku, np. przezroczystością, położeniem lub skalą. W połączeniu z Animated.timing i useNativeDriver pozwala budować płynne, wydajne animacje dla prostych scenariuszy interfejsowych.

Kod React Native	Wyjaśnienie
<code>const opacity = useRef(new Animated.Value(0)).current;</code>	useRef zapobiega tworzeniu nowej wartości przy każdym renderze. <code>.current</code> dostaje samą wartość.
<code>Animated.timing(opacity, {</code>	<code>timing()</code> = liniowa animacja do wartości docelowej w zadanym czasie.
<code>toValue: 1,</code>	Wartość docelowa animacji (od 0 do 1 = pełna nieprzezroczystość).

duration: 400,	Czas trwania w milisekundach. 200-400ms to 'naturalny' zakres.
easing: Easing.inOut(Easing.ease),	Krzywa animacji. Odpowiednik Curves.easeInOut z Fluttera.
useNativeDriver: true,	KLUCZOWE: true = animacja na watku UI (60fps). false = na wątku JS (laguje). Zawsze true gdy tylko możliwe.
}).start();	.start() uruchamia animację. Opcjonalnie callback: .start() => console.log('gotowe').
<Animated.View style={{ opacity }}>	Animated.View = natywny widok z animowanymi właściwościami. Tylko Animated.* komponenty przyjmują Animated.Value.

5.2. Animacje wejścia listy z staggered delay

Animacje wejścia listy z opóźnieniem typu staggered delay polegają na tym, że kolejne elementy pojawiają się sekwencyjnie, a nie jednocześnie. Dzięki temu lista sprawia wrażenie bardziej dynamicznej i uporządkowanej wizualnie. W praktyce taki efekt buduje się przez niezależne wartości animowane dla pozycji i przezroczystości oraz opóźnienie zależne od indeksu elementu.

Animowane wejście kart - każda karta przesuwana się z prawa z opóźnieniem

```
const CardItem: React.FC<{ weather: WeatherData; index: number }> = ({ weather, index }) => {
  const translateX = useRef(new Animated.Value(80)).current; // startowe przesunięcie
  const opacity = useRef(new Animated.Value(0)).current;
```

```
  useEffect(() => {
    // Staggered delay: karta N czeka index * 80ms
    const delay = index * 80;
```

```
    Animated.parallel([ // parallel = animuj jednocześnie translateX I opacity
      Animated.timing(translateX, {
        toValue: 0, // docelowo: brak przesunięcia
        delay,
        duration: 350,
        easing: Easing.out(Easing.back(1.5)), // efekt lekkiego 'odbicia' na końcu
        useNativeDriver: true,
      }),
      Animated.timing(opacity, {
        toValue: 1,
        delay,
        duration: 300,
        useNativeDriver: true,
      }),
    ]).start();
  }, []);
```

```
  return (
    <Animated.View style={{ opacity, transform: [{ translateX }] }}>
      <WeatherCard weather={weather} />
    </Animated.View>
  );
};
```

5.3. Animated.spring - efekt sprężynowy

Animated.spring służy do tworzenia animacji o charakterze fizycznym, przypominających ruch sprężyny. Jest szczególnie przydatny przy interakcjach użytkownika, np. podczas wciskania kart,

przycisków lub elementów listy, ponieważ nadaje im bardziej naturalny i responsywny charakter. Pozwala regulować zachowanie animacji parametrami takimi jak szybkość czy stopień odbicia.

„Scale” przy naciśnięciu daje efekt 'kliknięcia'

```
const PressableCard: React.FC<WeatherCardProps> = ({ weather, onPress }) => {
  const scale = useRef(new Animated.Value(1)).current;

  const handlePressIn = () => {
    Animated.spring(scale, {
      toValue: 0.95,      // zmniejsz do 95%
      useNativeDriver: true,
      speed: 50,          // szybkość sprężynowań
      bounciness: 0,      // brak odbicia (0 = jak easeOut)
    }).start();
  };

  const handlePressOut = () => {
    Animated.spring(scale, {
      toValue: 1,         // wróć do 100%
      useNativeDriver: true,
      speed: 50,
      bounciness: 4,      // lekkie odbicie przy powrocie
    }).start();
  };

  return (
    <Pressable onPress={onPress} onPressIn={handlePressIn} onPressOut={handlePressOut}>
      <Animated.View style={[styles.card, { transform: [{ scale }] }]>
        <WeatherCard weather={weather} />
      </Animated.View>
    </Pressable>
  );
};
```

6. Animacje jawne przez React Native Reanimated 3

React Native Reanimated 3 to zaawansowana biblioteka animacji działająca w całości na wątku UI (Worklets). Umożliwia płynne animacje gestów bez komunikacji z wątkiem JS. To taki odpowiednik AnimationController + GestureDetector z Fluttera, ale znacznie bardziej ergonomiczny.

Analogia: pilot zdalnego sterowania vs. autopilot

Animated API to pilot zdalnego sterowania: twój kod JS wysyła komendy przez radio (bridge) do wątku UI. Gdy JS jest zajęty (parsowanie JSON, renderowanie), komendy się opóźniają to animacja laguje. Reanimated Worklets to autopilot: kod animacji działa bezpośrednio w samolocie (wątku UI) bez oczekiwania na sygnał z zewnątrz. 60fps gwarantowane nawet gdy JS jest zajęty.

6.1. Konfiguracja i podstawy Reanimated

React Native Reanimated to bardziej zaawansowane rozwiązanie animacyjne niż klasyczne Animated API. Jego konfiguracja obejmuje instalację biblioteki, poprawne ustawienie w Babel oraz restart bundlera z wyczyszczeniem cache. Biblioteka ta jest szczególnie cenna tam, gdzie wymagane są bardzo płynne animacje, gesty i bardziej rozbudowane sekwencje ruchu.

```
# Instalacja:
npm install react-native-reanimated

# Konfiguracja Babel (babel.config.js) - Reanimated MUSI być ostatnim pluginem!
module.exports = {
  presets: ['babel-preset-expo'],
  plugins: ['react-native-reanimated/plugin'], // MUSI być na końcu!
};

# Restart Metro z wyczyszczeniem cache po konfiguracji Babel:
npx expo start --clear
```

6.2. Podstawowe wzorce stanowią useSharedValue i useAnimatedStyle

useSharedValue i useAnimatedStyle to dwa podstawowe mechanizmy pracy z Reanimated. Pierwszy przechowuje wartość współdzieloną między logiką a warstwą animacji, drugi zaś pozwala przeliczać styl komponentu na podstawie tej wartości. Dzięki temu możliwe jest sterowanie transformacjami i innymi cechami widoku bez wymuszania klasycznych renderów Reacta.

Reanimated 3	Wyjaśnienie
import Animated, {	Reanimated eksportuje zastępczy Animated - zamień import z react-native.
useSharedValue,	useSharedValue = odpowiednik Animated.Value, ale żyje na wątku UI.
useAnimatedStyle,	
withTiming, withSpring,	Funkcje animacji - odpowiedniki Animated.timing() itp.
withDelay,	
Easing,	Krzywe animacji - identyczne jak w Animated API.
} from 'react-native-reanimated';	Zastąpienie importu - reszta kodu pozostaje taka sama.
const offset = useSharedValue(0);	SharedValue = wartość współdzielona między wątkiem JS i UI. Brak rerenderów przy zmianie.
const style = useAnimatedStyle(()	useAnimatedStyle = HOF zwracający styl na podstawie SharedValues. Wykonuje się na wątku UI.
=> ({	
transform: [{ translateX:	.value = odczyt wartości. Zawsze użyj .value w workletach.
offset.value }],	
}));	Wynik jest używany jako style={style} w Animated.View.
// Uruchomienie animacji:	Przypisanie withTiming/withSpring do .value uruchamia animację.
offset.value = withTiming(100,	Animacja do 100 w 400ms. Brak .start() - automatyczne.
{ duration: 400 });	
offset.value = withSpring(-100,	Sprężynowa animacja do -100. damping = tłumienie sprężynowania.
{ damping: 12 });	

6.3. Obsługa gestów - Gesture Handler + Reanimated

Połączenie react-native-gesture-handler z Reanimated umożliwia budowanie zaawansowanych interakcji opartych na gestach, takich jak przeciąganie, przesuwanie czy

odrzuć element. Gesty są analizowane niezależnie od klasycznej logiki interfejsu, a ich wynik może bezpośrednio sterować animacją. Takie podejście jest szczególnie przydatne w nowoczesnych interfejsach mobilnych, gdzie gest stanowi pełnoprawny element nawigacji i obsługi danych.

```
# Instalacja Gesture Handler (wymagana współpraca z Reanimated):
npm install react-native-gesture-handler

// Owin aplikację w GestureHandlerRootView (app/_layout.tsx):
import { GestureHandlerRootView } from 'react-native-gesture-handler';
<GestureHandlerRootView style={{ flex: 1 }}>
  <Stack />{/* reszta aplikacji */}
</GestureHandlerRootView>

// Swipeable card - ciagniecie gestem w lewo (usuwanie z ulubionych):
import { Gesture, GestureDetector } from 'react-native-gesture-handler';
import Animated, { useSharedValue, useAnimatedStyle,
  withTiming, withSpring, runOnJS } from 'react-native-reanimated';

const SwipeableCard: React.FC<{ city: string; onDelete: () => void }> = ({ city, onDelete }) => {
  const translateX = useSharedValue(0);
  const SWIPE_THRESHOLD = -100; // Po przesunięciu o 100px w lewo - usun

  const panGesture = Gesture.Pan()
    .activeOffsetX([-10, 10]) // Aktywuj gest po 10px przesunięcia (rozróżnia od scroll)
    .onUpdate((event) => {
      translateX.value = Math.min(0, event.translationX); // Tylko w lewo (min 0)
    })
    .onEnd((event) => {
      if (event.translationX < SWIPE_THRESHOLD) {
        // Przelecieć poza ekran i wywołać usuwanie
        translateX.value = withTiming(-500, { duration: 200 });
        runOnJS(onDelete)(); // Wywołaj JS callback z wątku UI
      } else {
        translateX.value = withSpring(0); // Wróć na miejsce
      }
    });

  const animatedStyle = useAnimatedStyle(() => ({
    transform: [{ translateX: translateX.value }],
  }));

  return (
    <GestureDetector gesture={panGesture}>
      <Animated.View style={[styles.card, animatedStyle]}>
        <Text>{city}</Text>
      </Animated.View>
    </GestureDetector>
  );
};
```

runOnJS() to kluczowa funkcja

Worklets działają na wątku UI bez dostępu do stanu JS (useState, Zustand). Aby wywołać funkcję JS z Workletu (np. usunąć element ze store), użyj runOnJS(mojCallback)(). Jest to 'most' z wątku UI do wątku JS.

7. Testy jednostkowe - Jest i React Native Testing Library

React Native używa Jest jako test runnera i React Native Testing Library (RNTL) do testowania komponentów. W odróżnieniu od testów instrumentowanych Androida (emulator wymagany), testy RNTL działają na Node.js. Są szybkie i nie wymagają urządzenia.

Jak RNTL testuje bez emulatora?

React Native Testing Library renderuje komponenty w środowisko testowym (jest-native) które symuluje drzewo komponentów bez prawdziwego silnika renderującego. Sprawdza drzewo semantyczne, teksty i interakcje przez akcesory. Szybkie porównanie: testy jednostkowe Jest ~1ms, testy RNTL ~50-200ms, testy instrumentowane Detox ~5-30s na emulatorze.

7.1. Konfiguracja środowiska testowego

Środowisko testowe dla React Native opiera się zwykle na Jest oraz React Native Testing Library. Konfiguracja obejmuje instalację odpowiednich pakietów, ustawienie presetu dla Expo lub React Native oraz dodanie rozszerzeń wspierających asercje dla komponentów natywnych. Celem tej konfiguracji jest umożliwienie szybkiego uruchamiania testów bez potrzeby używania emulatora lub urządzenia fizycznego.

```
# Instalacja (jesli nie ma z Expo):
npm install --save-dev @testing-library/react-native
npm install --save-dev @testing-library/jest-native # dodatkowe matchery

# jest.config.js (lub 'jest' w package.json):
module.exports = {
  preset: 'jest-expo', // preset Expo z konfiguracją React Native
  setupFilesAfterFramework: ['@testing-library/jest-native/extend-expect'],
  transformIgnorePatterns: [
    'node_modules/(?!(jest-)?react-native|@react-native(-community)?|expo|@expo|zustand)'
  ],
};
```

7.2. Anatomia testu komponentu

Test komponentu w React Native zwykle składa się z przygotowania danych testowych, wyrenderowania komponentu, wyszukania elementów w drzewie interfejsu oraz sprawdzenia oczekiwanego zachowania. Taki test może weryfikować zarówno poprawność renderowanego tekstu, jak i reakcję na interakcje użytkownika, np. kliknięcie. Dzięki temu testy komponentów wspierają stabilność interfejsu i ograniczają ryzyko regresji.

```
// __tests__/WeatherCard.test.tsx
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react-native';
import { WeatherCard } from '../components/WeatherCard';
import { WeatherData } from '../types/weather';

// Dane testowe (fixture) - poza describe, współdzielone między testami
const mockWeather: WeatherData = {
```

```
city: 'Warszawa',
temperature: 22.5,
humidity: 65,
windSpeed: 12.0,
weatherCode: 1,
updatedAt: new Date('2024-01-15'),
};

describe('WeatherCard', () => {

  it('wyswietla nazwe miasta', () => {
    render(<WeatherCard weather={mockWeather} />);

    // screen.getByText - znajdz element po tekście (rzuca błąd jeśli nie znalazł)
    expect(screen.getByText('Warszawa')).toBeTruthy();
  });

  it('wyswietla temperature z jednostka', () => {
    render(<WeatherCard weather={mockWeather} />);
    expect(screen.getByText('22.5°C')).toBeTruthy();
  });

  it('wywołuje onPress po tapnięciu', () => {
    const mockOnPress = jest.fn(); // funkcja-szpieg

    render(<WeatherCard weather={mockWeather} onPress={mockOnPress} />);

    // fireEvent.press = symulacja tapnięcia
    fireEvent.press(screen.getByText('Warszawa'));

    expect(mockOnPress).toHaveBeenCalledTimes(1);
  });

  it('nie wyswietla przycisku onPress gdy prop nie podany', () => {
    render(<WeatherCard weather={mockWeather} />);
    // queryById zwraca null zamiast rzucać błąd - bezpieczne dla 'nie istnieje'
    expect(screen.queryById('press-action')).toBeNull();
  });
});
```

7.3. Finders, Queries i Events - API RNTL

API React Native Testing Library dostarcza zestaw metod do wyszukiwania elementów i symulowania działań użytkownika. Metody `getBy`, `queryBy` i `findBy` różnią się zachowaniem w sytuacji braku elementu oraz wsparciem dla scenariuszy asynchronicznych. Zrozumienie priorytetów selektorów i sposobów wywoływania zdarzeń jest podstawą tworzenia czytelnych i wiarygodnych testów.

Query	Zachowanie gdy nie znaleziono	Kiedy używać
<code>getBy*()</code>	Rzuca błąd (fail testu)	Gdy element MUSI istnieć. Najczęściej używane.
<code>queryBy*()</code>	Zwraca null	Gdy element może nie istnieć. Używaj z <code>.toBeNull()</code> .
<code>findBy*()</code>	Czeka (async), rzuca po timeout	Elementy pojawiające się asynchronicznie (po fetch).

getAllBy*()	Rzuca gdy nie ma żadnego	Gdy oczekujesz wielu elementów.
queryAllBy*()	Zwraca []	Lista może być pusta - bezpieczne.
findAllBy*()	Czeka (async), zwraca []	Lista asynchroniczna.

Selektor	Co szuka	Priorytet (wg RNTL)
ByRole()	Rola dostępności (button, text, textbox)	1 - Najwyższy. Testuje tak jak użytkownicy.
ByLabelText()	Etykieta accessibilityLabel	2 - Testuje dostępność
ByText()	Widoczny tekst (dokładne dopasowanie lub regex)	3 - Często używany.
ByPlaceholderText()	Tekst placeholder w TextInput	4 - Dla formularzy.
ByTestId()	Prop testID='...' na komponencie	5 - Ostatnia deska ratunku. Unikaj jeśli można.

7.4. Testowanie asynchroniczności

Testowanie asynchroniczności obejmuje scenariusze, w których interfejs reaguje na dane pobierane z API, opóźnione efekty lub zmianę stanu po czasie. W takich przypadkach stosuje się narzędzia takie jak `waitFor`, odpowiednie providery testowe oraz kontrolę stanów pośrednich, np. spinnera ładowania lub komunikatu o błędzie. Tego typu testy są niezbędne w aplikacjach korzystających z hooków sieciowych i cache danych.

```
// __tests__/HomeScreen.test.tsx - testowanie useQuery/useEffect
import { render, screen, waitFor, fireEvent } from '@testing-library/react-native';
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

// Helper: owij komponent w wymagane providery
const createWrapper = () => {
  const queryClient = new QueryClient({
    defaultOptions: { queries: { retry: false } }, // bez retry w testach!
  });
  return ({ children }: { children: React.ReactNode }) => (
    <QueryClientProvider client={queryClient}>{children}</QueryClientProvider>
  );
};

it('wyświetla dane po załadowaniu', async () => {
  render(<HomeScreen />, { wrapper: createWrapper() });

  // Na początku: spinner ładowania
  expect(screen.getByRole('progressbar')).toBeTruthy(); // ActivityIndicator

  // waitFor: czekaj aż element się pojawi (max 1000ms domyślnie)
  await waitFor(() => {
    expect(screen.getByText('Warszawa')).toBeTruthy();
  });

  // Po załadowaniu: spinner zniknął
  expect(screen.queryByRole('progressbar')).toBeNull();
});
```

```
it('wyswietla blad przy bledzie sieci', async () => {
  render(<HomeScreen city='NIEISTNIEJACE_MIASTO' />, { wrapper: createWrapper() });

  await waitFor(() => {
    expect(screen.getByText(/Blad/)).toBeTruthy(); // regex: zawiera 'Bład'
  });
});
```

8. Mockowanie zaleznosci - jest.mock() i MSW

Testy powinny być szybkie, izolowane i powtarzalne. Prawdziwe wywołania HTTP są powolne i niedeterministyczne (API może być niedostępne). Mockowanie pozwala zastąpić zewnętrzne zależności kontrolowanymi obiektami testowymi.

8.1. Mockowanie modułów jest.mock() więc jest mock.

`jest.mock()` pozwala zastąpić rzeczywiste zależności ich kontrolowanymi odpowiednikami używanymi wyłącznie podczas testu. Dzięki temu testy nie są uzależnione od zewnętrznych usług, opóźnień sieci ani niestabilności środowiska. Mockowanie modułów jest szczególnie ważne dla warstwy API, ponieważ pozwala deterministycznie sprawdzać zarówno scenariusze sukcesu, jak i błędów.

```
// Mockowanie całego modułu serwisu:
jest.mock('../services/weatherService', () => ({
  fetchWeather: jest.fn(), // Pusta funkcja-szpieg
}));

// Import mockowanego modułu - będziemy konfigurować odpowiedzi:
import { fetchWeather } from '../services/weatherService';
const mockFetchWeather = fetchWeather as jest.MockedFunction<typeof fetchWeather>;

// W tekście - konfiguracja odpowiedzi
describe('useWeatherQuery', () => {
  beforeEach(() => {
    jest.clearAllMocks(); // Resetuj liczniki i konfiguracje między testami
  });

  it('zwraca dane przy sukcesie', async () => {
    // Arrange: skonfiguruj co mock ma zwrócić
    mockFetchWeather.mockResolvedValue({
      city: 'Warszawa', temperature: 22.5, humidity: 65,
      windSpeed: 12, weatherCode: 1, updatedAt: new Date(),
    });

    // Act: uruchom hook przez renderHook
    const { result } = renderHook(() => useWeatherQuery('Warszawa'),
      { wrapper: createWrapper() }
    );

    // Assert: czekaj na sukces
    await waitFor(() => expect(result.current.isSuccess).toBe(true));
    expect(result.current.data?.city).toBe('Warszawa');
    expect(mockFetchWeather).toHaveBeenCalledWith('Warszawa');
  });

  it('zwraca blad przy niepowodzeniu sieci', async () => {
```

```
mockFetchWeather.mockRejectedValue(new Error('Network Error'));

const { result } = renderHook(() => useWeatherQuery('Warszawa'),
  { wrapper: createWrapper() }
);

await waitFor(() => expect(result.current.isError).toBe(true));
expect(result.current.error?.message).toBe('Network Error');
});
});
```

8.2. Testowanie Zustand store

Testowanie store'a Zustand polega na bezpośrednim sprawdzaniu zmian stanu po wywołaniu akcji. Tego typu testy są zwykle prostsze niż testowanie rozwiązań opartych na providerach, ponieważ store można łatwo zresetować i użyć w izolacji. Pozwala to szybko zweryfikować poprawność operacji takich jak dodawanie, usuwanie czy blokowanie duplikatów.

```
// __tests__/favoritesStore.test.ts
import { useFavoritesStore } from '../stores/favoritesStore';
import { act, renderHook } from '@testing-library/react-native';

// Resetuj store między testami (Zustand persystuje między testami w tym samym pliku!)
beforeEach(() => {
  useFavoritesStore.setState({ favorites: [] }); // Czyszcimy stan
});

it('dodaje miasto do ulubionych', () => {
  const { result } = renderHook(() => useFavoritesStore());

  act(() => {
    result.current.addCity('Gdansk'); // act() = 'otocz akcje zmieniająca stan'
  });

  expect(result.current.favorites).toContain('Gdansk');
  expect(result.current.favorites).toHaveLength(1);
});

it('nie dodaje duplikatów', () => {
  const { result } = renderHook(() => useFavoritesStore());

  act(() => {
    result.current.addCity('Krakow');
    result.current.addCity('Krakow'); // drugi raz ten sam!
  });

  expect(result.current.favorites).toHaveLength(1); // tylko jeden
});

it('usuwa miasto z ulubionych', () => {
  useFavoritesStore.setState({ favorites: ['Warszawa', 'Krakow'] });
  const { result } = renderHook(() => useFavoritesStore());

  act(() => {
    result.current.removeCity('Warszawa');
  });
});
```

```
expect(result.current.favorites).not.toContain('Warszawa');
expect(result.current.favorites).toContain('Krakow');
});
```

Uruchamianie testów

```
npm test           # Wszystkie testy (tryb watch)
npm test -- --watchAll=false # Jednorazowe uruchomienie (CI)
npm test -- WeatherCard    # Tylko pliki zawierające 'WeatherCard'
npm test -- --coverage      # Raport pokrycia kodu
npm test -- --verbose       # Wyświetl nazwy każdego testu
```

9. Architektura projektu Feature-First

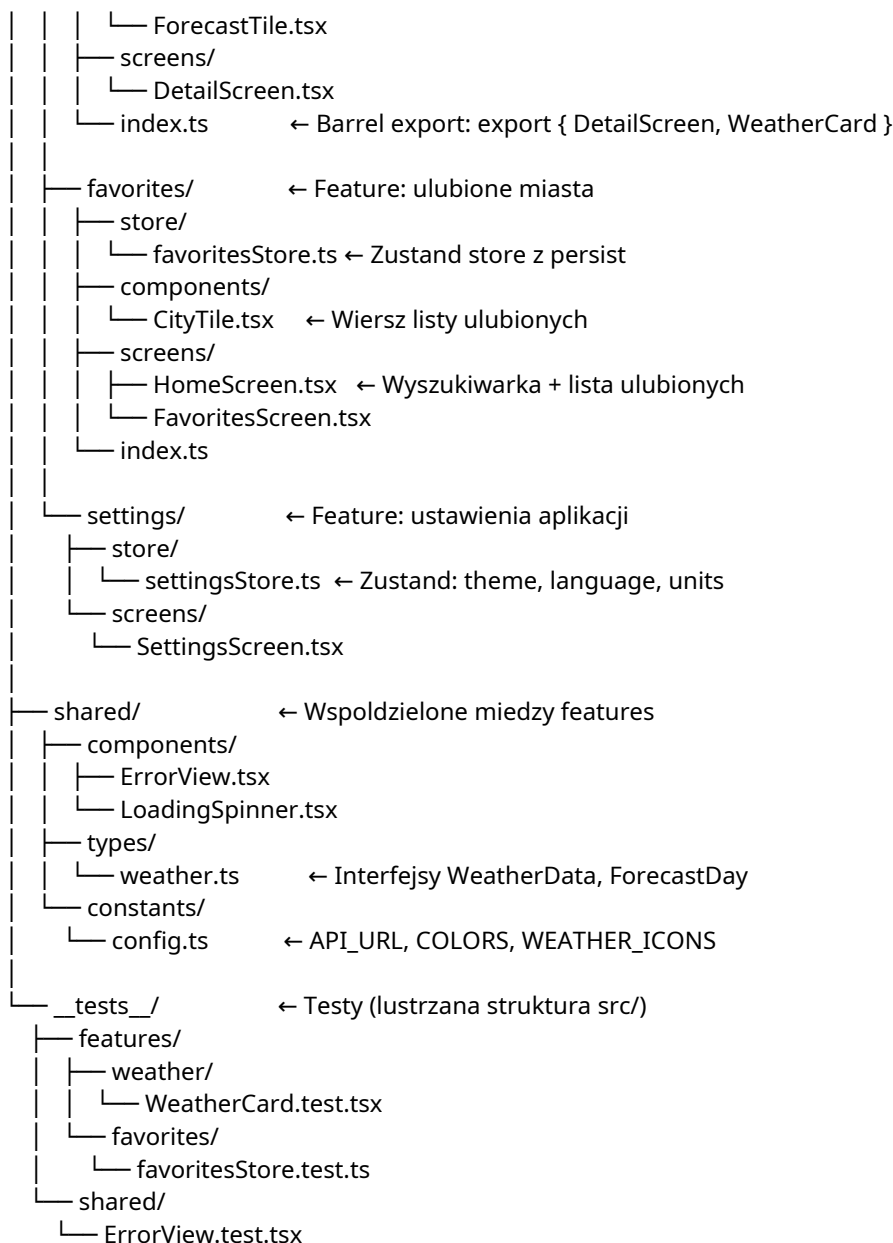
W RN Lab 1 używaliśmy prostej struktury z katalogami `components/`, `hooks/`, `services/`. Ta struktura, zwana Layer-First, jest czytelna dla małych projektów. Przy większej aplikacji zalecana jest architektura Feature-First: każda funkcjonalność ma własny, samowystarczalny katalog.

Layer-First (z Lab 1)	Feature-First (rekomendowany)
<code>components/</code> , <code>hooks/</code> , <code>services/</code> - grupowanie po typie technicznym	<code>features/weather/</code> , <code>features/favorites/</code> - grupowanie po funkcji biznesowej
Zmiana jednej funkcji dotyka wielu katalogów jednocześnie	Zmiana jednej funkcji ograniczona do jednego katalogu feature/
Latwo dodawać pliki na początku projektu	Latwo skalować: nowy feature = nowy katalog, brak konfliktu
Plik <code>weatherCard.tsx</code> 'sasiaduje' z <code>profileCard.tsx</code> mimo że to inne features	Wszystko związane z pogodą jest w <code>features/weather/</code> - nawigacja po kodzie jest intuicyjna

9.1. Struktura Feature-First dla WeatherApp v2

Architektura Feature-First porządkuje projekt według funkcjonalności biznesowych, a nie według typu technicznego plików. Oznacza to, że komponenty, hooki, store'y i ekrany związane np. z pogodą lub ulubionymi są grupowane razem w osobnych modułach. Takie podejście zwiększa skalowalność projektu, ułatwia pracę zespołową i poprawia lokalność zmian w kodzie.

```
WeatherApp/
├── app/                ← Expo Router: tylko routing (najcieńsza warstwa)
│   ├── _layout.tsx    ← QueryClient + GestureHandler + Providers
│   ├── (tabs)/
│   │   ├── _layout.tsx ← Konfiguracja zakładek
│   │   ├── index.tsx   ← import { HomeScreen } from '@features/home'
│   │   ├── favorites.tsx ← import { FavoritesScreen } from '@features/favorites'
│   │   └── detail/[city].tsx ← import { DetailScreen } from '@features/weather'
├── features/
│   ├── weather/       ← Feature: pogoda i prognoza
│   │   ├── api/
│   │   │   └── weatherApi.ts ← Tylko wywołania HTTP (fetchWeather, fetchForecast)
│   │   ├── hooks/
│   │   │   ├── useWeatherQuery.ts ← TanStack Query: aktualna pogoda
│   │   │   └── useForecastQuery.ts ← TanStack Query: prognoza 7 dni
│   │   └── components/
│   │       └── WeatherCard.tsx
```



10. Dostępność i internacjonalizacja

Dostępność i internacjonalizacja są ważnymi aspektami jakości aplikacji mobilnej, ponieważ wpływają zarówno na inkluzywność, jak i możliwość użycia produktu w różnych środowiskach językowych. Dostępność obejmuje m.in. poprawne opisywanie elementów dla czytników ekranu, natomiast internacjonalizacja dotyczy przygotowania tekstów, formatów i ustawień regionalnych do działania w wielu językach. Uwzględnienie tych zagadnień na etapie projektowania znacząco podnosi dojrzałość techniczną aplikacji.

10.1. Dostępność przez `accessibilityLabel` i rola

React Native generuje drzewo dostępności dla TalkBack (Android) i VoiceOver (iOS). Dla standardowych komponentów (Text, TextInput, Button) drzewo tworzone jest automatycznie. Dla niestandardowych elementów klikalnych i obrazów musisz opisać je ręcznie.

// Karta pogody z pełną dostępnością

```

<Pressable
  onPress={onPress}
  accessible={true}          // Element widoczny dla czytnika (domyślnie true)
  accessibilityRole='button' // Czytnik ogłasza 'przycisk' i można aktywować
  accessibilityLabel={`Pogoda w ${weather.city}: ${weather.temperature} stopni, wilgotność ${weather.humidity}
procent`}
  // Label czyta cały opis zamiast poszczególnych dzieci
  accessibilityHint='Aktywuj aby zobaczyć szczegóły prognozy'
  // Hint = co się stanie po aktywacji
>
  {/* Elementy dekoracyjne - ukryj przed czytnikiem */}
  <Image
    source={weatherIcon}
    accessibilityElementsHidden={true} // Android: ukryj cały element
    importantForAccessibility='no'     // iOS: ignoruj ten element
    style={styles.icon}
  />
  {/* Tekst - czytnik przeczyta automatycznie gdy nie ma label na rodzicu */}
  <Text style={styles.city}>{weather.city}</Text>
  <Text style={styles.temp}>{weather.temperature}°C</Text>
</Pressable>

```

10.2. Internacjonalizacja z i18next i expo-localization

Połączenie i18next i expo-localization umożliwia budowę wielojęzycznego interfejsu w aplikacjach React Native. expo-localization pozwala odczytać język systemowy urządzenia, a i18next zarządza słownikami tłumaczeń, interpolacją wartości i udostępnianiem przetłumaczonych treści komponentom. Jest to standardowe i elastyczne rozwiązanie dla aplikacji, które mają działać w więcej niż jednym języku.

Instalacja

```

npm install i18next react-i18next
npm install expo-localization # Odczyt języka systemu

```

```

// i18n/config.ts - konfiguracja
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import * as Localization from 'expo-localization';

const resources = {
  pl: { translation: {
    appTitle: 'Aplikacja Pogodowa',
    searchPlaceholder: 'Wpisz nazwę miasta...',
    humidity: 'Wilgotność: {{value}}%', // {{value}} = interpolacja
    lastUpdated: 'Zaktualizowano: {{date, datetimestring}}', // formatowanie daty
  }},
  en: { translation: {
    appTitle: 'Weather App',
    searchPlaceholder: 'Enter city name...',
    humidity: 'Humidity: {{value}}%',
    lastUpdated: 'Updated: {{date, datetimestring}}',
  }},
};

i18n.use(initReactI18next).init({
  resources,

```

```

    lng: Localization.getLocales()[0].languageCode ?? 'pl',
    fallbackLng: 'pl',
    interpolation: { escapeValue: false },
  });

// Użycie w komponencie:
import { useTranslation } from 'react-i18next';

const WeatherCard = ({ weather }: WeatherCardProps) => {
  const { t } = useTranslation();

  return (
    <View>
      <Text>{t('humidity', { value: weather.humidity })}</Text>
      <Text>{t('lastUpdated', { date: weather.updatedAt })}</Text>
    </View>
  );
};

```

11. React Native DevTools - diagnostyka i profilowanie

React Native DevTools oraz narzędzia pokrewne wspierają analizę działania aplikacji na poziomie kodu, stanu i komunikacji sieciowej. Pozwalają śledzić drzewo komponentów, breakpointy, zapytania HTTP, historię zmian stanu oraz zachowanie cache. Dzięki temu programista może nie tylko usuwać błędy, ale również świadomie oceniać architekturę i zachowanie aplikacji w czasie wykonania.

Narzędzie	Jak użyć	Do czego służy
React Native DevTools	Shake emulatora → Open DevTools lub j w terminalu Expo	Debugowanie JS: breakpointy, console.log, profiler wydajności, inspekcja sieci.
React Developer Tools	npm install -g react-devtools, potem react-devtools w terminalu	Drzewo komponentów na żywo. Inspekcja props i stanu. Zmiana props na żywo.
Flipper	Pobierz z fbflipper.com, uruchom i połącz z emulatorem	Inspekcja sieci (wszystkie HTTP), baza danych, crash logi, pluginy społeczności.
Zustand DevTools	Dodatek Chrome/Firefox 'Redux DevTools' + simple-zustand-devtools	Podgląd i time-travel historii stanu Zustand store.
TanStack Query DevTools	npm install @tanstack/react-query-devtools (web only)	Stan cache: kto jest świeży, kto stale, kiedy ostatni fetch.
Expo Dev Menu	Shake emulatora lub Ctrl+M (Android) / Cmd+D (iOS sim)	Fast Refresh, Perf Monitor, Element Inspector (zaznacz widget i zobacz jego styl).

11.1. Profilowanie wydajności

Profilowanie wydajności polega na obserwacji zużycia pamięci, czasu pracy wątku JavaScript, czasu renderowania UI oraz liczby klatek na sekundę. Pozwala wykrywać nadmiarowe renderowania, zbyt ciężkie komponenty oraz nieefektywne operacje wpływające na płynność interfejsu. W React Native jest to szczególnie ważne, ponieważ nawet drobne błędy projektowe mogą szybko przełożyć się na odczuwalne spadki responsywności aplikacji.

Performance Monitor (Expo Dev Menu > Perf Monitor)

RAM: zużycie pamięci w MB

JS: czas wykonania wątku JS w ms na klatkę (cel: < 16ms dla 60fps)

UI: czas renderowania wątku UI (cel: < 16ms)

FPS: aktualna częstotliwość klatek

Why-did-you-render - wykrywanie zbędnych renderowań:

npm install --save-dev @welldone-software/why-did-you-render

// App.tsx lub index.ts (przed importem React):

```
if (__DEV__) {  
  const whyDidYouRender = require('@welldone-software/why-did-you-render');  
  whyDidYouRender(React, {  
    trackAllPureComponents: true,  
    logOnDifferentValues: true, // Loguj gdy props różni się referencją ale nie wartością  
  });  
}
```

// Na wybranym komponencie:

WeatherCard.whyDidYouRender = true;

// Konsola pokaże: 'WeatherCard re-rendered because of props change: weather (same value!)'

12. Zadania do wykonania

Zadania są kontynuacją projektu WeatherApp z React Native Lab 1. Właśnie ten kod z Lab 1 będzie naszym punktem startowym. Migrujemy go do nowej architektury i rozbudowujemy o nowe funkcje.

Zadanie 1 (20 pkt) - Migracja na Zustand

1. Dodaj pakiety: zustand, react-native-mmkv (lub @react-native-async-storage/async-storage jeśli używasz Expo Go), @tanstack/react-query.
2. Utwórz stores/favoritesStore.ts z Zustand + persist. Pole favorites: string[]. Akcje: addCity(), removeCity(), clear(). Persistencja przez AsyncStorage lub MMKV.
3. Usun FavoritesContext.tsx i FavoritesProvider. Zastąp wszystkie użycia useContext(FavoritesContext) wywołaniami useFavoritesStore(sel).
4. Skonfiguruj QueryClient w app/_layout.tsx. Utwórz hooks/useWeatherQuery.ts z useQuery. Usun ręczne useEffect + useState do pobierania pogody tam gdzie to możliwe.
5. Przetestuj że ulubione są zachowywane po zamknięciu i reopenowaniu aplikacji.

WERYFIKACJA: Brak importów z FavoritesContext. Ulubione przeżyją restart aplikacji. useWeatherQuery zwraca isLoading, isError, data bez ręcznie pisanego stanu.

Zadanie 2 (25 pkt) - Offline i zaawansowany store

6. Rozszerz weatherStore.ts o mapę cache: weatherMap: Record<string, WeatherData & { cachedAt: Date }>. Zaimplementuj logikę 'cache-first': pokazuj stare dane natychmiast, pobieraj świeże w tle jeśli starsze niż 15 minut.
7. Dodaj wskaźnik świeżości danych (badge) w WeatherCard: 'Dane sprzed X minut'. Oblicz różnicę między aktualnym czasem a cachedAt.
8. Zaimplementuj Zustand store dla ustawień (settingsStore.ts): units ('celsius' | 'fahrenheit'), language ('pl' | 'en'). Przeliczaj temperaturę w zależności od jednostek.
9. Przetestuj offline-first: załaduj pogodę, wyłącz internet (tryb samolotowy), uruchom ponownie - stare dane powinny się wyświetlić z odpowiednim komunikatem.

WERYFIKACJA: Dane widoczne przy wyłączonym internecie. Badge z wiekiem danych. Przełącznik Celsjusz/Fahrenheit działa.

Zadanie 3 (30 pkt) - Animacje

10. Animowane wejście kart na HomeScreen: każda karta wchodzi z przesunięcia 80px z prawej z opóźnieniem $\text{index} * 70\text{ms}$. Użyj `Animated.parallel (translateX + opacity)`. Czas: 350ms, `Easing.out(Easing.back(1.5))`.
11. WeatherCard z efektem naciśnięcia: `scale 1.0 → 0.95` przy `onPressIn` (`Animated.spring, bounciness: 0`), powrót `0.95 → 1.0` przy `onPressOut` (`bounciness: 4`).
12. Zainstaluj `react-native-reanimated` i `react-native-gesture-handler`. Zaimplementuj `SwipeableCard` na ekranie Ulubionych: przeciągnięcie $\geq 100\text{px}$ w lewo usuwa miasto z listy (`Reanimated + Gesture.Pan()`).
13. Opcjonalnie: animacja rozwijania szczegółów w WeatherCard - po tapnięciu karta rozszerza się pionowo ujawniając wilgotność i wiatr (`Animated.timing` na `height` lub `withTiming` na `SharedValue`).

WERYFIKACJA: Demo na AVD przy 60fps. Staggered entry lista. Scale przy naciśnięciu. Swipe-to-delete na Ulubionych. Reanimated skonfigurowany (`babel.config.js`).

Zadanie 4 (25 pkt) - Testy

14. Napisz min. 4 testy jednostkowe dla `favoritesStore.ts`: `addCity` (normalny), `addCity` (duplikat), `removeCity`, `clear()`. Użyj `renderHook` i `setState` do resetowania stanu między testami.
15. Napisz min. 3 testy komponentu WeatherCard: wyświetlanie miasta, temperatury, wywołanie `onPress`. Użyj `mockWeather` fixture.
16. Napisz test integracyjny HomeScreen: wpisanie miasta w `SearchBar` → `submit` → załadowanie danych (`mock fetchWeather`) → wyświetlenie WeatherCard. Użyj `jest.mock('./services/weatherService')` i `waitFor`.
17. Uruchom `npm test --coverage`. Osiągnij pokrycie $> 60\%$ dla pliku `favoritesStore.ts` i `WeatherCard.tsx`. Dołącz screenshot raportu do sprawozdania.

WERYFIKACJA: `npm test` uruchamia się bez błędów. Min. 8 testów (4 jednostkowe + 3 komponentów + 1 integracyjny). Raport coverage $> 60\%$.

13. Kryteria oceniania

Zadanie	Punkty	Co weryfikuje prowadzący
Zad. 1: Zustand	20 pkt	Brak Context API. <code>persist</code> działa. <code>QueryClient</code> skonfigurowany. <code>useWeatherQuery</code> użyty zamiast <code>useEffect+useState</code> .
Zad. 2: Offline	25 pkt	Dane widoczne przy wyłączonym internecie. Badge czasu. <code>settingsStore</code> z przelicznikiem jednostek.
Zad. 3: Animacje	30 pkt	Staggered entry lista. Scale przy naciśnięciu. Swipe-to-delete z <code>Reanimated</code> . Babel plugin skonfigurowany.
Zad. 4: Testy	25 pkt	Min. 8 testów. Poprawne mockowanie. <code>waitFor</code> dla <code>async</code> . Coverage $> 60\%$.
RAZEM	100 pkt	

Ocena	Punkty / wymagania
5.0	90-100 pkt - Wszystkie zadania. <code>Reanimated + gesture</code> . Coverage $> 70\%$. <code>i18next</code> lub dostępność (<code>accessibilityLabel</code>).
4.5	80-89 pkt - Zadania 1-3 kompletne. Min. 6 testów. Swipe-to-delete działa.

4.0	70-79 pkt - Zadania 1-2 + staggered entry (Animated API). Min. 4 testy.
3.5	60-69 pkt - Zadania 1-2. Zustand + persist działa. Brak animacji.
3.0	50-59 pkt - Zadanie 1: Zustand zamiast Context. Ulubione persystują.
2.0	0-49 pkt - Projekt nie kompiluje lub Zustand nie działa.

14. Najczęstsze błędy i ich rozwiązania

W drugiej części materiału najczęstsze błędy dotyczą przede wszystkim niewłaściwego użycia Zustand, TanStack Query, Reanimated, środowiska testowego oraz persystencji stanu. Typowe problemy obejmują m.in. nieoptymalne subskrypcje store'a, błędną konfigurację pluginu Reanimated, źle zdefiniowane klucze queryKey, konflikty gestów, problemy z testami oraz trudności wynikające z niezgodności środowiska Expo Go z niektórymi bibliotekami natywnymi. Analiza takich przypadków rozwija umiejętność diagnozowania problemów w bardziej zaawansowanych projektach mobilnych.

Błąd / objaw	Przyczyna i rozwiązanie
Zustand: Store rerenderuje cały komponent mimo selektora	Destrukturyzujesz zamiast używać selektora: <code>const { favorites } = useStore()</code> = anti-pattern! Użyj: <code>const favorites = useStore(s => s.favorites)</code> . Jeśli potrzebujesz wielu pól, użyj <code>useShallow</code> .
Reanimated: 'Tried to synchronously call a non-worklet function'	Wywołałeś funkcje JS (<code>setState</code> , nawigacja) bezpośrednio w worklet. Owin w <code>runOnJS()</code> : zamiast <code>navigation.push()</code> napisz <code>runOnJS(navigation.push)('/detail')</code> .
Reanimated: Animacja nie działa po konfiguracji Babel	'react-native-reanimated/plugin' nie jest ostatnim pluginem w <code>babel.config.js</code> . Upewnij się że jest na koncu listy plugins. Wyczyść cache: <code>npx expo start --clear</code> .
TanStack Query: stale data po zmianie parametru	<code>queryKey</code> nie zawiera wszystkich parametrów wpływających na wynik. Zmień: <code>queryKey: ['weather']</code> na <code>queryKey: ['weather', city]</code> . Zmiana <code>city</code> automatycznie zainwaliduje cache.
MMKV: 'MMKV is not available in this environment'	MMKV wymaga native build i nie działa w Expo Go. Użyj <code>npx expo prebuild</code> i uruchom bezpośrednio przez <code>npx expo run:android</code> . Lub zamień na <code>AsyncStorage</code> dla testów w Expo Go.
Jest: 'Cannot find module react-native-reanimated'	Reanimated wymaga konfiguracji w <code>jest transformIgnorePatterns</code> . Dodaj 'react-native-reanimated' do listy: <code>transformIgnorePatterns: ['node_modules/(?!(... react-native-reanimated)/)']</code> .
RNLT: 'act() warning: Wrap in act(...)'	Zmiana stanu poza <code>act()</code> . Owiń każdą akcję zmieniającą stan w <code>act()</code> : <code>act(() => { result.current.addCity('Gdansk'); })</code> . Dla <code>async</code> : <code>await act(async () => { await loadData(); })</code> .
Zustand persist: Stare dane po zmianie struktury store	Zmieniłeś kształt stanu ale stary JSON w <code>AsyncStorage</code> nadal istnieje. Dodaj wersjonowanie: <code>version: 1</code> w opcjach <code>persist</code> i funkcję <code>migrate()</code> , lub zmień „name” klucza.
useAnimatedStyle: 'Reading from value during render'	Używasz <code>SharedValue.value</code> wewnątrz body komponentu (poza <code>useAnimatedStyle</code>). <code>SharedValues</code> można czytać TYLKO w workletach (<code>useAnimatedStyle</code> , <code>useDerivedValue</code>) lub przez <code>.value</code> w <code>handleEvent</code> ach.
GestureDetector: Gest nie reaguje na ScrollView	Konflikty gestów: <code>ScrollView</code> i <code>Pan</code> gest walczą o to samo dotknięcie. Użyj <code>activeOffsetX</code> w <code>Pan</code> gesture aby rozróżniać od pionowego scrolla. Lub owiń <code>ScrollView</code> w <code>NativeViewGestureHandler</code> .

14.1. Narzędzia diagnostyczne

Narzędzia diagnostyczne w rozszerzonym projekcie React Native obejmują zarówno klasyczne mechanizmy debugowania, jak i narzędzia specyficzne dla bibliotek stanu, animacji i testów. Oprócz logów, czyszczenia cache i devtoolsów, istotne staje się także sprawdzanie konfiguracji Babel, pełny reset środowiska, analiza persystowanego stanu czy profilowanie zbędnych renderów. Jest to naturalny etap przejścia od prostych ćwiczeń do bardziej profesjonalnej pracy nad aplikacją.

```
# Weryfikacja konfiguracji Babel (czy Reanimated jest ostatni):
cat babel.config.js

# Reset pełny (gdy cos dziwnego dzieje sie z pakietami):
rm -rf node_modules .expo
npm install
npx expo start --clear

# Sprawdź czy MMKV jest dostępne (wymaga native build):
npx expo run:android # Zamiast npx expo start

# Debugowanie Zustand persist - sprawdź co jest w AsyncStorage:
import AsyncStorage from '@react-native-async-storage/async-storage';
AsyncStorage.getAllKeys().then(console.log);
AsyncStorage.getItem('favorites-storage').then(console.log);

# Profil rerenderów (why-did-you-render musi być skonfigurowany):
WeatherCard.whyDidYouRender = true;

# Testy z verbose output:
npm test -- --verbose --watchAll=false
```

Instrukcja React Native 2 - Zustand, Reanimated, AsyncStorage i Testy

Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki

MobileHub

Następne ćwiczenie: React Native 2 - Publikacja w App Store i Play Store, EAS Build, CI/CD

ĆWICZENIE LABORATORYJNE REACT NATIVE 3

EAS Build · Publikacja · CI/CD z GitHub Actions

Programowanie Aplikacji Mobilnych · moduł RN

Projekt: WeatherApp v3

Wymagania wstępne: ukończony React Native Lab 2 (Zustand, Reanimated, testy)

Spis treści

1. Czym jest EAS i dlaczego zastępuje React Native CLI?	1
2. Konfiguracja EAS Build - środowiska i profile	2
3. Warianty aplikacji - osobne środowiska z app.config.ts	4
4. Podpisywanie aplikacji - klucze Android i certyfikaty iOS	6
5. Google Play Store - od konta do publikacji	7
6. Apple App Store - od Apple Developer Program do TestFlight	8
7. CI/CD z GitHub Actions i EAS	10
8. Zarządzanie sekretami - EAS Secrets i zmienne środowiskowe	12
9. Monitorowanie - Sentry i Firebase Crashlytics	13

1. Czym jest EAS i dlaczego zastępuje React Native CLI?

Expo Application Services (EAS) to zestaw usług chmurowych firmy Expo do budowania, dystrybucji i aktualizacji aplikacji React Native. Zastępuje lokalne budowanie przez React Native CLI i rozwiązuje trzy główne problemy dewelopera mobilnego: potrzebę posiadania Maca do budowania iOS, złożoną konfigurację kluczy i certyfikatów oraz ręczną publikację do sklepów.

Analogia: warsztat vs. outsourcing produkcji

Budowanie przez RN CLI to warsztat rzemieślniczy: wszystko robisz sam, potrzebujesz swoich narzędzi (macOS dla iOS, Android Studio), sam zarządzasz kluczami. EAS to fabryka z outsourcingiem: wysyłasz projekt (kod), fabryka buduje na swoich maszynach i oddaje gotowy produkt (IPA, AAB). Klucze trzymane są bezpiecznie w chmurze, brak uzależnienia od konkretnego komputera.

1.1. Trzy usługi EAS

Usługa	Do czego służy	Odpowiednik Flutter
EAS Build	Budowanie natywnych binariów (APK, AAB, IPA) w chmurze. Zarządzanie certyfikatami i kluczami. Profile: development, preview, production.	flutter build apk / flutter build app-bundle / flutter build ipa
EAS Submit	Automatyczne przesłanie gotowego buildu do Google Play lub App Store Connect. Odpowiednik ręcznego uploadu przez przeglądarkę.	Fastlane supply / r0adkill/upload-google-play
EAS Update	Over-the-Air (OTA) aktualizacje kodu JavaScript bez przechodzenia przez recenzję sklepu. Zmiany UI/logiki - bez nowego buildu.	CodePush (Microsoft) - brak bezpośredniego odpowiednika we Flutterze

1.2. Plany EAS - co jest dostępne za darmo?

Plan	Limity	Kiedy wystarczy
Free	30 buildów miesięcznie, kolejka wspólna, max 15 min na build.	Ćwiczenia laboratoryjne, projekty studenckie, MVPs. Absolutnie wystarczający na potrzeby tego kursu.
Production	100+ buildów, dedykowane workery, priorytetowa kolejka.	Zespoły z kilkoma developerami, aplikacje komercyjne z częstymi wydaniem.
Enterprise	Nieograniczone buildy, self-hosted runners, SLA.	Duże firmy z wymaganiami bezpieczeństwa danych.

Rejestracja konta Expo

zarejestruj się na expo.dev (darmowe). Konto jest wymagane do korzystania z EAS Build. Połącz projekt przez eas login i eas build:configure.

2. Konfiguracja EAS Build - środowiska i profile

EAS Build konfigurowany jest plikiem eas.json w katalogu głównym projektu. Plik definiuje profile budowania - zestawy ustawień dla różnych środowisk. Każdy profil określa: tryb budowania (debug/release), platformę docelową, zmienne środowiskowe i sposób zarządzania certyfikatami.

2.1. Instalacja i inicjalizacja EAS CLI

```
# Instalacja EAS CLI globalnie:
npm install -g eas-cli

# Logowanie na konto Expo:
eas login
# Podaj e-mail i hasło konta expo.dev
```

```
# Inicjalizacja EAS w projekcie (tworzy eas.json i aktualizuje app.json):
eas build:configure
# EAS zapyta o platformy: Android / iOS / All

# Weryfikacja konfiguracji:
eas build:list # Historia buildow (pusta na poczatku)
eas whoami    # Sprawdz zalogowanego uzytkownika
```

2.2. Plik eas.json - profile budowania

```
// eas.json - glowny plik konfiguracyjny EAS Build
{
  "cli": {
    "version": ">= 12.0.0" // Minimalna wersja EAS CLI
  },
  "build": {

    // PROFIL: development - instalowalne na emulatorze/urzedzeniu, z dev menu
    "development": {
      "developmentClient": true, // Wbuduje Expo Dev Client (nie Expo Go!)
      "distribution": "internal", // Dystrybucja wewnetrzna (nie sklep)
      "android": {
        "buildType": "apk", // APK zamiast AAB - latwiejsza instalacja
        "gradleCommand": ":app:assembleDebug"
      },
      "env": { // Zmienne srodowiskowe wbudowane w build
        "APP_VARIANT": "development",
        "API_URL": "https://api-dev.open-meteo.com"
      }
    },

    // PROFIL: preview - release build do testow wewnetrznych (bez sklepu)
    "preview": {
      "distribution": "internal",
      "android": {
        "buildType": "apk" // APK do bezposredniej instalacji (QR kod)
      },
      "env": {
        "APP_VARIANT": "preview",
        "API_URL": "https://api-staging.open-meteo.com"
      }
    },

    // PROFIL: production - release build do sklepow
    "production": {
      "distribution": "store", // Dystrybucja przez sklep
      "android": {
        "buildType": "app-bundle" // AAB wymagane przez Google Play
      },
      "autoIncrement": true, // Automatyczne zwiekszanie versionCode/buildNumber
      "env": {
        "APP_VARIANT": "production",
        "API_URL": "https://api.open-meteo.com"
      }
    }
  },

  "submit": {
    "production": {
      "android": {
        "serviceAccountKeyPath": "./service-account.json", // Klucz Google Play API
```

```
    "track": "internal"    // internal / alpha / beta / production
  }
}
```

2.3. Uruchamianie buildow

```
# Uruchomienie buildu w chmurze EAS:
eas build --platform android --profile development
eas build --platform android --profile preview
eas build --platform android --profile production
eas build --platform all      --profile production # Android + iOS jednocześnie

# EAS wyświetla URL do śledzenia postępów buildu w przeglądarce.
# Gotowy build można pobrać lub wysłać bezpośrednio do sklepu.

# Lokalny build (bez chmury) - wymaga Android Studio / Xcode:
eas build --platform android --profile development --local

# Sprawdź status i pobierz ostatni build:
eas build:list --platform android --limit 5
eas build:download --id <BUILD_ID>

# Wyślij gotowy build do sklepu (po eas build --profile production):
eas submit --platform android --profile production
```

3. Warianty aplikacji - osobne środowiska z app.config.ts

Odpowiednikami Flavor-ów z Fluttera i Androida są w React Native / Expo warianty konfiguracji aplikacji. Zamiast edytować app.json (statyczny), używamy dynamicznego app.config.ts, który może czytać zmienne środowiskowe i generować różną konfigurację dla każdego środowiska.

Analogia: instrukcja obsługi vs. system if-else

app.json to statyczna instrukcja obsługi - zawsze ta sama, niezależnie od sytuacji. app.config.ts to inteligentny asystent, który pyta 'dla którego środowiska mam skonfigurować aplikację?' i dobiera odpowiednie ustawienia: inna nazwa, ikona z badgem DEV, inny bundle identifier.

3.1. Dynamiczna konfiguracja z app.config.ts

```
// app.config.ts - zamień app.json na ten plik
import { ExpoConfig, ConfigContext } from 'expo/config';

// Odczyt zmiennej środowiskowej ustawianej przez EAS (pole env w eas.json)
const APP_VARIANT = process.env.APP_VARIANT ?? 'development';

// Mapowanie wariantu na konfigurację
const variants = {
  development: {
    name: '[DEV] WeatherApp', // Nazwa pod ikoną
    bundleId: 'pl.edu.pam.weather.dev', // iOS bundle ID
    packageName: 'pl.edu.pam.weather.dev', // Android package
  },
};
```

```

    icon: './assets/icon-dev.png', // Ikona z czerwonym badgem
  },
  preview: {
    name: '[STG] WeatherApp',
    bundleId: 'pl.edu.pam.weather.stg',
    packageName: 'pl.edu.pam.weather.stg',
    icon: './assets/icon-stg.png',
  },
  production: {
    name: 'WeatherApp',
    bundleId: 'pl.edu.pam.weather',
    packageName: 'pl.edu.pam.weather',
    icon: './assets/icon.png', // Ikona produkcyjna bez badgu
  },
} as const;

const variant = variants[APP_VARIANT as keyof typeof variants] ?? variants.development;

export default ({ config }: ConfigContext): ExpoConfig => ({
  ...config, // Wspolna konfiguracja bazowa
  name: variant.name,
  slug: 'weather-app', // Slug ZAWSZE taki sam (identyfikator Expo)
  version: '1.0.0',
  orientation: 'portrait',
  icon: variant.icon,
  extra: {
    apiUrl: process.env.API_URL, // Dostepne w aplikacji przez Constants.expoConfig.extra
    appVariant: APP_VARIANT,
  },
  android: {
    package: variant.packageName,
    adaptiveIcon: {
      foregroundImage: variant.icon,
      backgroundColor: '#0164FF',
    },
    versionCode: 1, // Nadpisywane przez autoIncrement w eas.json
  },
  ios: {
    bundleIdentifier: variant.bundleId,
    buildNumber: '1',
  },
});

```

3.2. Odczyt konfiguracji w kodzie aplikacji

```

// constants/config.ts - centralne miejsce konfiguracji
import Constants from 'expo-constants';

// Odczyt wartosci z extra (skonfigurowane w app.config.ts)
const extra = Constants.expoConfig?.extra ?? {};

export const Config = {
  apiUrl: extra.apiUrl as string ?? 'https://api.open-meteo.com',
  appVariant: extra.appVariant as string ?? 'development',

  // Gettery pochodne - czytelniejsze niz porownania stringow w kodzie
  get isDev() { return this.appVariant === 'development'; },
  get isPreview() { return this.appVariant === 'preview'; },
  get isProd() { return this.appVariant === 'production'; },
};

```

```
// Uzycie w komponencie:
import { Config } from '@constants/config';

// Badge srodowiska widoczny tylko w dev i preview:
{!Config.isProd && (
  <View style={styles.envBadge}>
    <Text style={styles.envText}>{Config.appVariant.toUpperCase()}</Text>
  </View>
)}

// Serwis API uzywajacy prawidlowego URL:
const response = await fetch(`${Config.apiUrl}/v1/forecast?...`);
```

3.3. Różne ikony dla wariantów

Podstawowym sygnałem wizualnym dla testera, że używa wersji deweloperskiej, jest ikona z kolorowym badgem. Nie pomył aplikacji produkcyjnej z testową - to częsta przyczyna incydentów.

Wariant	Ikona	Nazwa pod ikoną	Package / Bundle ID
development	Czerwony badge DEV	[DEV] WeatherApp	pl.edu.pam.weather.dev
preview	Pomarańczowy badge STG	[STG] WeatherApp	pl.edu.pam.weather.stg
production	Bez badgu	WeatherApp	pl.edu.pam.weather

Korzyści

Trzy różne package names oznaczają, że wszystkie trzy warianty mogą być zainstalowane jednocześnie na tym samym urządzeniu. Tester ma wygodę porównywania zachowania między środowiskami bez reinstalowania.

4. Podpisywanie aplikacji - klucze Android i certyfikaty iOS

Każda aplikacja opublikowana w sklepie musi być cyfrowo podpisana. Podpis gwarantuje użytkownikowi, że aktualizacja pochodzi od tego samego dewelopera co pierwotna instalacja. EAS Build może zarządzać kluczami automatycznie, co eliminuje najczęstszy problem początkujących: utratę klucza i brak możliwości aktualizacji aplikacji.

4.1. Podpisywanie Android - keystore

Google Play App Signing - dwa klucze

Klucz przesłania (upload key) jest w Twoim posiadaniu i służy do podpisania paczki przed wysłaniem do Play Console. Klucz aplikacji (app signing key) jest przechowywany przez Google i służy do podpisania APK dostarczonego użytkownikowi. Jeśli zgubisz upload key, możesz go zresetować. Jeśli Google zgubi app signing key, nikt go nie zgubi, bo Google go chroni.

```
# Podejscie 1: EAS zaradza kluczami automatycznie (REKOMENDOWANE dla poczatkujacych)
# Przy pierwszym eas build EAS wygeneruje keystore i zapisze go bezpiecznie w chmurze.
# Nie musisz recznie tworzyć keystore ani pamiętać hasła.

# Podejscie 2: Własny keystore (gdzie chcesz pełną kontrolę lub migrujesz istniejącą aplikację)

# Krok 1: Wygeneruj keystore lokalnie
keytool -genkey -v \
  -keystore weather-upload.jks \
  -keyalg RSA -keysize 2048 \
  -validity 10000 \
  -alias upload

# Krok 2: Załaduj keystore do EAS Credentials
eas credentials --platform android
# Wybierz: 'Set up a new keystore' > 'Use existing keystore'
# Podaj ścieżkę do .jks i hasła

# Sprawdź zapisane poświadczenia:
eas credentials --platform android
```

4.2. Podpisywanie iOS - certyfikaty i profile

Ekosystem Apple wymaga trzech elementów do podpisania aplikacji iOS: konta Apple Developer Program (99 USD rocznie), certyfikatu dystrybucyjnego oraz profilu apro wizacyjnego. EAS może wygenerować i zarządzać wszystkim automatycznie.

Element	Co to jest	EAS może to zarządzać?
Apple Developer Account	Konto w programie Apple Developer Program. Koszt: 99 USD rocznie. Wymagane do publikacji w App Store.	Nie - musisz mieć konto. EAS loguje się na Twoje konto podczas budowania.
Distribution Certificate	Klucz kryptograficzny identyfikujący Cię jako dewelopera u Apple. Ważny 1 rok.	Tak - EAS tworzy i odawia automatycznie jeśli wygaśnie.
Provisioning Profile	Plik łączący aplikację (bundle ID) z certyfikatem i listą urządzeń lub sklepem.	Tak - EAS generuje przy każdym nowym bundle ID.
App Store Connect API Key	Klucz API do automatycznego wysłania do App Store Connect bez interaktywnego logowania.	Musisz wygenerować w App Store Connect. EAS zapisuje go bezpiecznie.

```
# Automatyczne zarządzanie certyfikatami iOS przez EAS:
eas credentials --platform ios
# EAS zapyta o Apple ID i wygeneruje certyfikat oraz profil.
# Przy kolejnych buildach używa zapisanych poświadczeń.

# Konfiguracja App Store Connect API Key (do EAS Submit):
# 1. Przejdź do: App Store Connect > Użytkowników i dostęp > Klucze
# 2. Wygeneruj klucz z rolą 'App Manager'
# 3. Pobierz plik .p8 (tylko raz!)
# 4. Dodaj do EAS:
```

```
eas credentials --platform ios  
# Wybierz: 'App Store Connect API Key' > 'Add new'
```

Ważne rozróżnienie

Utrata certyfikatu iOS kontra keystore Android: W Androidzie Google chroni app signing key - zgubienie upload key jest naprawialne. W iOS utrata certyfikatu dystrybucyjnego jest mniej dramatyczna: można wygenerować nowy w Apple Developer Portal. Ale utrata prywatnego klucza certyfikatu z Twojego komputera bez kopii zapasowej wymaga generacji nowego, co może przerwać istniejące profile.

5. Google Play Store - od konta do publikacji

Proces publikacji na Google Play składa się z kilku etapów. Warto je znać także bez konta dewelopera: rozumienie procesu jest niezbędne do prawidłowej konfiguracji CI/CD, nawet jeśli publikacja następuje dopiero po zakończeniu kursu.

5.1. Cykl życia wersji

Ścieżka	Czas recenzji	Dostępność	Przeznaczenie
Internal Testing	Natychmiastowa	Max 100 adresów e-mail na liście	Testy CI/CD, szybka weryfikacja buildu przez zespół. Brak recenzji Google.
Closed Testing (alfa)	Kilka minut	Grupy testerów przez link	Testy z większym gronem: QA, beta-testerzy, klient.
Open Testing (beta)	Do 3 dni (pierwsza wersja)	Publiczna, widoczna w Play Store	Publiczne beta, zbieranie recenzji przed pełną publikacją.
Production	1-3 dni	Wszyscy użytkownicy	Staged rollout (10%, 50%, 100%). Możliwość cofnięcia.

5.2. Wymagane materiały przed pierwszą publikacją

1. Konto dewelopera Google Play - jednorazowa opłata 25 USD na koncie Google.
2. Opis aplikacji: nazwa (max 50 zn.), krótki opis (max 80 zn.), pełny opis (max 4000 zn.).
3. Grafiki: ikona 512x512 px (PNG), Feature Graphic 1024x500 px (JPG/PNG).
4. Zrzuty ekranu: min. 2, maks. 8. Wymagany format telefonowy (min. 320 dp po dłuższym boku).
5. Polityka prywatności: URL do strony. Wymagana, gdy aplikacja zbiera jakiegokolwiek dane.
6. Data Safety Form: deklaracja jakie dane zbierasz, jak je udostępniasz i chronisz.
7. Klasyfikacja treści: kwestionariusz IARC określający grupę wiekową.

5.3. Automatyzacja versionCode

Każdy upload do Play Store musi mieć wyższy versionCode niż poprzedni. Pole autoIncrement w eas.json i numer buildu GitHub automatyzują ten process.

```
// app.config.ts - dynamiczny versionCode z CI
const buildNumber = parseInt(process.env.EAS_BUILD_BUILD_PROFILE_NUMBER ?? '1');
// EAS_BUILD_BUILD_PROFILE_NUMBER = automatycznie ustawiana przez EAS

export default ({ config }: ConfigContext): ExpoConfig => ({
  ...config,
  android: {
    package: variant.packageName,
    versionCode: buildNumber, // Unikalny dla każdego buildu
  },
  ios: {
    buildNumber: String(buildNumber),
  },
});

// Lub przez autoIncrement w eas.json (prostsze, EAS zarządza licznikiem):
// "production": { "autoIncrement": true }
// EAS automatycznie pobiera ostatni versionCode z Play Store i zwiększy o 1.
```

6. Apple App Store - od Apple Developer Program do TestFlight

Publikacja na App Store jest bardziej restrykcyjna niż na Google Play. Apple manualnie weryfikuje każdą wersję aplikacji pod kątem zgodności z wytycznymi (App Store Review Guidelines). Recenzja trwa zwykle 24-48 godzin. Do testów wewnętrznych służy TestFlight.

TestFlight - odpowiednik Internal Testing w Play Store

TestFlight to oficjalna platforma Apple do dystrybucji wersji testowych. Wewnętrzni testerzy (do 100 osób z Twojego Apple Developer Account) otrzymują dostęp natychmiast. Zewnętrzni testerzy (do 10 000 osób przez link) wymagają jednorazowej recenzji beta przez Apple (zwykle 1-2 dni).

6.1. Porównanie recenzji: App Store vs Google Play

Aspekt	Google Play	App Store
Czas recenzji (pierwsza wersja)	1-3 dni robocze	24-48 godzin (zwykle)
Czas recenzji (aktualizacja)	Kilka godzin do 1 dnia	24-48 godzin (takie samo)
Podejście do recenzji	Zautomatyzowana analiza + część ręczna. Mniej restrykcyjna.	Głównie ręczna recenzja przez pracowników Apple. Bardziej rygorystyczna.
Najczęstsze powody odrzucenia	Naruszenia polityki danych, źle sklasyfikowana treść, problemy z	Uszkodzone funkcje, niekompletne opisy, zbyt agresywne prawa

	prywatnością.	dostępu, brak polityki prywatności.
Odwołanie od decyzji	E-mail do zespołu Play. Odpowiedz po kilku dniach.	Resolution Center w App Store Connect. Możliwość rozmowy telefonicznej.

6.2. Konfiguracja EAS Submit dla iOS

```
// eas.json - konfiguracja wysłania do App Store
{
  "submit": {
    "production": {
      "ios": {
        "appleId": "developer@example.com", // Twój Apple ID
        "ascAppId": "1234567890", // App ID z App Store Connect
        "appleTeamId": "XXXXXXXXXX", // Team ID z Apple Developer
        "ascApiKeyPath": "./AuthKey_XXXXXXXXXX.p8", // Klucz API
        "ascApiKeyIssuerId": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxx"
      }
    }
  }
}

# Wysłanie buildu iOS do TestFlight (po eas build --platform ios --profile production):
eas submit --platform ios --profile production

# EAS automatycznie prześle .ipa do App Store Connect i oznaczy jako gotowy do TestFlight.
# Wewnętrzni testerzy dostaną powiadomienie przez aplikację TestFlight na iPhone.
```

7. CI/CD z GitHub Actions i EAS

Połączenie GitHub Actions z EAS CLI tworzy potężny potok automatyzacji: każdy push na główną gałąź uruchamia testy, a po ich przejściu EAS buduje aplikacje w chmurze i opcjonalnie publikuje do sklepów. Cały proces bez ingerencji ludzkiej.

Analogia: linia montażowa w fabryce

Deweloper wysyła zmiany (commit) jak inżynier z rysunkiem technicznym. GitHub Actions to inspekcja jakości: sprawdza czy rysunek jest poprawny (testy). EAS Build to fabrykę która produkuje gotowy produkt (APK/AAB). EAS Submit to kurier dostarczający do sklepu (Play Store). Każdy etap jest zautomatyzowany i powtarzalny.

7.1. Pełny workflow CI/CD

```
# .github/workflows/ci-cd.yml
name: WeatherApp CI/CD

on:
```

```

push:
  branches: [main, develop]
pull_request:
  branches: [main]
workflow_dispatch:          # Manualne uruchomienie z panelu GitHub

jobs:
  # == JOB 1: Testy i analiza statyczna ==
  test:
    name: Test & Lint
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Konfiguracja Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'          # Cache node_modules - szybsze kolejne uruchomienia

      - name: Instalacja zaleznosci
        run: npm ci            # ci zamiast install: deterministyczny, szybszy

      - name: Sprawdzenie typow TypeScript
        run: npx tsc --noEmit  # Weryfikacja typow bez generowania plikow

      - name: Analiza ESLint
        run: npm run lint

      - name: Uruchomienie testow z pokryciem
        run: npm test -- --coverage --watchAll=false --ci

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

```

```

# == JOB 2: Build Android (tylko gałąź main) ==
build-android:
  name: EAS Build Android
  runs-on: ubuntu-latest
  needs: test          # Uruchom dopiero po przejściu testow
  if: github.ref == 'refs/heads/main'
  steps:
    - uses: actions/checkout@v4

    - uses: actions/setup-node@v4
      with:
        node-version: '20'
        cache: 'npm'

    - run: npm ci

    - name: Konfiguracja EAS CLI
      uses: expo/expo-github-action@v8
      with:
        eas-version: latest
        token: ${ secrets.EXPO_TOKEN } # Token konta Expo (Settings > Access To-
kens)

    - name: EAS Build - Android production
      run: |
        eas build \

```

```

--platform android \
--profile production \
--non-interactive \    # Bez interaktywnych pytan - wymagane w CI
--no-wait              # Nie czekaj na zakonczenie (buduje asynchronicz-
nie)

# Opcjonalny krok: wyslij do Play Store po zakonczeniu buildu
# - name: EAS Submit - Play Store Internal Testing
#   run: eas submit --platform android --profile production --non-interactive

```

7.2. Koncepcja pipeline-u

Pojecie GitHub Actions	Co oznacza / Przyklad	Odpowiednik EAS
Workflow (.yaml)	Plik YAML definiujacy cały potok. Trafia do .github/workflows/.	Brak bezposredniego odpowiednika - EAS to osobna uslug.
Trigger (on:)	Zdarzenie uruchamiajace: push, pull_request, schedule, workflow_dispatch.	Webhook EAS moze triggerowac GitHub Actions po zakonczeniu buildu.
Job	Grupa krokow na jednym runnerze. Job-y moga biec rownolegle (needs: brak) lub sekwencyjnie (needs: test).	Jeden EAS Build = jeden job po stronie EAS.
Secret	Zaszyfrowana zmienna: EXPO_TOKEN, SERVICE_ACCOUNT_JSON. Settings > Secrets.	EAS Secrets (sekcja 8) - alternatywne miejsce na sekrety specyficzne dla EAS.
Artifact	Plik zapisany po zakonczeniu job-u (np. raport coverage). Pobieralny przez 90 dni.	Build EAS jest przechowywany w chmurze Expo przez 30 dni (free) lub dluzej.

8. Zarządzanie EAS Secrets i zmiennymi środowiskowymi

Klucze API, tokeny i hasła nie mogą trafić do repozytorium Git. EAS oferuje własny system przechowywania sekretów, który automatycznie wstrzykuje je do procesu budowania.

8.1. Dwa miejsca na sekrety

Miejsce	Dostępne dla	Kiedy używać
GitHub Secrets (Settings > Secrets > Actions)	GitHub Actions workflow (kroki run: i uses:)	EXPO_TOKEN, SERVICE_ACCOUNT_JSON, CODECOV_TOKEN. Wszystko co potrzebne tylko do automatyzacji CI.
EAS Secrets (expo.dev > Project > Secrets)	Proces budowania EAS (wbudowane w binary)	API_KEY, SENTRY_DSN, MAPS_API_KEY. Wartości wstrzykiwane do aplikacji jako zmienne środowiskowe w trakcie kompilacji.

8.2. Konfiguracja EAS Secrets

```
# Dodawanie sekretów przez EAS CLI:
eas secret:create --scope project --name SENTRY_DSN --value 'https://xxx@sentry.io/yyy'
eas secret:create --scope project --name MAPS_API_KEY --value 'AIzaXXX...'

# Listowanie sekretów (wartości nie są wyświetlane - tylko nazwy):
eas secret:list

# Usuwanie sekretu:
eas secret:delete --id <SECRET_ID>

# Sekrety są automatycznie dostępne jako process.env.NAZWA podczas budowania EAS.
# W app.config.ts możesz je czytać i wbudowywać w extra aplikacji:

// app.config.ts
extra: {
  sentryDsn: process.env.SENTRY_DSN ?? '',
  mapsApiKey: process.env.MAPS_API_KEY ?? '',
  apiUrl: process.env.API_URL ?? 'https://api.open-meteo.com',
},
```

8.3. Zasady bezpiecznego zarządzania kluczami

1. Nigdy nie umieszczaj kluczy w kodzie źródłowym ani w plikach .env commitowanych do repozytorium.
2. Dodaj .env* do .gitignore przed pierwszym commitem. Sprawdź przez: git check-ignore -v .env.local
3. Używaj osobnych kluczy dla każdego środowiska. Skompromitowanie klucza dev nie dotyka produkcji.

4. Ograniczaj uprawnienia kluczy do minimum. Klucz API do pogody nie powinien mieć dostępu do bazy danych.
5. Rotuj klucze regularnie lub natychmiast po podejrzeniu wycieku.
6. Dla sekretów wstrzykiwanych do binarki (--dart-define w Flutter, extra w Expo) pamiętaj, że są one widoczne po inżynierii wstecznej. Nie uwywaj tej metody dla hasel użytkowników.

8.4. EXPO_TOKEN - klucz do CI

```
# Generowanie tokenu dostępowego Expo (jednorazowo):
# 1. Przejdź na expo.dev > Ustawienia konta > Access Tokens
# 2. Kliknij 'Create Token', nadaj nazwę (np. 'github-ci-token')
# 3. Skopiuj token (widoczny tylko raz!)
# 4. Dodaj do GitHub Secrets jako EXPO_TOKEN

# Token jest używany przez expo/expo-github-action do autentykacji EAS CLI w CI:
- uses: expo/expo-github-action@v8
  with:
    eas-version: latest
    token: ${{ secrets.EXPO_TOKEN }}

# UWAGA: Token ma te same uprawnienia co Twoje konto Expo.
# Nie udostępniaj go i nie umieszczaj w logach.
```

9. Monitorowanie - Sentry i Firebase Crashlytics

Każda aplikacja w produkcji napotka błędy, o których nie było mowy w trakcie testów. Monitorowanie krytycznych awarii w czasie rzeczywistym pozwala reagować zanim użytkownik zrezygnuje z aplikacji. Sentry i Firebase Crashlytics to dwa najpopularniejsze narzędzia w ekosystemie React Native.

Cecha	Sentry	Firebase Crashlytics
Jezyki / platformy	React Native, Flutter, web, backend (Node, Python, Go, ...). Jeden dashboard dla całego stosu.	Głównie aplikacje mobilne (iOS, Android). Ścisła integracja z Firebase Analytics.
Darmowy plan	5000 błędów miesięcznie, 14 dni retencja, 1 użytkownik.	Nieograniczone raporty crashy (Firebase jest bezpłatny dla większości użytkowników).
Informacje o błędzie	Stack trace, breadcrumby (ostatnie akcje użytkownika), zmienne lokalne, wersja OS.	Stack trace, wersja aplikacji, model urządzenia, liczba dotknięć użytkowników.
Alerty	E-mail, Slack, PagerDuty przy nowych lub regresujących błędach.	E-mail. Integracja ze Slackiem wymaga Firebase Extensions.
Integracja z RN	@sentry/react-native. Automatyczne wychwytywanie niezłapanych wyjątków.	@react-native-firebase/crashlytics. Wymaga native build (nie działa w Expo Go).

9.1. Konfiguracja Sentry

```
# Instalacja:
npm install @sentry/react-native
npx @sentry/wizard@latest -i reactNative # Automatyczna konfiguracja
# Wizard skonfiguruje: app.json, metro.config.js, babel.config.js

// app/_layout.tsx - inicjalizacja Sentry przed uruchomieniem aplikacji
import * as Sentry from '@sentry/react-native';
import { Config } from '@constants/config';

Sentry.init({
  dsn: Config.sentryDsn,          // Z EAS Secrets przez extra w app.config.ts
  enabled: Config.isProd,        // Wylacz w dev - nie zasmiecaj dashboardu
  tracesSampleRate: 0.2,        // 20% transakcji do profilowania wydajnosci
  environment: Config.appVariant, // 'development' | 'preview' | 'production'
  attachStacktrace: true,
});

// Przechwytywanie wszystkich niezłapanych błędów:
export default Sentry.wrap(RootLayout);
// Owin główny komponent - Sentry przechwyci wszystkie Unhandled Rejections

// Reczne logowanie błędu (nie-fatalnego):
try {
  const data = await fetchWeather(city);
} catch (error) {
  Sentry.captureException(error, {
    tags: { feature: 'weather', city },
    level: 'error',
  });
  // Pokaz komunikat uzytkownikowi i kontynuuj
}

// Breadcrumbs - slad akcji uzytkownika przed bledem:
Sentry.addBreadcrumb({
  message: `Uzytkownik wyszukal miasto: ${city}`,
  category: 'user-action',
  level: 'info',
});
```

Source maps w Sentry

bez source maps stack trace pokazuje zaciemniony kod (a.b.c zamiast fetchWeather). EAS Build automatycznie generuje source maps i może je wysłać do Sentry przy użyciu sentry-expo lub @sentry/react-native. Konfiguracja w eas.json: dodaj 'sentry': { 'uploadSourceMaps': true } do profilu production.

10. Zadania do wykonania

Zadania są kontynuacją projektu WeatherApp v2 z React Native Lab 2. Każde zadanie jest niezależne i można je weryfikować osobno bez oczekiwania na zakończenie poprzednich.

Zadanie 1 (20 pkt) - Konfiguracja EAS i wariantow aplikacji

8. Załóż konto na expo.dev (darmowe). Zainstaluj EAS CLI: `npm install -g eas-cli`. Zaloguj się: `eas login`. Połącz projekt: `eas init` (w katalogu WeatherApp).
9. Skonfiguruj EAS Build: `eas build:configure`. Utwórz plik `eas.json` z trzema profilami: `development` (APK, `developmentClient: true`), `preview` (APK, `internal`) i `production` (AAB, `store, autoIncrement: true`).
10. Zamień `app.json` na dynamiczny `app.config.ts`. Zdefiniuj obiekt `variants` z trzema wpisami (`development`, `preview`, `production`). Każdy wariant ma osobną nazwę, `package name` i ścieżkę do ikony.
11. Utwórz plik `constants/config.ts` czytający wartości z `Constants.expoConfig.extra`. Dodaj badge środowiska w `HomeScreen` widoczny tylko dla `development` i `preview`.
12. Uruchom lokalny build `development`: `eas build --platform android --profile development --local`. Zainstaluj APK na emulatorze i potwierdź że badge [DEV] jest widoczny.

WERYFIKACJA: Trzy warianty aplikacji mogą być zainstalowane jednocześnie na emulatorze (trzy ikony z różnymi nazwami). Plik `app.config.ts` kompiluje się bez błędów TypeScript (`npx tsc --noEmit`).

Zadanie 2 (25 pkt) - Podpisywanie i budowanie

13. Wygeneruj keystore poleceniem `keytool`. Dodaj `weather-upload.jks` i `android/key.properties` do `.gitignore`. Sprawdź przez `git status` że pliki nie są śladowe.
14. Dodaj keystore do EAS Credentials: `eas credentials --platform android`. Wybierz 'Set up a new keystore' > 'Use existing keystore'. Podaj ścieżkę, alias i hasło.
15. Uruchom chmurowy build `production`: `eas build --platform android --profile production`. Skopiuj link do dashboardu EAS i obserwuj postępy w przeglądarce.
16. Po zakończeniu buildu pobierz AAB: `eas build:download`. Sprawdź plik poleceniem: `file app-production.aab`. Plik powinien być rozpoznany jako ZIP archive.
17. Opcjonalnie: załóż konto Google Play (25 USD) i wyslij AAB do Internal Testing: `eas submit --platform android --profile production`.

WERYFIKACJA: Plik `.aab` istnieje i ma rozmiar > 10 MB. Historia buildów w EAS (`eas build:list`) pokazuje zakończony build `production`. `.gitignore` chroni klucze.

Zadanie 3 (35 pkt) - Pipeline CI/CD z GitHub Actions

18. Utwórz repozytorium GitHub i wypchnij projekt WeatherApp. Wygeneruj token Expo: `expo.dev` > Ustawienia > Access Tokens. Dodaj `EXPO_TOKEN` jako GitHub Secret.
19. Utwórz plik `.github/workflows/ci.yml` z jobem `test`: `checkout`, `setup-node (v20)`, `npm ci`, `npx tsc --noEmit`, `npm test --watchAll=false --ci`.
20. Dodaj job `build-android` uruchamiany tylko na gałęzi `main`, zależy od `test` (`needs: test`). Krok: `expo/expo-github-action@v8` z `EXPO_TOKEN`, potem `eas build --platform android --profile production --non-interactive --no-wait`.
21. Przetestuj pipeline: wypchnij commit na `main` i obserwuj GitHub Actions. Oba joby powinny zakończyć się zielonym statusem. Zrzut ekranu załącz do sprawozdania.
22. Opcjonalnie: dodaj EAS Secret `SENTRY_DSN` przez CLI (`eas secret:create`) i przeczytaj go w `app.config.ts`. Zainicjalizuj Sentry w `app/_layout.tsx` z `enabled: Config.isProd`.

WERYFIKACJA: Screenshot GitHub Actions z zielonymi jobami: `test` i `build-android`. EAS dashboard pokazuje nowy build uruchomiony przez CI (rozpoznawalny po tytule commitu).

Zadanie 4 (20 pkt) - Dokumentacja i monitorowanie

23. Napisz plik README.md w repozytorium GitHub zawierający: opis projektu, instrukcje uruchomienia każdego wariantu (development/preview/production), opis struktury eas.json i opis pipeline CI/CD.
24. Dodaj do projektu EAS Update: eas update:configure. Opublikuj aktualizację OTA: eas update --branch main --message 'Aktualizacja UI'. Sprawdź czy zmiana jest widoczna na urządzeniu bez reinstalowania aplikacji.
25. Dodaj Sentry do projektu (pakiet @sentry/react-native). Zainicjalizuj w app/_layout.tsx z enabled: Config.isProd. Otocz główny komponent przez Sentry.wrap(). Dodaj jedno ręczne logowanie błędu w bloku catch.
26. Sprawdź poprawność całej konfiguracji: npx tsc --noEmit (brak błędów TypeScript), npm test --watchAll=false (testy przechodzą), eas build --platform android --profile development --local (lokalny build działa).

WERYFIKACJA: README.md w repozytorium. EAS Update skonfigurowany (widoczny w expo.dev). Sentry zainicjalizowany w kodzie. Wszystkie polecenia weryfikacyjne bez błędów.

11. Kryteria oceniania

Zadanie	Punkty	Co weryfikuje prowadzący
Zad. 1: EAS i warianty	20 pkt	Trzy warianty na emulatorze, app.config.ts bez błędów TS, badge środowiska w UI.
Zad. 2: Podpisywanie	25 pkt	Plik .aab, historia buildów w EAS, .gitignore chroni klucze.
Zad. 3: CI/CD	35 pkt	Zielone joby GitHub Actions, build wyzwolony przez CI w EAS, EXPO_TOKEN jako Secret.
Zad. 4: Dokumentacja	20 pkt	README.md, EAS Update, Sentry w kodzie, polecenia weryfikacyjne bez błędów.
RAZEM	100 pkt	

Ocena	Punkty / wymagania
5.0	90-100 pkt - Wszystkie zadania. EAS Submit do Play Store. Sentry ze source maps. README kompletny.
4.5	80-89 pkt - Zadania 1-3 kompletne. README podstawowy. EAS Update skonfigurowany.
4.0	70-79 pkt - Zadania 1-2 + GitHub Actions z jobem test i build-android.
3.5	60-69 pkt - Zadania 1-2. Trzy warianty i podpisany build. Brak CI/CD.
3.0	50-59 pkt - Zadanie 1: trzy warianty działają, app.config.ts poprawny.
2.0	0-49 pkt - Projekt nie kompiluje lub EAS nie jest skonfigurowany.

12. EAS vs flutter build - tabela porównawcza

Jeżeli ukończone zostały laboratoria z Fluttera, poniższe zestawienie pomoże przełączyć zapas wiedzy na odpowiedniki w ekosystemie React Native i EAS.

Flutter	React Native / EAS - odpowiednik
<code>flutter build apk --release</code>	<code>eas build --platform android --profile preview</code>
<code>flutter build appbundle --release</code>	<code>eas build --platform android --profile production</code>
<code>flutter build ipa --release</code>	<code>eas build --platform ios --profile production</code>
<code>keytool -genkey</code> (lokalny keystore)	<code>eas credentials --platform android</code> (EAS zarządza lub importujesz własny)
<code>android/key.properties</code>	EAS Secrets (sekrety przechowywane w chmurze Expo)
<code>flutter build --flavor dev -t lib/main_dev.dart</code>	<code>APP_VARIANT=development eas build --profile development</code>
Flavors w <code>build.gradle</code> (<code>productFlavors</code>)	Warianty w <code>app.config.ts</code> (obiekt <code>variants</code> + <code>process.env.APP_VARIANT</code>)
<code>AppConfig.initDev()</code> w <code>main_dev.dart</code>	<code>Config.ts</code> czytający <code>Constants.expoConfig.extra.appVariant</code>
<code>pubspec.yaml</code> <code>version: 1.0.0+47</code>	<code>app.config.ts</code> <code>version</code> + <code>versionCode</code> / <code>autoIncrement</code> w <code>eas.json</code>
<code>--obfuscate --split-debug-info</code>	Minifikacja i source maps przez Sentry + EAS Build automatycznie
Fastlane supply (upload do Play)	<code>eas submit --platform android</code>
<code>r0adkll/upload-google-play</code> (GitHub Action)	<code>expo/expo-github-action</code> + <code>eas submit</code> w workflow
Firebase Crashlytics dla Flutter	<code>@sentry/react-native</code> lub <code>@react-native-firebase/crashlytics</code>
<code>flutter analyze</code>	<code>npx tsc --noEmit</code> + <code>npm run lint</code> (ESLint)
<code>flutter test --coverage</code>	<code>npm test -- --coverage --watchAll=false</code>
<code>debugShowCheckedModeBanner: false</code>	Brak bezpośredniego odpowiednika; badge środowiska w UI
Hot Reload (r w terminalu)	Fast Refresh (automatyczny przy zapisie pliku)
<code>flutter doctor</code>	<code>eas diagnostics</code> (sprawdza konfigurację projektu EAS)
<code>subosito/flutter-action</code> (GitHub Action)	<code>expo/expo-github-action</code> (GitHub Action)

13. Najczęstsze błędy i ich rozwiązania

Błąd / objaw	Przyczyna i rozwiązanie
eas build: 'Could not find the project'	Projekt nie jest powiazany z kontem Expo. Uruchom eas init w katalogu projektu. Sprawdź czy w app.json/app.config.ts jest pole 'slug' i 'owner'.
eas build: 'Keystore not found'	Poswiadczenia Android nie sa skonfigurowane dla projektu. Uruchom eas credentials --platform android i wybierz 'Set up a new keystore' lub zaimportuj istniejący.
Build zakonczony bledem: 'Gradle build failed'	Najczestsze przyczyny: brakujacy package name w app.config.ts, bledna wersja Gradle lub konflikty zaleznosci. Sprawdź logi szczegolowe w dashboardzie EAS (zakladka Build Logs).
app.config.ts: 'Cannot find module expo/config'	Brakuje pakietu expo. Upewnij sie ze expo jest w dependencies (nie devDependencies) i wykonaj npm ci.
process.env.APP_VARIANT jest undefined	Przy lokalnym uruchamianiu (npx expo start) zmienne z eas.json nie sa wstrzykiwane. Ustaw recznie: APP_VARIANT=development npx expo start. Lub sprawdź wartosc domyslana w app.config.ts.
GitHub Actions: 'EXPO_TOKEN not found'	Token nie zostal dodany jako GitHub Secret lub ma nieprawidlowa nazwe. Przejdź do: Settings > Secrets and variables > Actions > New repository secret. Nazwa musi byc dokładnie EXPO_TOKEN.
Sentry: stack trace pokazuje zaciemniony kod	Source maps nie zostaly wyslane do Sentry. Sprawdź konfiguracje @sentry/wizard i upewnij sie ze sentry.properties istnieje. Uruchom eas build --profile production (nie --local) aby EAS wyslal source maps.
eas submit: 'Service account credentials are invalid'	Plik service-account.json jest nieaktualny lub konto serwisowe nie ma uprawnień. W Google Play Console: Konfiguracja > Dostep do API > Konta uslug. Przyznaj role 'Release Manager'.
EAS Update: 'Update not applied'	Kanal aktualizacji (branch) nie pasuje do konfiguracji w eas.json lub wersja runtime różni sie od zainstalowanej. Upewnij sie ze eas update --branch main i ze build ma to samo runtimeVersion co update.
Constants.expoConfig.extra jest undefined	Odczytujesz extra przed zakonczeniem inicjalizacji Expo, lub pole extra nie zostalo zdefiniowane w app.config.ts. Sprawdź czy app.config.ts eksportuje extra i czy importujesz Constants z expo-constants (nie z react-native).

Komendy diagnostyczne

```
# Sprawdź konfiguracje EAS projektu:
eas diagnostics

# Podgląd wygenerowanej konfiguracji app.config.ts dla danego wariantu:
APP_VARIANT=production npx expo config --type introspect

# Sprawdź zapisane poswiadczenia Android:
eas credentials --platform android

# Lista buildow z ostatnich 7 dni:
eas build:list --limit 10 --status finished
```

```
# Pobierz logi konkretnego buildu:
eas build:view <BUILD_ID>

# Weryfikacja typow TypeScript bez kompilacji:
npx tsc --noEmit

# Sprawdź czy .gitignore chroni klucze:
git check-ignore -v weather-upload.jks
git check-ignore -v android/key.properties

# Pełny reset cache (gdy Metro lub EAS zachowuje sie dziwnie):
rm -rf node_modules .expo
npm ci
npx expo start --clear
```

Instrukcja Laboratoryjna React Native 3 - EAS Build, Publikacja, CI/CD z GitHub Actions

Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki

Następne ćwiczenie: [React Native 4 - Testy end-to-end z Detox](#), [React Native Web](#), [Monorepo](#)

ĆWICZENIE LABORATORYJNE REACT NATIVE 4

Detox · RT Web · Monorepo z Turborepo

Programowanie Aplikacji Mobilnych · moduł RN

Projekt WeatherApp v4, testowanie E2E, wersja webowa i monorepo

Wymagania wstępne: ukończony React Native Lab 3 (EAS Build, CI/CD, podpisywanie)

Spis treści

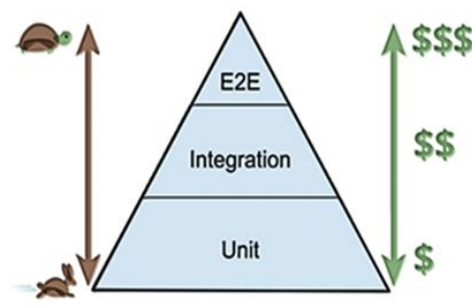
1. Piramida testów w aplikacjach mobilnych.....	1
2. Detox - testy end-to-end na prawdziwym urządzeniu.....	2
3. Pisanie testów Detox - selektory, akcje i asercje.....	5
4. Detox w potoku CI/CD.....	7
5. React Native Web, czyli jedna baza kodu na mobile i w (l)web.....	9
6. Współdzielenie kodu między platforma mobilna a webowa.....	11
7. Monorepo z Turborepo - zarządzanie wieloma pakietami.....	12
8. Współdzielony pakiet komponentów UI.....	14
9. Testowanie dostępności - a11y w praktyce.....	15
10. Profilowanie wydajności - FlashList i optymalizacja list.....	16

1. Piramida testów w aplikacjach mobilnych

Dobre pokrycie testami nie oznacza posiadania jak największej testów E2E. Oznacza odpowiednie proporcje między trzema poziomami: testami jednostkowymi, integracyjnymi i end-to-end. Piramida testów to model, który precyzuje te proporcje i wyjaśnia, dlaczego nie należy budować testów od góry.

Analogia kontroli jakości w fabryce samochodów

Testy jednostkowe to sprawdzenie każdej części osobno (silnik, hamulce, alternator). Szybko i tanio. Testy integracyjne to złożenie zespołów i sprawdzenie współpracy. Czy silnik jest poprawnie podłączony do skrzyni biegów. Testy E2E to jazda próbna cała trasa. Kosztowna, powolna, ale jedyna, która potwierdza że wszystko działa razem. Fabryka stosuje je w odpowiednich proporcjach.



Analogia na przykładzie kontroli jakości w fabryce samochodów

Testy jednostkowe to sprawdzenie każdej części osobno (silnik, hamulce, alternator). Szybko i tanio. Testy integracyjne to złożenie zespołów i sprawdzenie współpracy - czy silnik jest

poprawnie podłączony do skrzyni biegów. Testy E2E to jazda próbna cała trasa - kosztowna, powolna, ale jedyna, która potwierdza że wszystko działa razem. Fabryka stosuje je w odpowiednich proporcjach.

Poziom testu	Narzędzie w RN	Szybkość	Koszt	Proporcja
Jednostkowe	Jest + TypeScript	~1 ms / test	Niski	~70%
Integracyjne (widżetów)	React Native Testing Library	~100 ms / test	Średni	~20%
End-to-end	Detox	~5-30 s / test	Wysoki	~10%

1.1. Czym różni się Detox od RNTL?

React Native Testing Library (używana w Labie 2) renderuje komponenty w środowisku Node.js bez prawdziwego urządzenia. Detox uruchamia pełną aplikację na emulatorze lub fizycznym urządzeniu i steruje nią programatycznie, tak jak robiłby to użytkownik.

Cecha	RNTL (Lab 2)	Detox (ten lab)
Środowisko uruchomienia	Node.js - brak emulatora	Prawdziwy emulator lub urządzenie fizyczne
Co testuje	Renderowanie komponentu, props, stan, zdarzenia w izolacji	Cała aplikacja: nawigacje, HTTP, local storage, animacje, gesty
Czas jednego testu	Milisekundy	Sekundy (wymaga uruchomienia emulatora i aplikacji)
Czy wymaga budowania	Nie - uruchamia się przez Jest bezpośrednio	Tak - Detox buduje debug APK przed uruchomieniem testów
Co sprawdza najlepiej	Logikę komponentów, transformację danych, niestandardowe hooki	Ścieżki krytyczne użytkownika: logowanie, wyszukiwanie, zakup
Stabilność	Bardzo wysoka - deterministyczne	Średnio wysoka - zależy od szybkości emulatora, animacji, sieci

Zasada 10%

Testy E2E to 10% całości, bo są drogie w pisaniu, wolne w uruchamianiu i wymagają utrzymania przy każdej zmianie UI. Pisz Detox tylko dla ścieżek krytycznych: logowanie, kluczowy przepływ zakupu, wyszukiwanie. Reszta pokrycia - przez RNTL i Jest.

2. Detox - testy end-to-end na prawdziwym urządzeniu

Detox to framework E2E stworzony przez Wix Engineering. Jego główną przewagą nad konkurentami (Appium, Maestro) jest synchronizacja z wątkiem JS aplikacji: Detox czeka aż aplikacja zakończy wszystkie operacje asynchroniczne (fetch, animacje, nawigacje) zanim wykona następną akcję.

Eliminuje to potrzebe recznie pisanych usniec (await sleep(2000)), ktore sa glownym powodem nie-stabilnych testow E2E.

Analogia jako mówca i tłumacz synchroniczny

Appium i inne narzędzia działają jak tłumacz sekwencyjny: steruj aplikacją, zaczekaj arbitralną ilość czasu (sleep), sprawdź wynik. Detox działa jak mówca z tłumaczem synchronicznym: mówca zatrzymuje się i kontynuuje dopiero gdy tłumacz skończy bieżące zdanie. Aplikacja może kontynuować następny krok dopiero gdy obecny jest w pełni zakończony.

2.1. Instalacja i konfiguracja

```
# Instalacja Detox CLI globalnie:
npm install -g detox-cli

# Instalacja zależności w projekcie:
npm install --save-dev detox @config-plugins/detox jest-circus

# Konfiguracja Detox w projekcie Expo (expo-detox-plugin):
npm install --save-dev expo-detox-plugin

# Dodaj plugin do app.config.ts:
plugins: [
  ['expo-detox-plugin', {}],
],

# Przebuduj projekt natywny po dodaniu pluginu:
npx expo prebuild --clean
# Expo wygeneruje katalogi android/ i ios/ z natywnym kodem
```

2.2. Plik .detoxrc.js - konfiguracja testów

.detoxrc.js - konfiguracja Detox w katalogu głównym projektu

```
/** @type {Detox.DetoxConfig} */
module.exports = {
  testRunner: {
    args: {
      '$0': 'jest',
      config: 'e2e/jest.config.js', // Osobny plik konfiguracyjny dla testów E2E
    },
    jest: { setupTimeout: 120000 }, // 2 minuty na uruchomienie aplikacji
  },
  apps: {
    'android.debug': {
      type: 'android.apk',
      // APK zbudowany przez 'detox build' (ponizej)
      build: 'cd android && ./gradlew assembleDebug assembleAndroidTest -DtestBuildType=debug',
      binaryPath: 'android/app/build/outputs/apk/debug/app-debug.apk',
    },
  },
}
```

```

    },
    'android.release': {
      type: 'android.apk',
      build: 'cd android && ./gradlew assembleRelease assembleAndroidTest -DtestBuildType=release',
      binaryPath: 'android/app/build/outputs/apk/release/app-release.apk',
    },
  },
  devices: {
    emulator: {
      type: 'android.emulator',
      device: { avdName: 'Pixel_7_API_34' }, // Nazwa AVD z Android Studio
    },
  },
  configurations: {
    // Konfiguracja uzywana podczas: detox test -c android.debug
    'android.debug': {
      device: 'emulator',
      app: 'android.debug',
    },
    'android.release': {
      device: 'emulator',
      app: 'android.release',
    },
  },
};

```

2.3. Konfiguracja Jest dla testow E2E

e2e/jest.config.js - osobna konfiguracja Jest dla Detox

```

module.exports = {
  rootDir: '..', // Katalog glowny projektu
  testMatch: ['<rootDir>/e2e/**/*.e2e.ts'], // Wzorzec nazw plikow testowych
  testTimeout: 120000, // 2 minuty - E2E sa wolniejsze niz testy jednostkowe
  maxWorkers: 1, // Jeden watek - tylko jeden emulator
  globalSetup: 'detox/runners/jest/globalSetup',
  globalTeardown: 'detox/runners/jest/globalTeardown',
  reporters: ['detox/runners/jest/reporter'],
  testEnvironment: 'detox/runners/jest/testEnvironment',
  verbose: true,
};

```

2.4. Przygotowanie i uruchomienie testow

```

# Krok 1: Budowanie aplikacji do testow (raz, potem tylko przy zmianach natywnych)
detox build -c android.debug

```

```
# Krok 2: Uruchomienie testów E2E
detox test -c android.debug

# Uruchomienie wybranych testów (grep po nazwie):
detox test -c android.debug --testNamePattern='wyszukiwanie miasta'

# Uruchomienie z raportem HTML (wymaga jest-html-reporter):
detox test -c android.debug --reporters=jest-html-reporter

# Dodaj skrypty do package.json dla wygody:
"scripts": {
  "e2e:build": "detox build -c android.debug",
  "e2e:test": "detox test -c android.debug"
}
```

3. Pisanie testów Detox - selektory, akcje i asercje

Test Detox wygląda podobnie do testu RNTL: opisujesz akcje użytkownika i sprawdzasz oczekiwany stan. Różnica polega na tym, że Detox steruje prawdziwą aplikacją przez API automatyzacji Androida (UIAutomator2) lub iOS (XCTest). Selektory, akcje i asercje to trzy filary pisania testów E2E.

3.1. Selektory, czyli jak znaleźć element

Selektor	Co wyszukuje / Kiedy używać
<code>element(by.id('search-input'))</code>	Szuka po prop <code>testID='search-input'</code> . Najbardziej niezawodny. Ustaw <code>testID</code> na kluczowych elementach.
<code>element(by.text('Warszawa'))</code>	Szuka po widocznym tekście. Podatny na zmiany etykiet i i18n - unikaj w testach wielojęzycznych.
<code>element(by.label('Szukaj'))</code>	Szuka po <code>accessibilityLabel</code> . Dobre dla elementów bez widocznego tekstu (ikony, przyciski graficzne).
<code>element(by.type('TextInput'))</code>	Szuka po typie natywnego komponentu. Używaj gdy brak <code>testID</code> i brak tekstu.
<code>element(by.id('list')).atIndex(0)</code>	Pierwszy element z podanym <code>testID</code> . Przydatny dla list gdzie wiele elementów ma podobne ID.

3.2. Akcje i asercje

Metoda	Co robi / Odpowiednik gesty użytkownika
<code>await element(...).tap()</code>	Kliknięcie (tap). Odpowiednik <code>onPress</code> .
<code>await element(...).longPress()</code>	Długie przyciśnięcie. Odpowiednik <code>onLongPress</code> .
<code>await element(...).typeText('Gdansk')</code>	Wpisanie tekstu w pole. Odpowiednik wpisywania przez klawiaturę.
<code>await element(...).clearText()</code>	Wyczyszczenie pola tekstowego.
<code>await element(...).replaceText('nowy')</code>	Zamiana całej zawartości pola jednym krokiem.
<code>await element(...).scroll(200, 'down')</code>	Przewinięcie o 200 pikseli w dół. Przydatne dla

	dlugich list.
<code>await element(...).swipe('left', 'slow')</code>	Swipe w lewo powolnym gestem. Uzywaj 'fast' lub 'slow' dla roznej predkosci.
<code>await expect(element(...)).toBeVisible()</code>	Asercja: element jest widoczny na ekranie.
<code>await expect(element(...)).toHaveText('22.5°C')</code>	Asercja: element zawiera dokładnie ten tekst.
<code>await expect(element(...)).toExist()</code>	Asercja: element istnieje w drzewie (niekoniecznie widoczny).
<code>await expect(element(...)).not.toBeVisible()</code>	Asercja negatywna: element nie jest widoczny (np. spinner po zaladowaniu).

3.3. Kompletny test dla wyszukiwania pogody

e2e/weather-search.e2e.ts

```
import { device, element, by, expect as detoxExpect } from 'detox';

describe('Wyszukiwanie pogody', () => {

  // beforeAll: uruchom raz przed wszystkimi testami w tym bloku
  beforeAll(async () => {
    await device.launchApp({
      newInstance: true, // Swieza instancja aplikacji (czysty stan)
      permissions: {
        location: 'always', // Zgoda na lokalizacje jesli aplikacja jej wymaga
      },
    });
  });

  // beforeEach: uruchom przed kazdym testem
  beforeEach(async () => {
    await device.reloadReactNative(); // Resetuje stan JS bez reinstalowania
  });

  it('powinien wyswietlic wyniki dla istniejacego miasta', async () => {
    // ARRANGE: znajdz pole wyszukiwania
    const searchInput = element(by.id('search-input'));

    // ACT: wpisz miasto i zatwierdz
    await searchInput.tap();
    await searchInput.typeText('Gdansk');
    await element(by.id('search-button')).tap();

    // Detox automatycznie czeka az fetch sie zakonczy i UI sie zaktualizuje
    // Brak await sleep() - synchronizacja jest wbudowana

    // ASSERT: sprawdz czy wyniki sa widoczne
    await detoxExpect(element(by.id('weather-card'))).toBeVisible();
    await detoxExpect(element(by.id('city-name'))).toHaveText('Gdansk');

    // Sprawdz ze temperatura ma prawidlowy format (regex przez toHaveText nie istnieje -
    // uzyj toExist na elemencie z testID zamiast sprawdzac tresc)
    await detoxExpect(element(by.id('temperature-display'))).toExist();
  });
});
```

```
});

it('powinien wyswietlic komunikat bledu dla nieistniejacego miasta', async () => {
  const searchInput = element(by.id('search-input'));
  await searchInput.tap();
  await searchInput.typeText('ZupelnieNieistniejaceMiasto12345');
  await element(by.id('search-button')).tap();

  // Poczekaj az serwer odpowie bledem
  await detoxExpect(element(by.id('error-view'))).toBeVisible();
  await detoxExpect(element(by.id('error-message'))).toExist();
});

it('powinien dodac miasto do ulubionych i zachowac po przeladowaniu', async () => {
  // Wyszukaj miasto
  await element(by.id('search-input')).typeText('Wroclaw');
  await element(by.id('search-button')).tap();
  await detoxExpect(element(by.id('weather-card'))).toBeVisible();

  // Dodaj do ulubionych
  await element(by.id('favorite-button')).tap();

  // Przeladuj aplikacje (sprawdza persystencje AsyncStorage / MMKV)
  await device.reloadReactNative();

  // Przejdz do ekranu ulubionych i sprawdz
  await element(by.id('tab-favorites')).tap();
  await detoxExpect(element(by.text('Wroclaw'))).toBeVisible();
});
});
```

3.4. Oznaczanie elementów, czyli testID w komponentach

Aby Detox mogl niezawodnie odnalezc elementy, nalezy dodac prop testID do kluczowych elementow interfejsu. Wazne: testID jest widoczny przez Detox niezaleznie od jezyka interfejsu (i18n) i nie zmienia wygladu dla uzytkownika.

Cztery przykłady na dodawanie **testID** do komponentów

```
// 1. Pole wyszukiwania:
<TextInput
  testID='search-input' // Detox: element(by.id('search-input'))
  placeholder={t('searchPlaceholder')}
  value={query}
  onChangeText={setQuery}
/>

// 2. Karta pogody z dynamicznym ID (miasto jest czesc ID):
<Pressable
  testID={`weather-card-${weather.city}`} // Detox: by.id('weather-card-Warszawa')
  onPress={() => router.push(`/detail/${weather.city}`)}
>
  <Text testID='city-name'>{weather.city}</Text>
  <Text testID='temperature-display'>{weather.temperature}°C</Text>
</Pressable>
```

```
// 3. Przycisk bledu z retry:
<Pressable testID='retry-button' onPress={refetch}>
  <Text>Spróbuj ponownie</Text>
</Pressable>

// 4. Ukrycie testID w produkcji (opcjonalne, nie wymagane):
const testProps = __DEV__ ? { testID: 'dev-debug-panel' } : {};
<View {...testProps} />
```

Konwencja nazewnictwa testID

Używaj formatu 'nazwa-elementu' (kebab-case) lub 'ekran-element' (home-search-input).

Unikaj cyfr w stylu 'button1', bo nic nie mówi o roli elementu. Dobre nazwy: 'search-input', 'weather-card', 'error-retry-button', 'tab-favorites'.

4. Detox w potoku CI/CD

Uruchomienie testów Detox w GitHub Actions wymaga emulatora Androida w środowisku CI. Oficjalny runner ubuntu-latest nie ma akceleracji sprzętowej KVM, co spowalnia emulator. Rozwiązaniem jest repozytorium actions z wbudowanym emulatorem lub konfiguracja KVM na hostowanych runnerach.

4.1. Workflow GitHub Actions z Detox

.github/workflows/e2e.yml

```
name: Testy E2E (Detox)

on:
  push:
    branches: [main]
  workflow_dispatch:

jobs:
  e2e-android:
    name: Detox - Android
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - uses: actions/setup-java@v4
        with:
          distribution: 'temurin'
          java-version: '17'
```

- name: Instalacja zaleznosci
run: npm ci
- name: Konfiguracja emulatora Androida
uses: reactivecircus/android-emulator-runner@v2
with:
 api-level: 34
 arch: x86_64
 profile: pixel_7
 script: |
 # Budowanie i uruchomienie testow w ramach emulatora
 npm run e2e:build
 npm run e2e:test
- name: Zapisz artefakty (screenshots z nieudanych testow)
uses: actions/upload-artifact@v4
if: failure() # Tylko gdy testy nie przejdą
with:
 name: detox-artifacts
 path: artifacts/
 retention-days: 7

4.2. Zbieranie artefaktów z nieudanych testów

Detox automatycznie zapisuje zrzuty ekranu i nagrania wideo nieudanych testów. To kluczowe dla diagnostyki błędów w CI, gdzie nie można ręcznie podejrzeć emulatora.

.detoxrc.js - konfiguracja zbierania artefaktów

```
module.exports = {
  // ...
  artifacts: {
    rootDir: 'artifacts/',
    plugins: {
      screenshot: {
        shouldTakeAutomaticSnapshots: true,
        // 'all' - kazdy krok | 'failing' - tylko bledy | 'none' - wyłączone
        takeWhen: { testStart: false, testFailure: true, testDone: false },
      },
      video: {
        // Nagrywanie wideo calego testu (pomocne do analizy plynnosci animacji)
        enabled: false, // Wlacz tylko gdy potrzebne - zwieksze czas testu
      },
      log: { enabled: true }, // Logi systemowe z okresu testu
    },
  },
};
```

Tip - Stabilność testów E2E w CI

Emulatory w CI mogą być wolniejsze niż lokalne. Jeśli testy sporadycznie nie przechodzą z powodu timeoutów, zwiększ jest.setupTimeout w e2e/jest.config.js do 180000 (3 minuty). Można też rozważyć dodanie kroku warmup: device.launchApp() ale bez newInstance: true.

5. React Native Web, czyli jedna baza kodu na mobile i w (t)web

React Native Web (RNW) to biblioteka umożliwiająca uruchomienie aplikacji React Native w przeglądarce internetowej. Komponenty takie jak View, Text, Pressable, FlatList są renderowane jako natywne elementy HTML z odpowiednimi stylami CSS. Pozwala to na współdzielenie kodu między aplikacją mobilną a wersją webową bez przepisywania komponentów.

Analogia: adaptator podróźniczy

React Native Web działa jak adapter podróźniczy: ta sama wtyczka (komponent), inne gniazdko (platforma). View staje się `divem`, Text staje się `spanem`, Pressable staje się `buttonem`. Kod pozostaje niezmienny, a adapter zajmuje się konwersją. Nie każdy komponent ma adapter - platformospecyficzne API (Camera, Bluetooth) wymagają osobnej obsługi dla weba.

5.1. Konfiguracja React Native Web w projekcie Expo

React Native Web już jest zawarty w Expo SDK!

Wystarczy uruchomić wersję webową:

```
npx expo start --web
```

Lub nacisnąć 'w' podczas expo start

Budowanie statycznej wersji webowej do deploymentu:

```
npx expo export --platform web
```

Wynik: katalog `dist/` gotowy do hostowania na Netlify / Vercel / GitHub Pages

Konfiguracja w `app.config.ts` - włączenie webowej platformy:

```
export default ({ config }: ConfigContext): ExpoConfig => ({
  ...config,
  web: {
    output: 'static', // 'static' = SPA | 'server' = SSR (wymaga serwera)
    favicon: './assets/favicon.png',
    bundler: 'metro', // Metro bundler (zamiast domyślnego webpack)
  },
});
```

5.2. Komponenty współdzielone i platformospecyficzne

Nie wszystkie funkcje są dostępne na wszystkich platformach. React Native udostępnia dwa mechanizmy do obsługi różnic: plik z rozszerzeniem platformy (`Platform.OS`) i pliki z sufiksem `.native.ts` / `.web.ts`.

Technika	Jak działa	Kiedy używać
<code>Platform.OS</code>	Sprawdzenie w czasie wykonania: <code>if (Platform.OS === 'web') { ... }</code> . Kod obu ścieżek trafia do bundla.	Małe różnice logiczne, np. inny padding na web, inne komunikaty błędów.
<code>Platform.select()</code>	<code>Platform.select({ ios: ..., android: ..., web: ..., default: ... })</code> . Również w	Wartości stylów różne per platformą: <code>Platform.select({ web: '100vh',</code>

	czasie wykonania.	default: '100%' })).
Plik .native.ts / .web.ts	Metro automatycznie wybiera .native.ts dla mobile i .web.ts dla web. Dwa oddzielne pliki.	Duże różnice implementacji: cały hook lub serwis ma inną logikę per platforma.
Plik .ios.ts / .android.ts	Analogicznie do web - osobny plik per platforma mobilna.	Różnice między iOS a Android, np. inne API haptics.

hooks/useWindowHeight.native.ts - wersja mobilna

```
import { Dimensions } from 'react-native';

export function useWindowHeight(): number {
  return Dimensions.get('window').height;
  // Na mobile: fizyczna wysokość okna aplikacji
}

// hooks/useWindowHeight.web.ts - wersja webowa
import { useState, useEffect } from 'react';

export function useWindowHeight(): number {
  const [height, setHeight] = useState(window.innerHeight);

  useEffect(() => {
    const handler = () => setHeight(window.innerHeight);
    window.addEventListener('resize', handler);
    return () => window.removeEventListener('resize', handler);
  }, []);

  return height;
  // Na web: reaguje na zmianę rozmiaru okna przeglądarki
}

// W komponencie - importujesz ZAWSZE bez sufiksu:
import { useWindowHeight } from '@hooks/useWindowHeight';
// Metro automatycznie wybierze właściwy plik per platforma
```

5.3. Komponenty nieobsługiwane przez RNW

Komponent / API	Na mobile	Alternatywa na web
CameraView (expo-camera)	Pełna obsługa kamery	HTMLMediaElement przez ref lub biblioteka react-webcam
react-native-maps	Mapy natywne Google / Apple	react-leaflet lub Google Maps JS API przez iframę
Animated API (część)	Akceleracja sprzętowa przez natywny wątek	CSS transitions / Web Animations API przez useAnimatedStyle z Reanimated
Haptics (expo-haptics)	Wibracje i dotykowa informacja zwrotna	Brak odpowiednika - pomiń lub warunkuj przez Platform.OS !== 'web'

SQLite / MMKV	Natywna baza i sharedPreferences	IndexedDB lub localStorage przez kompatybilny adapter
---------------	----------------------------------	---

6. Wspoldzielenie kodu miedzy platforma mobilna a webowa

Celem React Native Web nie jest identyczny wyglad na wszystkich platformach, ale maksymalne wspoldzielenie logiki biznesowej: hookow, serwisow, modeli danych i store. Warstwa UI moze sie roznic, ale logika pozostaje ta sama.

6.1. Zasada podzialu kodu

Warstwa	Wspoldzielona?	Przyklad
Modele danych (typy TS)	Tak - 100%	interface WeatherData, interface ForecastDay - identyczne na wszystkich platformach
Serwisy API	Tak - 100%	weatherService.ts korzysta z fetch() dostepnego na web i mobile jednakowo
Store (Zustand)	Tak - 100%	favoritesStore.ts - zustand dziala na wszystkich platformach bez zmian
Custom hooki	Zwykle tak	useWeatherQuery - TanStack Query dziala identycznie. Wyjatek: hooki uzywajace natywnego API
Komponenty UI	Czesto tak	WeatherCard korzystajacy z View/Text/Image - wspoldzielony. Wyjatek: komponenty z native-only API
Nawigacja	Czesciowo	Expo Router dziala na web (URL-based). Niektore opcje nawigacji sa mobile-only
Konfiguracja natywna	Nie	AndroidManifest.xml, Info.plist, uprawnienia - tylko mobile

6.2. Adaptacyjny uklad - responsive design w RN

hooks/useBreakpoint.ts - wspoldzielony hook responsywnosci

```
import { useWindowDimensions } from 'react-native';
```

```
export type Breakpoint = 'mobile' | 'tablet' | 'desktop';
```

```
export function useBreakpoint(): Breakpoint {
  const { width } = useWindowDimensions();
  if (width >= 1200) return 'desktop';
  if (width >= 768) return 'tablet';
  return 'mobile';
}
```

```
// Uzycie w komponencie - rozny uklad per rozmiar ekranu:
```

```
const HomeScreen = () => {
  const breakpoint = useBreakpoint();
```

```

return (
  <View style={[
    styles.container,
    breakpoint === 'desktop' && styles.desktopContainer,
  ]}>
    {breakpoint !== 'mobile'
      ? <TwoColumnLayout /> // Sidebar + tresc glowna na duzych ekranach
      : <SingleColumnLayout />
    }
  </View>
);
};

const styles = StyleSheet.create({
  container: { flex: 1, backgroundColor: '#fff' },
  desktopContainer: { maxWidth: 1200, alignSelf: 'center', width: '100%' },
});

```

7. Monorepo z Turborepo - zarządzanie wieloma pakietami

Monorepo to struktura projektu, w której wiele powiazanych aplikacji i bibliotek jest przechowywanych w jednym repozytorium Git. Zamiast trzech oddzielnych repozytoriow (apps/mobile, apps/web, packages/shared), mamy jedno z wyraźnymi granicami między pakietami.

Analogia: biuro vs. oddzielne firmy

Trzy oddzielne repozytoria to trzy firmy: każda ma własną recepcję (konfigurację), działy (katalogi) i procedury (skrypty). Koordynacja wymaga telefonowania (npm link, wersjonowania) i spotkań (sync PR). Monorepo to jedno biuro z trzema zespołami: wspólna recepcja, wspólna biblioteka kodów (packages/shared), ale wyraźnie oddzielone przestrzenie robocze.

7.1. Kiedy monorepo ma sens?

Scenariusz	Rekomendacja
Jedna aplikacja mobilna, brak wersji webowej	Pojedyncze repozytorium - monorepo jest nadmiarowoscia.
Aplikacja mobilna + wersja webowa wspoldzielajace logike	Monorepo - naturalne miejsce na shared pakiet z hookami i serwisami.
Aplikacja mobilna + dashboard admin (web) + serwis backendowy	Monorepo - wspoldzielone typy TypeScript eliminuja desynchronizacje kontraktow API.
Wiele niezaleznych aplikacji mobilnych bez wspolnego kodu	Pojedyncze repozytoria - monorepo nie przynosi korzysci przy braku wspolnego kodu.

7.2. Struktura monorepo z Turborepo

```

weather-monorepo/
├── apps/
│   ├── mobile/           # Aplikacja React Native (Expo)
│   │   ├── app/          # Ekrany (Expo Router)
│   │   ├── package.json
│   │   └── app.config.ts
│   └── web/              # Wersja webowa (Next.js)
│       ├── app/          # App Router Next.js 14
│       ├── package.json
│       └── next.config.js
├── packages/
│   ├── shared/           # Współdzielona logika (TS, React)
│   │   ├── src/
│   │   │   ├── types/    # Interfejsy TypeScript
│   │   │   ├── services/ # Serwisy API (fetch)
│   │   │   ├── hooks/    # Custom hooks (useWeather, useFavorites)
│   │   │   └── index.ts   # Barrel export
│   │   ├── package.json  # name: '@weather/shared'
│   │   └── tsconfig.json
│   └── ui/               # Współdzielone komponenty UI (RN + RNW)
│       ├── src/
│       │   ├── WeatherCard.tsx
│       │   └── ForecastTile.tsx
│       ├── package.json  # name: '@weather/ui'
│       └── tsconfig.json
├── turbo.json            # Konfiguracja Turborepo
├── package.json          # Root - workspaces
└── tsconfig.base.json    # Wspólna konfiguracja TypeScript

```

7.3. Konfiguracja Turborepo

Inicjalizacja monorepo Turborepo:

```

npx create-turbo@latest weather-monorepo
# Wybierz: 'npm workspaces' jako package manager

```

Root package.json - definicja workspaces:

```

{
  "name": "weather-monorepo",
  "private": true,
  "workspaces": ["apps/*", "packages/*"],
  "devDependencies": {
    "turbo": "^2.0.0"
  },
  "scripts": {
    "dev": "turbo run dev", // Uruchamia dev dla wszystkich apps
    "build": "turbo run build", // Buduje wszystkie apps i packages
    "test": "turbo run test", // Testy we wszystkich pakietach
    "lint": "turbo run lint"
  }
}

```

// turbo.json - konfiguracja potoku Turborepo:

```

{
  "$schema": "https://turbo.build/schema.json",

```

```

"tasks": {
  "build": {
    // build apps/mobile zalezy od build packages/shared i packages/ui
    "dependsOn": ["^build"], // ^ = zadania z pakietow zaleznych
    "outputs": ["dist/**"] // Cache'owane pliki wyjsciowe
  },
  "test": {
    "dependsOn": ["^build"],
    "outputs": ["coverage/**"]
  },
  "dev": {
    "persistent": true, // Proces dev nie konczy sie (watch mode)
    "cache": false // Nie cachuj - dev zawsze uruchamia od nowa
  }
}
}

```

Kluczowa zaleta to cache Turborepo

Turborepo cachuje wyniki budowania na podstawie skrotu plikow wejsciowych. Jesli package/shared sie nie zmienil, ponowne uruchomienie turbo run build uzyta wynik z cache bez ponownego budowania. W CI zmniejsza to czas o 60-80% dla duzych projektow.

8. Wspoldzielony pakiet komponentow UI

Pakiet packages/ui zawiera komponenty React Native wspoldzielone miedzy aplikacja mobilna, a wersja webowa. Dzieki React Native Web te same komponenty renderują sie poprawnie w przeglądarce. Pakiet jest wewnętrzny (private: true) i nie jest publikowany w npm.

8.1. Konfiguracja pakietu shared UI

packages/ui/package.json

```

{
  "name": "@weather/ui",
  "version": "0.0.0",
  "private": true,
  "main": "./src/index.ts", // Punkt wejscia dla TypeScript
  "types": "./src/index.ts",
  "exports": {
    ".": "./src/index.ts"
  },
  "peerDependencies": {
    // PeerDependencies zamiast dependencies - kazda app dostarcza swoje wersje
    "react": "*",
    "react-native": "*"
  }
}

```

```

// packages/ui/src/index.ts - barrel export
export { WeatherCard } from './WeatherCard';
export { ForecastTile } from './ForecastTile';
export { ErrorView } from './ErrorView';

```

```
export { LoadingSpinner } from './LoadingSpinner';  
// Eksportuj również typy:  
export type { WeatherCardProps } from './WeatherCard';
```

8.2. Użycie współdzielonych komponentów

// apps/mobile/app/(tabs)/index.tsx - użycie z pakietu @weather/ui

```
import { WeatherCard, LoadingSpinner } from '@weather/ui';  
import { useWeather } from '@weather/shared'; // Współdzielony hook
```

// apps/web/app/page.tsx (Next.js) - ten sam import!

```
import { WeatherCard, LoadingSpinner } from '@weather/ui';  
import { useWeather } from '@weather/shared';
```

// Użycie w obu aplikacjach jest identyczne.

// Różnica: Next.js renderuje komponenty jako HTML na serwerze (SSR).

// React Native Web renderuje je jako HTML w przeglądarce (CSR).

// apps/mobile/package.json - deklaracja zależności wewnętrznej:

```
{  
  "dependencies": {  
    "@weather/ui": "*", // '*' = lokalna wersja z workspace  
    "@weather/shared": "*"   
  }  
}
```

// Instalacja zależności w całym monorepo (z katalogu root):

npm install

npm workspaces automatycznie linkuje @weather/ui i @weather/shared

Brak potrzeby 'npm link' ani lokalnego publishowania

8.3. Konfiguracja TypeScript dla monorepo

// tsconfig.base.json - konfiguracja bazowa w root monorepo

```
{  
  "compilerOptions": {  
    "strict": true,  
    "skipLibCheck": true,  
    "moduleResolution": "bundler",  
    "jsx": "react-native", // Wspólna konfiguracja JSX  
    "paths": {  
      // Aliasy ścieżek wspólne dla całego monorepo:  
      "@weather/shared": ["../../packages/shared/src/index.ts"],  
      "@weather/ui": ["../../packages/ui/src/index.ts"]  
    }  
  }  
}
```

// packages/shared/tsconfig.json - rozszerza bazową:

```
{
```

```
"extends": "../../tsconfig.base.json",
"compilerOptions": {
  "outDir": "./dist",
  "rootDir": "./src"
},
"include": ["src/**/*.ts", "src/**/*.tsx"]
}
```

9. Testowanie dostępności - a11y w praktyce

Dostępność (accessibility, a11y) to projektowanie aplikacji tak, aby były użyteczne dla osób z niepełnosprawnościami korzystających z czytników ekranu (TalkBack na Androidzie, VoiceOver na iOS). Testowanie a11y można prowadzić na trzech poziomach: statyczna analiza kodu, testy jednostkowe i ręczna weryfikacja z czytnikiem ekranu.

9.1. Automatyczna weryfikacja – eslint-plugin-jsx-a11y

Instalacja pluginu ESLint do a11y (dla wersji webowej):

```
npm install --save-dev eslint-plugin-jsx-a11y
```

.eslintrc.js - konfiguracja:

```
module.exports = {
  extends: [
    'expo',
    'plugin:jsx-a11y/recommended', // Zasady dostępności
  ],
  plugins: ['jsx-a11y'],
  rules: {
    // Blokuje obrazy bez alt:
    'jsx-a11y/alt-text': 'error',
    // Ostrzega gdy klikalny element nie ma roli:
    'jsx-a11y/interactive-supports-focus': 'warn',
  },
};
```

9.2. Testy a11y przez RNTL – jest-native-a11y

// __tests__/WeatherCard.a11y.test.tsx

```
import { render } from '@testing-library/react-native';
import { WeatherCard } from '@weather/ui';

const mockWeather = {
  city: 'Krakow', temperature: 18.5, humidity: 72,
  windSpeed: 8.0, weatherCode: 3, updatedAt: new Date(),
};

describe('WeatherCard - dostępność', () => {
```

```

it('powinien miec label czytnika ekranu', () => {
  const { getByRole } = render(<WeatherCard weather={mockWeather} />);

  // Karta ma role 'button' i accessibilityLabel
  const card = getByRole('button');
  expect(card).toBeTruthy();
  // Label powinien opisywac zawartosc w sposob zrozumialy bez wizualnego kontekstu
});

it('ikony dekoracyjne nie sa ogłaszane przez czytnik', () => {
  const { queryByRole } = render(<WeatherCard weather={mockWeather} />);

  // Elementy z importantForAccessibility='no' nie powinny byc dostepne
  // Brak elementow o roli 'image' - ikony sa dekoracyjne
  expect(queryByRole('image')).toBeNull();
});

it('pola tekstowe maja widoczne etykiety', () => {
  const { getAllByRole } = render(<WeatherCard weather={mockWeather} />);

  // Sprawdz ze wszystkie elementy tekstowe sa dostepne
  const textElements = getAllByRole('text');
  expect(textElements.length).toBeGreaterThanOrEqual(2); // Miasto + temperatura
});
});

```

9.3. Reczna weryfikacja z TalkBack

Automatyczne testy sprawdzaja strukture, ale nie zastapią weryfikacji z prawdziwym czytnikiem ekranu. TalkBack (Android) i VoiceOver (iOS) to narzedzia wbudowane w system operacyjny.

Krok	TalkBack (Android)	VoiceOver (iOS)
Wlaczenie	Ustawienia > Ulatwienia dostepu > Talk-Back > Wlacz. Lub: wlacz przez Skrot dostepu (trzy razy Power).	Ustawienia > Dostepnosc > VoiceOver > Wlacz. Lub: trzy razy boczny przycisk na iPhone.
Nawigacja	Swipe w prawo = nastepny element. Swipe w lewo = poprzedni. Podwojne tapniecie = aktywacja.	Swipe w prawo = nastepny element. Swipe w lewo = poprzedni. Podwojne tapniecie = aktywacja.
Co sprawdzic	Kolejnosc ogłaszania elementow, tresc etykiet, dostepnosc przyciskow, powiadomienia o zmianach stanu.	Te same elementy co na Androidzie. VoiceOver jest bardziej restrykcyjny dla elementow interaktywnych.

10. Profilowanie wydajnosci - FlashList i optymalizacja list

Listy sa najczestszym powodem problemow z wydajnoscia w aplikacjach React Native. FlatList jest wbudowanym rozwiazaniem, ale dla duzych zestawow danych (setki, tysiace elementow) moze byc za wolna. FlashList od Shopify jest zamiennikiem o znacznie wyzszej wydajnosci.

10.1. FlatList vs FlashList - roznice

Cecha	FlatList	FlashList (Shopify)
Algorytm recyklingu	Wirtualizacja przez okno renderowania (windowSize). Elementy spoza okna sa odmontowywane.	Natywny recykling podobny do RecyclerView (Android). Elementy sa reuzywane, nie odmontowywane.
Wydajnosć dla 1000+ elementow	Problemy z plynnoscia przy szybkim przewijaniu. Widoczne biale paski.	Stale 60fps nawet dla 10000 elementow dzieki natywnej warstwie.
Migracja z FlatList	- (baza)	Zamiana FlatList na FlashList i dodanie estimatedItemSize - 95% kompatybilnosci API.
Wymagany native build	Nie - dziala w Expo Go	Tak - wymaga npx expo prebuild

10.2. Migracja z FlatList na FlashList

```
# Instalacja:
npm install @shopify/flash-list
npx expo prebuild # Wymagany rebuild natywny

// Przed (FlatList):
import { FlatList } from 'react-native';

<FlatList
  data={weatherList}
  renderItem={({ item }) => <WeatherCard weather={item} />}
  keyExtractor={(item) => item.city}
/>

// Po (FlashList) - minimalna zmiana:
import { FlashList } from '@shopify/flash-list';

<FlashList
  data={weatherList}
  renderItem={({ item }) => <WeatherCard weather={item} />}
  keyExtractor={(item) => item.city}
  estimatedItemSize={120} // JEDYNA wymagana nowa wlasciwosc
  // estimatedItemSize: przyblizona wysokosc jednego elementu w dp
  // Uzyj DevTools > Performance Profiler aby zmierzyc realny rozmiar
/>
```

10.3. Profilowanie przez React DevTools Profiler

Uruchomienie React DevTools (wymagana wersja standalone)

```
npm install -g react-devtools
```

```
react-devtools
```

```
# Potem uruchom aplikacje przez Expo z wlaczonym trybie deweloperskim
```

```
# W React DevTools:
```

```
# 1. Zakladka Profiler > naciśnij czerwony przycisk Record
```

```
# 2. Wykonaj akcje ktora chcesz profilowac (np. szybkie przewijanie listy)
# 3. Zatrzymaj nagranie
# 4. Kliknij na pasek czasu aby zobaczyc ktore komponenty sie rerenderowaly

# Interpretacja wynikow:
# Szary komponent: nie renderowal sie w tej klatce (dobrze!)
# Zolty komponent: renderowal sie (sprawdz czy to konieczne)
# Czerwony komponent: renderowal sie dlugo (ponad 16ms - problem!)

# Naprawa zbędnych rerenderow - trzy techniki:

// 1. React.memo - zapobiega rerenderowi gdy props sie nie zmienil
export const WeatherCard = React.memo(({ weather, onPress }: WeatherCardProps) => {
  // Komponent nie rerenderuje sie gdy rodzic sie rerenderuje
  // ALE props weather jest porownywany przez referencje - uzyj dla obiektow niezmiennych
  return <View>{/* ... */</View>;
});

// 2. useCallback - stabilna referencja funkcji miedzy renderami
const handlePress = useCallback((city: string) => {
  router.push(`/detail/${city}`);
}, []); // Pusta tablica - funkcja tworzona raz

// 3. useMemo - memoizacja kosztownych obliczen
const sortedCities = useMemo(() => {
  return [...favorites].sort((a, b) => a.localeCompare(b));
}, [favorites]); // Przelicz tylko gdy favorites sie zmieni
```

11. Zadania do wykonania

Zadania sa niezalezne od siebie i moga byc realizowane w dowolnej kolejnosci. Kazde wprowadza jedna technologie z tego cwiczenia.

Zadanie 1 (25 pkt) - Testy E2E z Detox

1. Zainstaluj Detox i skonfiguruj projekt: `npm install --save-dev detox expo-detox-plugin`. Uruchom `npx expo prebuild --clean`. Utwórz plik `.detoxrc.js` z konfiguracja dla emulatora Android.
2. Dodaj prop `testID` do kluczowych elementow w `HomeScreen` i `DetailScreen`: `search-input`, `search-button`, `weather-card`, `city-name`, `temperature-display`, `error-view`, `tab-favorites`, `favorite-button`.
3. Zbuduj aplikacje do testow: `npm run e2e:build`. Napisz test `e2e/weather-search.e2e.ts` z trzema przypadkami testowymi: wyszukiwanie istniejacego miasta, blad dla nieistniejacego, persystencja ulubionych po przeladowaniu.
4. Uruchom testy: `npm run e2e:test`. Dolacz screenshot artefaktu (zrzutu ekranu z nieudanego testu) lub screenshota terminala z zielonymi testami do sprawozdania.
5. Opcjonalnie: dodaj job `e2e` do GitHub Actions workflow z `reactivecircus/android-emulator-runner@v2`.

WERYFIKACJA: detox test uruchamia sie bez bledow. Min. 3 testy E2E przechodza na emulatorze. `testID` dodane do kluczowych elementow. Katalog `artifacts/` istnieje po uruchomieniu testow.

Zadanie 2 (25 pkt) - React Native Web

6. Uruchom wersję webową projektu: `npx expo start --web`. Sprawdź jakie komponenty renderują się poprawnie, a które wymagają poprawek. Zidentyfikuj min. 2 komponenty problematyczne.
7. Utwórz plik `hooks/useWindowHeight.native.ts` i `hooks/useWindowHeight.web.ts` z różnymi implementacjami. Użyj nowego hooka w `HomeScreen` do dynamicznego obliczenia wysokości listy.
8. Zaimplementuj responsywny układ w `HomeScreen`: na web (`width >= 768px`) wyświetl dwie kolumny kart pogody obok siebie (`flexDirection: 'row'`, `flexWrap: 'wrap'`). Na mobile zachowaj pojedynczą kolumnę.
9. Zbuduj statyczną wersję webową: `npx expo export --platform web`. Sprawdź że katalog `dist/` istnieje i zawiera `index.html`. Opcjonalnie: opublikuj na Netlify (`netlify deploy --dir dist`).

WERYFIKACJA: `npx expo start --web` uruchamia się bez błędów. `HomeScreen` wyświetla dwie kolumny na szerokim oknie przeglądarki. Katalog `dist/` istnieje po budowaniu.

Zadanie 3 (30 pkt) - Monorepo z Turborepo

10. Utwórz nowe monorepo: `npx create-turbo@latest weather-monorepo`. Przenieś kod `WeatherApp` do `apps/mobile/`. Utwórz pakiet `packages/shared` z plikami: `types/weather.ts`, `services/weatherService.ts`, `hooks/useWeatherQuery.ts`.
11. Utwórz pakiet `packages/ui` z komponentami współdzielonymi: `WeatherCard.tsx`, `ForecastTile.tsx`, `ErrorView.tsx`. Skonfiguruj `package.json` dla obu pakietów (`@weather/shared`, `@weather/ui`). Dodaj zależności od `@weather/shared` i `@weather/ui` w `apps/mobile`.
12. Zaktualizuj importy w `apps/mobile`: zamień `'../components/WeatherCard'` na import z `'@weather/ui'`. Zamień `'../services/weatherService'` na import z `'@weather/shared'`. Sprawdź że kompilacja działa: `npx tsc --noEmit`.
13. Dodaj do `turbo.json` konfigurację tasków: `build` (`dependsOn: ['^build']`), `test` i `lint`. Uruchom `turbo run test` z roota monorepo i sprawdź że testy ze wszystkich pakietów przechodzą.

WERYFIKACJA: `turbo run build` kończy się bez błędów. `turbo run test` uruchamia testy ze wszystkich pakietów. Importy z `@weather/shared` i `@weather/ui` działają w `apps/mobile`.

Zadanie 4 (20 pkt) - Optymalizacja i dostępność

14. Zamień `FlatList` na `FlashList` w `HomeScreen`: `npm install @shopify/flash-list`, `npx expo prebuild`, zamień import i dodaj `estimatedItemSize={120}`. Uruchom `Performance Monitor` (Expo Dev Menu) i sprawdź czy FPS jest stabilne.
15. Dodaj `React.memo` do `WeatherCard` i `ForecastTile`. Dodaj `useCallback` do funkcji obsługi `onPress` w `HomeScreen`. Uruchom `React DevTools Profiler` i sprawdź że komponenty nie rerenderują się zbędnie.
16. Przeprowadź weryfikację dostępności: włącz TalkBack na emulatorze (Ustawienia > Ułatwienia dostępu > TalkBack). Sprawdź czy kolejność czytania kart jest logiczna i czy przyciski mają zrozumiałe etykiety. Popraw min. 2 miejsca z brakującymi `accessibilityLabel`.
17. Napisz min. 2 testy dostępności w pliku `__tests__/WeatherCard.a11y.test.tsx`: sprawdź że karta ma rolę `'button'`, sprawdź że `accessibilityLabel` jest ustawione i nie jest puste.

WERYFIKACJA: FlashList działa, FPS stabilne w Performance Monitor. React.memo dodane. TalkBack czyta karte poprawnie. 2 testy a11y przechodzą.

12. Kryteria oceniania

Zadanie	Punkty	Co weryfikuje prowadzący
Zad. 1: Detox E2E	25 pkt	Min. 3 testy E2E przechodzą. testID w komponentach. Konfiguracja .detoxrc.js.
Zad. 2: RN Web	25 pkt	Wersja web działa. Responsywny układ. Katalog dist/ po expo export.
Zad. 3: Monorepo	30 pkt	Struktura apps/mobile + packages/shared + packages/ui. turbo run build i test bez błędów.
Zad. 4: Optymalizacja	20 pkt	FlashList zamiast FlatList. React.memo. 2 testy a11y. TalkBack działa poprawnie.
RAZEM	100 pkt	

Ocena	Punkty / wymagania
5.0	90-100 pkt - Wszystkie zadania. Detox w CI (GitHub Actions). Monorepo z wersją Next.js. Coverage a11y > 80%.
4.5	80-89 pkt - Zadania 1-3 kompletne. Min. 2 testy a11y. FlashList zamiast FlatList.
4.0	70-79 pkt - Zadania 1-2 + Monorepo z współdzieloną logiką (packages/shared). Brak pełnego turbo run build.
3.5	60-69 pkt - Zadania 1-2. Detox z min. 2 testami. Wersja webowa działa.
3.0	50-59 pkt - Zadanie 1: min. 2 testy Detox przechodzą na emulatorze.
2.0	0-49 pkt - Detox nie jest skonfigurowany lub testy E2E nie uruchamiają się.

13. Najczęstsze błędy i ich rozwiązania

Błąd / objaw	Przyczyna i rozwiązanie
Detox: 'No simulators found for device query'	Nazwa AVD w .detoxrc.js nie pasuje do nazwy w Android Studio. Sprawdź dokładną nazwę przez: emulator -list-avds. Wpisz ją dosłownie w polu avdName.
Detox: 'Test timed out while waiting for idle'	Aplikacja ma niezakończone zadania asynchroniczne (fetch w petli, animacje bez stopu). Sprawdź czy wszystkie połączenia sieciowe kończą się (sukces lub błąd). W testach użyj mockowania sieci przez jest.mock.
Detox: element nie znaleziony po typeText()	Klawiatura wirtualna zasłania element lub focus nie został ustawiony. Dodaj await element(by.id('search-input')).tap() przed typeText(). Sprawdź czy testID jest poprawnie

	ustawiony w komponencie.
RN Web: 'Text strings must be rendered within a Text component'	Na web React Native jest bardziej restrykcyjny co do zagnieźdzonego tekstu. Sprawdź czy wszystkie stringi są w komponencie Text, także te warunkowo renderowane: {condition && 'tekst'} zamień na {condition && <Text>tekst</Text>}
RN Web: Ikony nie renderują się	@expo/vector-icons korzysta z fontów które mogą nie być załadowane na web. Dodaj do app.config.ts: web: { fonts: ['MaterialCommunityIcons'] } lub użyj react-native-svg z odpowiednimi ikonami.
Turborepo: 'Cannot find module @weather/shared'	npm install nie zostało uruchomione z katalogu root monorepo lub workspaces nie są poprawnie skonfigurowane. Uruchom npm install z root. Sprawdź że package.json w root ma pole workspaces: ['apps/*', 'packages/*'].
Turborepo: turbo run build nie buduje pakietów w dobrej kolejności	Brak zależy od dependsOn: ['^build'] w turbo.json dla zadania build. Karetą ^ oznacza 'najpierw zbuduj zależności pakietów'. Bez niej Turborepo może próbować budować apps/mobile przed packages/shared.
FlashList: białe paski podczas szybkiego przewijania	Wartość estimatedItemSize jest zbyt odległa od rzeczywistego rozmiaru elementów. Uruchom aplikację w trybie deweloperskim - FlashList wyświetla ostrzeżenie z sugestią wartości. Zmierz też rzeczywisty rozmiar przez onLayout.
React.memo nie zapobiega re-renderom	Funkcja przekazana jako prop ma nową referencję przy każdym renderze rodzica. Owin ją w useCallback w rodzicu. Dla obiektów: sprawdź czy nie tworzysz nowego obiektu inline: weather={{ city }} tworzy nowy obiekt za każdym razem.
TalkBack czyta elementy w złej kolejności	Kolejność czytania zależy od pozycji na ekranie, nie od kolejności w JSX. Na Androidzie użyj importantForAccessibility='yes' z accessibilityViewIsModal={true} dla dialogów. Dla niestandardowej kolejności: accessibilityTraversalBefore i accessibilityTraversalAfter (iOS) lub accessibilityLiveRegion.

Komendy diagnostyczne

Weryfikacja konfiguracji Detox

```
detox doctor      # Sprawdza zainstalowane narzędzia
detox list-configs # Wyświetla dostępne konfiguracje z .detoxrc.js
emulator -list-avds # Lista dostępnych AVD (użyj nazwy w avdName)
```

Sprawdzenie kompatybilności React Native Web

```
npx expo start --web # Uruchom web i sprawdź konsolę przeglądarki
npx expo export --platform web --dump-assetmap # Szczegóły bundlowania
```

Diagnostyka Turborepo

```
npx turbo run build --dry-run # Pokaż co by było zbudowane (bez budowania)
npx turbo run build --graph   # Wygeneruj graf zależności (wymaga graphviz)
npx turbo daemon clean        # Wyczyść cache demona Turborepo
```

Pomiar rozmiaru elementów FlashList

```
# Dodaj tymczasowo onLayout do elementu renderItem:
onLayout={({e}) => console.log('height:', e.nativeEvent.layout.height)}
```

Profilowanie rerenderow (why-did-you-render)

WeatherCard.whyDidYouRender = true; // Po wlaczeniu biblioteki

Weryfikacja typow w calym monorepo

npx turbo run type-check # Wymaga skryptu 'type-check' w package.json pakietow

Lub bezposrednio:

cd apps/mobile && npx tsc --noEmit

cd packages/shared && npx tsc --noEmit

Instrukcja Laboratoryjna React Native 4 - Detox, React Native Web i Monorepo z Turborepo

Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki

Nastepne cwiczenie: React Native 5 - Offline First, GraphQL, Push Notifications

YOLO dalej?

SLAM na telefonie?

POLITECHNIKA RZESZOWSKA

KATEDRA INFORMATYKI I AUTOMATYKI

DR INZ. MATEUSZ POMIANEK

CWICZENIE LABORATORYJNE REACT NATIVE 5

Przetwarzanie obrazu w czasie rzeczywistym

MediaPipe Holistic · MediaPipe Iris · React Native Skia · Vision Camera

Programowanie Aplikacji Mobilnych - modul React Native

Projekt: PoseTracker - detekcja i sledzenie ciała, twarzy, rak i wzroku w czasie rzeczywistym

Wymagania wstępne: ukończony React Native Lab 4 (Detox, RN Web, Monorepo). Znajomość podstaw algebry wektorowej (iloczyn skalarny, atan2).

Spis treści

1. Wizja komputerowa na urządzeniach mobilnych

Przetwarzanie obrazu w czasie rzeczywistym na smartfonie było jeszcze dekadę temu domeną specjalistycznych procesorów sygnałowych. Współczesne urządzenia mobilne posiadają dedykowane jednostki przetwarzania neuronowego (NPU) i akceleratory sprzętowe, które pozwalają uruchamiać modele uczenia maszynowego z wydajnością nieosiągalną dotąd poza serwerownią. MediaPipe, biblioteka Google, projektowana jest od podstaw z myślą o tej architekturze.

Celem tego ćwiczenia jest zbudowanie modułu PoseTracker: lekkiej, zoptymalizowanej aplikacji React Native, która w czasie rzeczywistym przetwarza obraz z kamery urządzenia, wykrywa ciało, twarz, ręce i oczy użytkownika oraz nakłada na podgląd interaktywny szkielet z parametrami liczbowymi. Wymagana wydajność to minimum 24 klatki na sekundę na średniej klasy smartfonie z Androidem.

Analogia na przykładzie kinezylogii z cyfrowym asystentem

Kinezylog analizujący ruch sportowca musi jednocześnie obserwować dziesiątki punktów ciała, mierzyć kąty w stawach i oceniać czas reakcji. PoseTracker to jego cyfrowy asystent: obserwuje 33 punkty ciała, 21 punktów każdej dłoni, 468 punktów siatki twarzy i 5 punktów każdej tętnówki - wszystko w czasie 33 milisekund przypadających na jedną klatkę.

1.1. Przegląd bibliotek do wizji komputerowej w React Native

Biblioteka	Podejscie	Wydajność	Trudność konfiguracji
MediaPipe Tasks (przez react-native-mediapipe)	Natywne modele MediaPipe przez JSI bridge. Dostęp do GPU/NPU.	Wysoka - 30+ fps	Srednia
TensorFlow.js + RN (tfjs-react-native)	Modele TF.js uruchamiane na urządzeniu. Wsparcie dla GPU przez expo-gl.	Srednia - 15-25 fps	Srednia
Google ML Kit (react-native-mlkit)	Natywny ML Kit przez modul natywny. Ograniczone API w porównaniu z MediaPipe.	Wysoka - natywna	Niska
OpenCV (react-native-opencv3)	Pełne OpenCV przez natywny modul. Ogromne możliwości, duży rozmiar biblioteki.	Wysoka	Wysoka
WebView + MediaPipe JS	MediaPipe uruchamiany w WebView przez przeglądarkowy JS. Wolny bridge dla danych wideo.	Niska - 5-15 fps	Niska

1.2. Stos technologiczny ćwiczenia

Komponent	Biblioteka / Technologia	Rola w module PoseTracker
Dostęp do kamery	react-native-vision-camera v4	Przechwytywanie klatek wideo w rozdzielczości i częstotliwości odpowiedniej do analizy.
Przetwarzanie klatek	VisionCamera Frame Processors + Worklets	Uruchamianie kodu analizy na wątku UI bez blokowania JS thread. Synchronizacja z MediaPipe.
Detekcja pozy i twarzy	react-native-mediapipe (Holistic)	Detekcja 33 punktów ciała, 468 punktów twarzy, 21 punktów każdej dłoni.
Sledzenie wzroku	react-native-mediapipe (Iris)	Detekcja 5 punktów tęczówki każdego oka. Obliczanie kierunku patrzenia.
Renderowanie szkieletu	@shopify/react-native-skia	Rysowanie linii, okręgów i tekstu bezpośrednio na płótnie z akceleracją GPU.
Współdzielenie danych	Reanimated SharedValues	Przekazywanie punktów landmarków z wątku UI (Frame Processor) do wątku rysowania (Skia).

2. Wydajny dostęp do klatek dzięki react-native-vision-camera

react-native-vision-camera (VisionCamera) to najwydajniejsza biblioteka do obsługi kamery w React Native. Jej kluczowa cecha dla naszego zastosowania jest Frame Processor API: możliwość uruchomienia synchronicznego kodu JavaScript (lub WorkLetu Reanimated) dla każdej klatki wideo,

bezpośrednio na watku UI, bez przejścia przez główny wątek JS. Eliminuje to opóźnienia komunikacji i pozwala osiągnąć wydajność równą natywnym rozwiązaniom.

Frame Processor - wątek UI kontra wątek JS

Zwykły JavaScript w React Native działa na oddzielnym wątku JS. Przesyłanie każdej klatki wideo (kilka megabajtów) z wątku UI do wątku JS i z powrotem wprowadzałoby opóźnienie uniemożliwiające pracę w czasie rzeczywistym. Frame Processor Worklet działa bezpośrednio na wątku UI: klatka nigdy nie opuszcza pamięci GPU. Wynik analizy (puncty landmarks) jest małym obiektem JSON - dopiero on jest przesyłany do wątku JS.

2.1. Instalacja i konfiguracja

```
# Instalacja wszystkich wymaganych bibliotek modułu PoseTracker:
npm install react-native-vision-camera
npm install react-native-worklets-core      # Silnik WorkLetow dla VisionCamera
npm install react-native-mediapipe         # Natywne modele MediaPipe
npm install @shopify/react-native-skia     # Renderowanie GPU
npm install react-native-reanimated        # SharedValues (już z Lab 2)

# Przebuduj projekt natywny (wszystkie te biblioteki wymagają native build):
npx expo prebuild --clean
npx expo run:android

# Konfiguracja Babel (babel.config.js) - kolejność pluginów ma znaczenie!
module.exports = {
  presets: ['babel-preset-expo'],
  plugins: [
    'react-native-worklets-core/plugin',    // 1. WorkLety VisionCamera
    'react-native-reanimated/plugin',       // 2. Reanimated (zawsze ostatni!)
  ],
};
```

2.2. Uprawnienia i konfiguracja kamery

```
// app.config.ts - uprawnienia kamery
export default ({ config }: ConfigContext): ExpoConfig => ({
  ...config,
  plugins: [
    [
      'react-native-vision-camera',
      {
        cameraPermissionText: 'PoseTracker potrzebuje dostępu do kamery',
        enableMicrophonePermission: false, // Tylko kamera, bez mikrofonu
      },
    ],
    ['expo-detox-plugin', {}],
  ],
});

// hooks/useCameraPermission.ts - obsługa uprawnień
import { Camera } from 'react-native-vision-camera';

export function useCameraPermission() {
  const [hasPermission, setHasPermission] = useState<boolean | null>(null);

  useEffect(() => {
    Camera.requestCameraPermission().then(status => {
```

```
    setHasPermission(status === 'granted');
  });
}, []);

return hasPermission;
}
```

2.3. Konfiguracja strumienia wideo dla analizy

```
// components/CameraView.tsx - konfiguracja kamery do analizy w czasie rzeczywistym
import { Camera, useCameraDevice, useCameraFormat } from 'react-native-vision-camera';

export function PoseTrackerCamera() {
  // Kamera przednia (front) do sledzenia twarzy; tylna (back) do pozy ciała
  const device = useCameraDevice('front');

  // Wybierz format: 720p dla balansu jakosci i wydajnosci
  // MediaPipe wewnetrznie przeskalowuje do 256x256, wiec 4K to zbędny narzut
  const format = useCameraFormat(device, [
    { videoResolution: { width: 1280, height: 720 } },
    { fps: 30 }, // 30 klatek/s jako cel
  ]);

  if (!device) return <Text>Brak kamery</Text>;

  return (
    <Camera
      device={device}
      format={format}
      isActive={true}
      frameProcessor={frameProcessor} // Omowiony w sekcji 8
      fps={30}
      // pixelFormat musi pasowac do oczekiwania MediaPipe:
      pixelFormat='yuv' // YUV jest wydajniejszy niz 'rgb' dla analizy
      style={StyleSheet.absoluteFill}
    />
  );
}
```

3. MediaPipe - architektura i modele detekcji

MediaPipe to otwarty źródłowy framework Google do budowania potoków przetwarzania multimedialnego w czasie rzeczywistym. Jego kluczowa zaleta jest możliwość uruchamiania modeli uczenia maszynowego bezpośrednio na urządzeniu (on-device inference), bez przysyłania żadnych danych na serwer. Modele są zoptymalizowane dla mobilnych procesorów przez kwantyzację (float32 → int8) i przycinanie grafu obliczeniowego.

3.1. Modele wykrywania używane w PoseTracker

Model MediaPipe	Co wykrywa	Liczba punktów	Rozmiar modelu	Latencja (Pixel 7)
Pose Landmarker	Punkty ciała człowieka (poza głową): ramiona, łokcie,	33 punkty 3D	~8 MB (Heavy)	~15 ms

	nadgarstki, biodra, kolana, kostki.			
Face Landmarker	Siatka twarzy: kontur, brew, oczodoly, nos, usta, policzki.	468 punktów 3D	~10 MB	~12 ms
Hand Landmarker	Punkty dłoni i palców: nadgarstek, kostki, stawy, czubki palców.	21 punktów / dłoni	~4 MB	~8 ms
Holistic Landmarker	Połączenie Pose + Face + oba Hands w jednym przebiegu. Bardziej wydajne niż trzy oddzielne modele.	33+468+42	~12 MB	~25 ms
Iris Landmarker	Punkty tęczówki i rzenic obojga oczu. Umożliwia estymację kierunku patrzenia.	5 punktów / oko	~2 MB	~3 ms

3.2. Format wyników dla obiektów Landmark

Każdy model MediaPipe zwraca listę obiektów Landmark. Współrzędne są znormalizowane do zakresu [0, 1] względem rozmiaru obrazu wejściowego. Aby przeliczyć na piksele ekranu, należy pomnożyć x przez szerokość i y przez wysokość widoku kamery.

Pole obiektu Landmark	Typ	Opis
x	number	Współrzędna pozioma: 0.0 = lewy kraniec obrazu, 1.0 = prawy kraniec. Uwaga: dla kamery przedniej obraz jest lustrzany.
y	number	Współrzędna pionowa: 0.0 = górny kraniec, 1.0 = dolny kraniec obrazu.
z	number	Głębokość względna. Dla Pose: w jednostkach porównalnych z x (ciało w rozmiarach klatki). Dla Face i Hands: względna głębokość w ramach wykrytego obszaru.
visibility	number [0-1]	Prawdopodobieństwo, że punkt jest widoczny (nie zasłonięty). Dostępne tylko dla Pose Landmarker. Poniżej 0.5 punkt należy traktować jako nieokreślony.
presence	number [0-1]	Prawdopodobieństwo, że punkt istnieje w kadrze. Różni się od visibility: presence ocenia czy punkt jest w obrazie, visibility - czy widoczny.

```
// Typowy wynik detekcji z MediaPipe - przykład dla jednego landmarka:
interface Landmark {
  x: number; // 0.0 - 1.0 (znormalizowana pozycja pozioma)
  y: number; // 0.0 - 1.0 (znormalizowana pozycja pionowa)
  z: number; // Względna głębokość (ujemna = bliżej kamery)
  visibility?: number; // 0.0 - 1.0 (tylko Pose)
  presence?: number; // 0.0 - 1.0 (tylko Pose)
}

// Przeliczenie na współrzędne ekranu:
const toScreen = (lm: Landmark, width: number, height: number) => ({
  x: lm.x * width,
  y: lm.y * height,
});

// Ważne: dla kamery przedniej (selfie) obraz jest już lustrzany przez VisionCamera.
// Jeśli szkielet wychodzi po zły stronie, dodaj: x = (1 - lm.x) * width
```

4. MediaPipe Holistic - detekcja pozy, twarzy i rąk

4.1. Indeksy landmarków ciała (Pose Landmarker)

Pose Landmarker wykrywa 33 punkty ciała. Znajomość indeksów jest kluczowa do budowania połączeń szkieletu i obliczania kątów w stawach. Poniżej lista najważniejszych punktów z ich indeksami.

Indeks	Nazwa punktu	In-deks	Nazwa punktu	Indeks	Nazwa punktu
0	Nose (nosek)	11	Left shoulder	22	Right pinky
1	Left eye (inner)	12	Right shoulder	23	Left hip
2	Left eye	13	Left elbow	24	Right hip
4	Right eye	14	Right elbow	25	Left knee
7	Left ear	15	Left wrist	26	Right knee
8	Right ear	16	Right wrist	27	Left ankle
9	Mouth left	17	Left pinky	28	Right ankle
10	Mouth right	19	Left index	31	Left foot index
11	Left shoulder	21	Left thumb	32	Right foot index

4.2. Połączenia szkieletu ciała - POSE_CONNECTIONS

```
// constants/mediapipe.ts - definicje połączeń szkieletu

// Połączenia szkieletu ciała: [indeks_A, indeks_B]
// Linia jest rysowana między landmarkiem A a landmarkiem B
export const POSE_CONNECTIONS: [number, number][] = [
  // Twarz (uproszczona):
  [0, 1], [1, 2], [2, 3], [3, 7], // Nos - lewe oko - lewe ucho
  [0, 4], [4, 5], [5, 6], [6, 8], // Nos - prawe oko - prawe ucho
  [9, 10], // Usta
  // Tors:
  [11, 12], // Linia barków
```

```

    [11, 23], [12, 24], // Bark - biodro (boki ciała)
    [23, 24], // Linia bioder
    // Lewe ramie:
    [11, 13], [13, 15], // Bark - łokiec - nadgarstek
    [15, 17], [15, 19], [15, 21], // Nadgarstek - palce
    [17, 19], // Palce (zewnetrzna krawedz dłoni)
    // Prawe ramie:
    [12, 14], [14, 16],
    [16, 18], [16, 20], [16, 22],
    [18, 20],
    // Lewa noga:
    [23, 25], [25, 27], [27, 29], [27, 31], [29, 31],
    // Prawa noga:
    [24, 26], [26, 28], [28, 30], [28, 32], [30, 32],
  ];

  // Polaczenia dłoni: 21 punktow, 5 palcow + grzebien dłoni
  export const HAND_CONNECTIONS: [number, number][] = [
    // Kciuk (0=nadgarstek, 1=kostka, 2=staw, 3=staw, 4=czubek):
    [0, 1], [1, 2], [2, 3], [3, 4],
    // Wskazujacy: [0,5], [5,6], [6,7], [7,8]
    [0, 5], [5, 6], [6, 7], [7, 8],
    // Srodkowy: [0,9], [9,10], [10,11], [11,12]
    [0, 9], [9, 10], [10, 11], [11, 12],
    // Serdeczny: [0,13], [13,14], [14,15], [15,16]
    [0, 13], [13, 14], [14, 15], [15, 16],
    // Maly: [0,17], [17,18], [18,19], [19,20]
    [0, 17], [17, 18], [18, 19], [19, 20],
    // Grzebien dłoni:
    [5, 9], [9, 13], [13, 17],
  ];

  // Uprozczone polaczenia twarzy (kontur twarzy + brwi + usta):
  // Pelna siatka Face Mesh ma 468 punktow i kilka tysiecy trojkatow -
  // do celow wizualizacji uzywamy podzbioru konturowego.
  export const FACE_OUTLINE_CONNECTIONS: [number, number][] = [
    // Kontur twarzy (punkty 0-16 lewy+prawy):
    [10, 338], [338, 297], [297, 332], [332, 284],
    [284, 251], [251, 389], [389, 356], [356, 454],
    [454, 323], [323, 361], [361, 288], [288, 397],
    [397, 365], [365, 379], [379, 378], [378, 400],
    [400, 377], [377, 152], [152, 148], [148, 176],
    [176, 149], [149, 150], [150, 136], [136, 172],
    [172, 58], [58, 132], [132, 93], [93, 234],
    [234, 127], [127, 162], [162, 21], [21, 54],
    [54, 103], [103, 67], [67, 109], [109, 10],
  ];
};

```

4.3. Inicjalizacja Holistic Landmarker

```

// services/mediapipe/holisticService.ts
import {
  HolisticLandmarker,
  HolisticLandmarkerResult,
  FilesetResolver,
} from 'react-native-mediapipe';

let holisticLandmarker: HolisticLandmarker | null = null;

export async function initHolistic(): Promise<void> {
  // FilesetResolver pobiera pliki WASM/modeli - jednorazowe ladowanie przy starcie
  const vision = await FilesetResolver.forVisionTasks(
    'https://cdn.jsdelivr.net/npm/@mediapipe/tasks-vision/wasm'
  );
};

```

```
// Dla produkcji: dołącz pliki WASM do assets i użyj lokalnych URI

holisticLandmarker = await HolisticLandmarker.createFromOptions(vision, {
  baseOptions: {
    modelAssetPath:
      'https://storage.googleapis.com/mediapipe-models/holistic_landmarker/
holistic_landmarker/float16/latest/holistic_landmarker.task',
    // Dla produkcji: skopiuj plik .task do assets/ i podaj lokalny URI
    delegate: 'GPU', // Użyj GPU/NPU zamiast CPU - 3-5x szybsze
  },
  runningMode: 'VIDEO', // VIDEO = optymalizacja dla strumienia klatek
  // 'IMAGE' = jednorazowa detekcja; 'VIDEO' = ciągły strumień
  minPoseDetectionConfidence: 0.5, // Prog pewności detekcji powyżej
  minPosePresenceConfidence: 0.5, // Prog obecności w kadrze
  minTrackingConfidence: 0.5, // Prog śledzenia między klatkami
  minFaceDetectionConfidence: 0.5,
  minFacePresenceConfidence: 0.5,
  minHandDetectionConfidence: 0.5,
  minHandPresenceConfidence: 0.5,
  outputFaceBlendshapes: false, // Blendshapes (ARKIT wyraży) - nie potrzebujemy
  outputSegmentationMasks: false, // Maski segmentacji - nie potrzebujemy
});

export function detectHolistic(
  frame: any,
  timestampMs: number
): HolisticLandmarkerResult | null {
  if (!holisticLandmarker) return null;
  // detectForVideo wymaga monotonicznie rosnącego timestamp (ms od uruchomienia)
  return holisticLandmarker.detectForVideo(frame, timestampMs);
}
```

5. MediaPipe Iris - śledzenie wzroku i punkty tęczówki

MediaPipe Iris to model uzupełniający Face Landmarker o dokładne śledzenie tęczówki. Wykrywa 5 punktów na oku: centrum tęczówki oraz 4 punkty jej krawędzi. Na podstawie tych punktów można estymować kierunek patrzenia i rozmiar tęczówki (przydatny do szacowania odległości od kamery).

5.1. Punkty Iris Landmarker

Indeks	Nazwa punktu	Zastosowanie
0	Centrum tęczówki (iris center)	Główny punkt śledzenia wzroku. Pozycja centro tęczówki względem oczodołu.
1	Górna krawędź tęczówki (top)	Średnica pionowa tęczówki. Różnica między punktem 1, a 3 to pełna średnica tęczówki.
2	Prawa krawędź tęczówki (right)	Średnica pozioma tęczówki. Różnica między punktem 2, a 4 to szerokość tęczówki.
3	Dolna krawędź tęczówki (bottom)	Razem z punktem 1 definiuje oś pionową.
4	Lewa krawędź tęczówki (left)	Razem z punktem 2 definiuje oś poziomą.

5.2. Estymacja kierunku patrzenia

Kierunek wzroku (gaze direction) oblicza się przez porównanie pozycji centrum teczowki względem środka oczodołu. Oczodoł definiują punkty Face Mesh: okolice indeksów 33 (lewy kąt oka) i 133 (prawy kąt oka) dla lewego oka, analogicznie dla prawego.

```
// services/mediapipe/irisService.ts
import { IrisLandmarker, FilesetResolver } from 'react-native-mediapipe';

let irisLandmarker: IrisLandmarker | null = null;

export async function initIris(): Promise<void> {
  const vision = await FilesetResolver.forVisionTasks(
    'https://cdn.jsdelivr.net/npm/@mediapipe/tasks-vision/wasm'
  );

  irisLandmarker = await IrisLandmarker.createFromOptions(vision, {
    baseOptions: {
      modelAssetPath:
        'https://storage.googleapis.com/mediapipe-models/face_landmarker/face_landmarker/float16/latest/face_landmarker.task',
      delegate: 'GPU',
    },
    runningMode: 'VIDEO',
    outputIrisLandmarks: true, // Kluczowe: włączenie detekcji teczowki
    numFaces: 1,              // Śledzenie jednej twarzy - bardziej wydajne
  });
}

// Obliczenie wektora kierunku wzroku dla jednego oka
export interface GazeVector {
  x: number; // -1.0 = skrajnie lewo, 0 = prosto, +1.0 = skrajnie prawo
  y: number; // -1.0 = skrajnie góra, 0 = prosto, +1.0 = skrajnie dół
  irisRadiusPx: number; // Promień teczowki w pikselach (do estymacji odległości)
}

export function computeGazeVector(
  irisLandmarks: Landmark[], // 5 punktów teczowki
  eyeCornerLeft: Landmark,    // Lewy kąt oczodołu (Face Mesh pkt 33 lub 362)
  eyeCornerRight: Landmark,   // Prawy kąt oczodołu (Face Mesh pkt 133 lub 263)
  frameWidth: number,
  frameHeight: number,
): GazeVector {
  const center = irisLandmarks[0]; // Centrum teczowki
  const top = irisLandmarks[1];
  const right = irisLandmarks[2];

  // Promień teczowki: średnia z osi pionowej i poziomej (pikselach ekranu)
  const radiusH = Math.abs(right.x - center.x) * frameWidth;
  const radiusV = Math.abs(top.y - center.y) * frameHeight;
  const irisRadiusPx = (radiusH + radiusV) / 2;

  // Środek oczodołu - średnia lewego i prawego kąta
  const eyeCenterX = (eyeCornerLeft.x + eyeCornerRight.x) / 2;
  const eyeCenterY = (eyeCornerLeft.y + eyeCornerRight.y) / 2;

  // Szerokość oczodołu jako jednostka normalizacji
  const eyeWidth = Math.abs(eyeCornerRight.x - eyeCornerLeft.x);

  // Wektor patrzenia: odchylenie centrum teczowki od centrum oczodołu
  // Normalizacja przez szerokość oczodołu: wynik w zakresie ~[-1, 1]
  const gazeX = (center.x - eyeCenterX) / (eyeWidth * 0.5);
```

```
const gazeY = (center.y - eyeCenterY) / (eyewidth * 0.3);

return {
  x: Math.max(-1, Math.min(1, gazeX)),
  y: Math.max(-1, Math.min(1, gazeY)),
  irisRadiusPx,
};
}
```

Ciekawostka: Średnica tęczówki a odległość:

Tęczówka dorosłego człowieka ma średnicę ~11-12 mm. Znając rzeczywisty rozmiar tęczówki i zmierzoną średnicę w pikselach, można obliczyć odległość od kamery ze wzoru:

$\text{odległość} = (\text{ogniskowaKamery} * \text{średnicaRealnejŻrenicy}) / \text{średnicaŻrenicyPixelach}$.

Ogniskowa (focal) kamer mobilnych to typowo 500-1200 pikseli dla 720p.

6. Renderowanie szkieletów „live” przez React Native Skia

React Native Skia (@shopify/react-native-skia) to biblioteka renderowania grafiki 2D oparta na silniku Skia (tym samym, którego używa Chrome i Flutter). Pozwala rysować na płotnie (Canvas) bezpośrednio z wątku UI przy pełnej akceleracji GPU. Jest to kluczowe dla zachowania 30+ fps podczas równoległego przetwarzania kamer przez MediaPipe.

Analogia: rysownik i asystent obliczeniowy.

Wyobraź sobie artystę rysującego szkielet postaci w czasie rzeczywistym. MediaPipe to asystent który mierzy pozycje stawów i krzyczy liczby przez pomieszczenie. Rysownik (Skia) słysząc nowe współrzędne natychmiast rysuje nowe linie nie czekając na potwierdzenie. Obaj pracują równocześnie na tym samym wątku (UI thread), bez przerywania sobie nawzajem.

6.1. Komponent plotna Skia

```
// components/SkeletonOverlay.tsx - płotno nakładane na podgląd kamery
import { Canvas, Line, Circle, Text as SkiaText, useFont,
  Group, Paint } from '@shopify/react-native-skia';
import { useSharedValue, useDerivedValue } from 'react-native-reanimated';

interface SkeletonOverlayProps {
  poseLandmarks: SharedValue<Landmark[]>;
  leftHandLMs: SharedValue<Landmark[]>;
  rightHandLMs: SharedValue<Landmark[]>;
  faceLandmarks: SharedValue<Landmark[]>;
  irisLeft: SharedValue<Landmark[]>;
  irisRight: SharedValue<Landmark[]>;
  frameWidth: number;
  frameHeight: number;
}

export function SkeletonOverlay({
  poseLandmarks, leftHandLMs, rightHandLMs,
  faceLandmarks, irisLeft, irisRight,
  frameWidth, frameHeight
}: SkeletonOverlayProps) {

  // Skia czyta SharedValues bezpośrednio na wątku UI - brak kopiowania danych!
  return (
    <Canvas style={StyleSheet.absoluteFill}>
```

```

    {/* Szkielet ciala */}
    <PoseSkeleton
      landmarks={poseLandmarks}
      connections={POSE_CONNECTIONS}
      fw={frameWidth} fh={frameHeight}
    />
    {/* Dlonie */}
    <HandSkeleton
      landmarks={leftHandLMs}
      color='#00FF88'
      fw={frameWidth} fh={frameHeight}
    />
    <HandSkeleton
      landmarks={rightHandLMs}
      color='#FF8800'
      fw={frameWidth} fh={frameHeight}
    />
    {/* Kontur twarzy */}
    <FaceOutline
      landmarks={faceLandmarks}
      fw={frameWidth} fh={frameHeight}
    />
    {/* Teczowki */}
    <IrisOverlay
      left={irisLeft} right={irisRight}
      fw={frameWidth} fh={frameHeight}
    />
  </Canvas>
);
}

```

6.2. Rysowanie linii polaczen szkieletu

```

// components/skeleton/PoseSkeleton.tsx
import { Line, Group } from '@shopify/react-native-skia';
import { useDerivedValue, SharedValue } from 'react-native-reanimated';

// Kolory per czesc ciala (styl MediaPipe Studio):
const COLORS = {
  leftSide: '#FF3B30', // Czerwony - lewa strona ciala
  rightSide: '#007AFF', // Niebieski - prawa strona ciala
  centerLine: '#FFFFFF', // Bialy - os centralna (tors, kregoslup)
  landmark: '#FFD60A', // Zolty - punkty (okragle markery)
};

// Definicja ktore polaczenia naleza do ktorej strony:
const LEFT_CONNECTIONS = new Set([[11,13],[13,15],[11,23],[23,25],[25,27]].map(JSON.stringify));
const RIGHT_CONNECTIONS = new Set([[12,14],[14,16],[12,24],[24,26],[26,28]].map(JSON.stringify));

interface PoseSkeletonProps {
  landmarks: SharedValue<Landmark[]>;
  connections: [number, number][];
  fw: number; fh: number;
}

export function PoseSkeleton({ landmarks, connections, fw, fh }: PoseSkeletonProps) {
  // useDerivedValue - przelicza dane gdy landmarks sie zmienia, na watku UI
  const lines = useDerivedValue(() => {
    const lms = landmarks.value;
    if (!lms || lms.length === 0) return [];

    return connections

```

```

    .filter(([a, b]) => {
      // Pomin połączenia z landmarkami o niskiej pewności
      const va = lms[a]?.visibility ?? 1;
      const vb = lms[b]?.visibility ?? 1;
      return va > 0.5 && vb > 0.5;
    })
    .map(([a, b]) => {
      const key = JSON.stringify([a, b]);
      const color = LEFT_CONNECTIONS.has(key) ? COLORS.leftSide
        : RIGHT_CONNECTIONS.has(key) ? COLORS.rightSide
        : COLORS.centerLine;

      return {
        x1: lms[a].x * fw, y1: lms[a].y * fh,
        x2: lms[b].x * fw, y2: lms[b].y * fh,
        color,
      };
    });
  });

  // Skia renderuje tablice linii bezpośrednio na GPU
  return (
    <Group>
      {lines.value.map((line, i) => (
        <Line key={i}>
          p1={{ x: line.x1, y: line.y1 }}
          p2={{ x: line.x2, y: line.y2 }}
          strokeWidth={2}
          color={line.color}
          strokeCap='round'
        </>
      ))}
    </Group>
  );
}

```

6.3. Rysowanie teczówki - okrag z promieniem

```

// components/skeleton/IrisOverlay.tsx
import { Circle, Group } from '@shopify/react-native-skia';

export function IrisOverlay({ left, right, fw, fh }) {
  const drawIris = (irisLMs: Landmark[], color: string) => {
    if (!irisLMs || irisLMs.length < 5) return null;

    const cx = irisLMs[0].x * fw;
    const cy = irisLMs[0].y * fh;

    // Oblicz promień jako średnia odległości do krawędzi teczówki:
    const r1 = Math.abs(irisLMs[1].y - irisLMs[0].y) * fh; // os pionowa
    const r2 = Math.abs(irisLMs[2].x - irisLMs[0].x) * fw; // os pozioma
    const radius = (r1 + r2) / 2;

    return (
      <Group>
        {/* Zewnętrzny okrag - obwód teczówki */}
        <Circle cx={cx} cy={cy} r={radius}>
          color='transparent'
          style='stroke'
          strokeWidth={2}
          strokeColor={color}
        </>
        {/* Wewnętrzny punkt - centrum teczówki (zrenica) */}
        <Circle cx={cx} cy={cy} r={3}>
          color={color}
        </>
      </Group>
    );
  };
}

```

```

        style='fill'
      />
    </Group>
  );
};

return (
  <Group>
    {drawIris(left.value, '#00FFFF')} { /* Cyan - lewe oko */}
    {drawIris(right.value, '#FF00FF')} { /* Magenta - prawe oko */}
  </Group>
);
}

```

7. Obliczanie parametrów liczbowych - kąty i odległości

Sama wizualizacja szkieletu to punkt wyjścia. Wartość analityczna modułu tkwi w parametrach liczbowych: kątach w stawach, prędkości ruchu, symetrii ciała i wektorze wzroku. Parametry te są wyświetlane na podglądzie i mogą być eksportowane do analizy.

7.1. Obliczanie kąta w stawie - wzór z atan2

Kąt w stawie oblicza się między trzema punktami: końcem, środkiem (stawem) i drugim końcem. Funkcja `atan2` zwraca kąt między wektorem a osią X, więc obliczamy dwa wektory ze stawu i obliczamy kąt między nimi.

```

// utils/angles.ts - obliczenia katow

/**
 * Oblicza kąt w stawie między trzema punktami (w stopniach).
 * @param a - punkt A (np. ramie)
 * @param b - punkt B (np. łokiec - STAW - tu mierzony kąt)
 * @param c - punkt C (np. nadgarstek)
 * @returns Kąt w stopniach [0, 180]
 */
export function calculateJointAngle(
  a: { x: number; y: number },
  b: { x: number; y: number },
  c: { x: number; y: number },
): number {
  // Wektor BA (od stawu do punktu A):
  const vecBAx = a.x - b.x;
  const vecBAy = a.y - b.y;
  // Wektor BC (od stawu do punktu C):
  const vecBCx = c.x - b.x;
  const vecBCy = c.y - b.y;

  // Kąt między wektorami przez iloczyn skalarny:
  // cos(theta) = (BA · BC) / (|BA| * |BC|)
  const dot = vecBAx * vecBCx + vecBAy * vecBCy;
  const magBA = Math.sqrt(vecBAx ** 2 + vecBAy ** 2);
  const magBC = Math.sqrt(vecBCx ** 2 + vecBCy ** 2);

  if (magBA < 1e-6 || magBC < 1e-6) return 0; // Zabezpieczenie przed dzieleniem przez 0

  // Clamp do [-1, 1] ze względu na błędy zmiennoprzecinkowe:
  const cosTheta = Math.max(-1, Math.min(1, dot / (magBA * magBC)));
  const angleRad = Math.acos(cosTheta);
  return (angleRad * 180) / Math.PI; // Konwersja na stopnie
}

```

```

}

// Przykładowe użycie - kat lewego łokcia:
// a = poseLandmarks[11] (lewe ramię)
// b = poseLandmarks[13] (lewy łokiec)
// c = poseLandmarks[15] (lewy nadgarstek)
const leftElbowAngle = calculateJointAngle(
  { x: landmarks[11].x, y: landmarks[11].y },
  { x: landmarks[13].x, y: landmarks[13].y },
  { x: landmarks[15].x, y: landmarks[15].y },
);
// Wynik: np. 167.3 (prawie wyprostowane ramię) lub 45.2 (mocno zgięty łokiec)

```

7.2. Katalog obliczanych kątów stawów

Nazwa kąta	Punkt A (indeks)	Staw B (indeks)	Punkt C (indeks)	Zakres zdrowy
Lewy łokiec	Left shoulder (11)	Left elbow (13)	Left wrist (15)	0-180°
Prawy łokiec	Right shoulder (12)	Right elbow (14)	Right wrist (16)	0-180°
Lewe kolano	Left hip (23)	Left knee (25)	Left ankle (27)	0-180°
Prawe kolano	Right hip (24)	Right knee (26)	Right ankle (28)	0-180°
Lewe biodro	Left shoulder (11)	Left hip (23)	Left knee (25)	0-180°
Prawe biodro	Right shoulder (12)	Right hip (24)	Right knee (26)	0-180°
Lewy bark	Left elbow (13)	Left shoulder (11)	Left hip (23)	0-180°
Prawy bark	Right elbow (14)	Right shoulder (12)	Right hip (24)	0-180°
Nachylenie głowy	Left shoulder (11)	Nose (0)	Right shoulder (12)	75-105°

7.3. Pomiar odległości między punktami

```

// utils/angles.ts (cd.) - obliczenia odległości

/**
 * Normalizowana odległość między dwoma landmarkami.
 * Normalizacja względem szerokości barków eliminuje wpływ odległości od kamery.
 * @returns Odległość jako wielokrotność szerokości barków [0.0 - ~2.0]
 */
export function normalizedDistance(
  a: Landmark,
  b: Landmark,
  leftShoulder: Landmark, // Punkt referencyjny normalizacji
  rightShoulder: Landmark,
): number {
  const dx = (a.x - b.x);
  const dy = (a.y - b.y);
  const dist = Math.sqrt(dx * dx + dy * dy);
}

```

```
// Szerokosc barkow jako jednostka referencyjna:
const sdx = leftShoulder.x - rightShoulder.x;
const sdy = leftShoulder.y - rightShoulder.y;
const shoulderWidth = Math.sqrt(sdx * sdx + sdy * sdy);

if (shoulderWidth < 1e-6) return 0;
return dist / shoulderWidth;
}

// Wymagana odleglosc dloni od twarzy (np. do rozpoznawania gestow):
const leftHandToFaceDistance = normalizedDistance(
  poseLandmarks[15], // Lewy nadgarstek
  poseLandmarks[0], // Nos (centrum twarzy)
  poseLandmarks[11],
  poseLandmarks[12],
);
// Wynik < 0.3: dloni blisko twarzy; > 1.0: dloni daleko od twarzy
```

8. Potok przetwarzania klatek - architektura modulu

Serce modulu PoseTracker to potok przetwarzania klatek (Frame Processing Pipeline). Łączył VisionCamera (klatki), MediaPipe (analiza) i Skia (rendering), w sposób niepowodujący zakładek i zachowujący płynność 30+ fps. Kluczem jest praca wyłącznie na wątku UI, bez przejść przez główny wątek JavaScript.

8.1. Schemat przepływu danych

Krok	Co sie dzieje	Watek	Czas (cel)	Technologia
1	Kamera przechwytuje klatke wideo (1280x720, YUV)	UI thread	0 ms	VisionCamera
2	Frame Processor Worklet uruchamia sie synchronicznie dla kazdej klatki	UI thread	< 1 ms	VisionCamera Worklet
3	MediaPipe analizuje klatke: Holistic (cialo+twarz+rece) + Iris (teczowki)	GPU/NPU	< 25 ms	react-native-mediapipe
4	Wyniki (tablice Landmark) sa zapisywane do SharedValues Reanimated	UI thread	< 1 ms	Reanimated
5	Skia odczytuje SharedValues i renderuje szkielec + tekst na GPU	UI thread (GPU)	< 5 ms	React Native Skia
6	Gotowa klatka z nakladka jest wyswietlana uzytkownikowi	Display	16-33 ms	System OS

8.2. Implementacja Frame Processor

```
// hooks/useFrameProcessor.ts - serce modulu PoseTracker
import { useFrameProcessor } from 'react-native-vision-camera';
import { useSharedValue, runOnJS } from 'react-native-reanimated';
import { detectHolistic } from '../services/mediapipe/holisticService';
import { detectIris, computeGazeVector } from '../services/mediapipe/irisService';
import { calculateJointAngle } from '../utils/angles';

export function usePoseFrameProcessor(frameWidth: number, frameHeight: number) {
  // SharedValues - dane przeplywaja miedzy Frame Processor (UI) a Skia (UI)
```

```

const poseLandmarks = useSharedValue<Landmark[]>([]);
const leftHandLMs = useSharedValue<Landmark[]>([]);
const rightHandLMs = useSharedValue<Landmark[]>([]);
const faceLandmarks = useSharedValue<Landmark[]>([]);
const irisLeft = useSharedValue<Landmark[]>([]);
const irisRight = useSharedValue<Landmark[]>([]);
const jointAngles = useSharedValue<Record<string, number>>({});
const gazeVector = useSharedValue<{ lx: number; ly: number; rx: number; ry:
number }>
    ({ lx: 0, ly: 0, rx: 0, ry: 0 });
const fps = useSharedValue<number>(0);

// Pomiar FPS:
let lastFrameTime = 0;

// Definicja Frame Processora - uruchamia sie dla kazdej klatki na watku UI
// 'worklet' - oznacza ze funkcja moze byc uruchomiona poza watkiem JS
const frameProcessor = useFrameProcessor((frame) => {
  'worklet';

  const now = Date.now();
  const delta = now - lastFrameTime;
  if (delta > 0) fps.value = Math.round(1000 / delta);
  lastFrameTime = now;

  // Detekcja Holistic (cialo + twarz + rece):
  const holisticResult = detectHolistic(frame, now);

  if (holisticResult) {
    const pose = holisticResult.poseLandmarks?.[0] ?? [];
    const face = holisticResult.faceLandmarks?.[0] ?? [];
    const leftHand = holisticResult.leftHandLandmarks?.[0] ?? [];
    const rightHand = holisticResult.rightHandLandmarks?.[0] ?? [];

    // Aktualizacja SharedValues (bez kopii - bezposrednie przypisanie):
    poseLandmarks.value = pose;
    faceLandmarks.value = face;
    leftHandLMs.value = leftHand;
    rightHandLMs.value = rightHand;

    // Obliczenie katow stawow (tylko gdy pose wykryta):
    if (pose.length >= 29) {
      jointAngles.value = {
        leftElbow: calculateJointAngle(pose[11], pose[13], pose[15]),
        rightElbow: calculateJointAngle(pose[12], pose[14], pose[16]),
        leftKnee: calculateJointAngle(pose[23], pose[25], pose[27]),
        rightKnee: calculateJointAngle(pose[24], pose[26], pose[28]),
        leftHip: calculateJointAngle(pose[11], pose[23], pose[25]),
        rightHip: calculateJointAngle(pose[12], pose[24], pose[26]),
      };
    }
  }

  // Detekcja Iris (teczowka) z wynikow Face Landmarker:
  if (face.length >= 478) { // 468 Face + 10 Iris (5 per oko)
    // Punkty Iris sa dolaczone na koniec tablicy Face Landmarks:
    const leftIrisLMs = face.slice(468, 473); // 5 punktow lewego oka
    const rightIrisLMs = face.slice(473, 478); // 5 punktow prawego oka
    irisLeft.value = leftIrisLMs;
    irisRight.value = rightIrisLMs;

    // Obliczenie wektora patrzenia (katy oczodolu z Face Mesh):
    const leftGaze = computeGazeVector(
      leftIrisLMs, face[362], face[263], frameWidth, frameHeight
    );
  }
});

```

```

    const rightGaze = computeGazeVector(
      rightIrisLMs, face[33], face[133], frameWidth, frameHeight
    );
    gazeVector.value = {
      lx: leftGaze.x, ly: leftGaze.y,
      rx: rightGaze.x, ry: rightGaze.y,
    };
  }
}
}, []));

return {
  frameProcessor,
  poseLandmarks, leftHandLMs, rightHandLMs,
  faceLandmarks, irisLeft, irisRight,
  jointAngles, gazeVector, fps,
};
}

```

9. Wyświetlanie parametrów liczbowych na podglądzie

Parametry liczbowe (katy, pozycje, ID) są wyświetlane bezpośrednio na podglądzie kamery w postaci tekstów Skia. Użycie Skia do renderowania tekstu (zamiast standardowych komponentów React Native Text) jest kluczowe: tekst i szkielet rysowane są w tym samym przebiegu na GPU, bez przełączania warstw.

9.1. Panel parametrów z boku ekranu

```

// components/ParametersPanel.tsx - panel liczbowy na tle półprzezroczystym
import { Canvas, Rect, Text as SKText, useFont, Group } from '@shopify/react-native-skia';
import { useDerivedValue, SharedValue } from 'react-native-reanimated';

interface ParametersPanelProps {
  jointAngles: SharedValue<Record<string, number>>;
  gazeVector: SharedValue<{ lx: number; ly: number; rx: number; ry: number }>;
  fps: SharedValue<number>;
  poseLandmarks: SharedValue<Landmark[]>;
  width: number;
  height: number;
}

export function ParametersPanel({
  jointAngles, gazeVector, fps, poseLandmarks, width, height
}: ParametersPanelProps) {
  // Załaduj font dla Skia (musi być załadowany asynchronicznie):
  const font = useFont(require('../assets/fonts/RobotoMono-Regular.ttf'), 12);

  // Dane panelu obliczane na watku UI przez useDerivedValue:
  const panelLines = useDerivedValue(() => {
    const angles = jointAngles.value;
    const gaze = gazeVector.value;
    const pose = poseLandmarks.value;

    const lines: string[] = [];

    lines.push(`FPS: ${fps.value}`);
    lines.push(`--- KATY STAWOW ---`);
    lines.push(`Lokiec L:  ${Math.round(angles.leftElbow ?? 0)}°`);
    lines.push(`Lokiec P:  ${Math.round(angles.rightElbow ?? 0)}°`);
    lines.push(`Kolano L:  ${Math.round(angles.leftKnee ?? 0)}°`);
  });
}

```

```

    lines.push(`Kolano P:  ${Math.round(angles.rightKnee ?? 0)}°`);
    lines.push(`Biodro L:  ${Math.round(angles.leftHip ?? 0)}°`);
    lines.push(`Biodro P:  ${Math.round(angles.rightHip ?? 0)}°`);
    lines.push('--- WZROK ---');
    lines.push(`Lewe oko: X=${gaze.lx.toFixed(2)} Y=${gaze.ly.toFixed(2)}`);
    lines.push(`Prawe oko: X=${gaze.rx.toFixed(2)} Y=${gaze.ry.toFixed(2)}`);

    // Pozycja nosa (punkt 0 - centrum twarzy):
    if (pose.length > 0) {
      const nose = pose[0];
      lines.push('--- POZYCJA ---');
      lines.push(`Nos: (${nose.x.toFixed(3)}, ${nose.y.toFixed(3)})`);
    }

    return lines;
  });

  if (!font) return null;

  const panelW = 160, lineH = 16, padX = 6, padY = 4;
  const panelH = panelLines.value.length * lineH + padY * 2;

  return (
    <Canvas style={{ position: 'absolute', top: 60, right: 8, width: panelW, height:
panelH }}>
      {/* Tło panelu - półprzezroczysty czarny prostokąt */}
      <Rect x={0} y={0} width={panelW} height={panelH}
        color='rgba(0,0,0,0.65)' r={6} />

      {/* Linie tekstu z parametrami */}
      {panelLines.value.map((line, i) => (
        <SkText key={i}
          x={padX}
          y={padY + (i + 1) * lineH}
          text={line}
          font={font}
          color={line.startsWith('---') ? '#FFD60A' : '#FFFFFF'}
        />
      ))}
    </Canvas>
  );
}

```

9.2. Etykiety przy punktach landmarkow

```

// Wświetlanie ID i koordynatów bezpośrednio przy punktach na szkielecie
// Rysujemy etykiety tylko przy wybranych punktach (nie wszystkich 468!)

const KEY_POSE_POINTS = [0, 11, 12, 13, 14, 15, 16, 23, 24, 25, 26, 27, 28];

export function LandmarkLabels({ landmarks, fw, fh, font }) {
  return (
    <Group>
      {KEY_POSE_POINTS.map(idx => {
        const lm = landmarks.value[idx];
        if (!lm || (lm.visibility ?? 1) < 0.5) return null;

        const x = lm.x * fw;
        const y = lm.y * fh;

        // Żółty krążek z ID:
        return (
          <Group key={idx}>

```

```
    <Circle cx={x} cy={y} r={4} color='#FFD60A' />
    <SkText
      x={x + 6} y={y - 4}
      text={` ${idx} `}
      font={font} color='#FFD60A'
    />
    { /* Pozycja znormalizowana pod krazkiem: */ }
    <SkText
      x={x + 6} y={y + 10}
      text={` (${lm.x.toFixed(2)}, ${lm.y.toFixed(2)}) `}
      font={font} color='rgba(255,255,255,0.7)'
    />
  </Group>
);
  )}
</Group>
);
}
```

10. Optymalizacja wydajności - stabilne 30 fps na urządzeniu

Przetwarzanie wizji komputerowej w czasie rzeczywistym stawia ekstremalne wymagania wydajnościowe. Błąd projektowy w jednym miejscu może obniżyć fps z 30 do 8, co jest nieakceptowalne dla użytkownika. Poniższe techniki pozwalają osiągnąć i utrzymać stabilne 30 fps na średniej klasy smartfonach z Androidem (2021+).

10.1. Strategie optymalizacji - zestawienie

Technika	Efekt	Implementacja
Delegate: GPU w MediaPipe	3-5x szybsze wnioskowanie	delegate: 'GPU' w opcjach createFromOptions. GPU/NPU jest specjalizowany dla operacji tensorowych.
runningMode: VIDEO zamiast IMAGE	20-40% szybsze dla strumieni	Tryb VIDEO optymalizuje między klatkami (śledzenie zamiast pełnej detekcji co klatkę).
Frame skipping - pomijanie klatek	Redukcja obciążenia CPU o 50%	Analizuj co 2. klatkę dla Holistic (25ms), teczówki co 3. klatkę (mniej się zmienia).
Nizsze progi pewności śledzenia	Mniej pełnych detekcji od nowa	minTrackingConfidence: 0.3 zamiast 0.5 - rzadziej resetuje śledzenie.
Mniejsza rozdzielczość kamery	2-4x mniejszy bufor do przes.	720p wystarczy. MediaPipe i tak przeskalowuje do 256x256.
Ograniczenie etykiet landmarków	Szybsze renderowanie Skia	Wyswietlaj etykiety tylko dla KEY_POSE_POINTS (13 z 33), nie dla wszystkich.
useDerivedValue zamiast useAnimatedStyle	Mniej przeliczeń na watku UI	useDerivedValue przelicza tylko gdy SharedValue się zmienia. useAnimatedStyle przelicza przy każdym renderze.

10.2. Implementacja frame skipping

```
// Frame skipping - optymalizacja dla długich operacji
const frameCounter = useSharedValue<number>(0);
const HOLISTIC_EVERY_N_FRAMES = 2; // Holistic co 2. klatkę
const IRIS_EVERY_N_FRAMES = 3; // Iris co 3. klatkę

const frameProcessor = useFrameProcessor((frame) => {
  'worklet';

  frameCounter.value += 1;
  const n = frameCounter.value;

  // Holistic co 2. klatkę:
  if (n % HOLISTIC_EVERY_N_FRAMES === 0) {
    const result = detectHolistic(frame, Date.now());
    if (result) {
```

```

    poseLandmarks.value = result.poseLandmarks?.[0] ?? [];
    faceLandmarks.value = result.faceLandmarks?.[0] ?? [];
    leftHandLMs.value   = result.leftHandLandmarks?.[0] ?? [];
    rightHandLMs.value  = result.rightHandLandmarks?.[0] ?? [];
  }
}

// Iris co 3. klatke (wolniej sie zmienia):
if (n % IRIS_EVERY_N_FRAMES === 0 && faceLandmarks.value.length >= 468) {
  // Iris jest w punktach 468-477 wyników Face Landmarker
  irisLeft.value = faceLandmarks.value.slice(468, 473);
  irisRight.value = faceLandmarks.value.slice(473, 478);
}

}, [])); // Pusta tablica - frameProcessor jest stabilny (nie zmienia sie miedzy ren-
derami)

```

10.3. Pomiar i wyświetlanie FPS

```

// Pomiar realnego FPS (nie tylko nominalne 30 z konfiguracji kamery):
const fpsHistory = useSharedValue<number[]>([]);
let lastTs = 0;

// W frameProcessor:
const ts = Date.now();
if (lastTs > 0) {
  const instantFps = 1000 / (ts - lastTs);
  // Srednia krocząca z ostatnich 10 klatek:
  const history = [...fpsHistory.value.slice(-9), instantFps];
  fpsHistory.value = history;
  const avgFps = history.reduce((a, b) => a + b, 0) / history.length;
  fps.value = Math.round(avgFps);
}
lastTs = ts;

// Kolor FPS zalezy od wartosci (zielony-zolty-czerwony):
const fpsColor = useDerivedValue(() => {
  const f = fps.value;
  if (f >= 25) return '#00FF00'; // Zielony: 25+ fps - OK
  if (f >= 15) return '#FFD60A'; // Zolty: 15-24 fps - akceptowalne
  return '#FF3B30';             // Czerwony: < 15 fps - problem
});

```

11. Zadania do wykonania

Zadania prowadzi przez budowę kompletnego modułu PoseTracker. Realizuj je w kolejności - każde następne zależy od poprzedniego. Weryfikuj osiągnięte fps po każdym kroku.

Zadanie 1 (20 pkt) - Konfiguracja środowiska i podgląd kamery

1. Zainstaluj wszystkie zależności: react-native-vision-camera, react-native-worklets-core, react-native-mediapipe, @shopify/react-native-skia, react-native-reanimated. Skonfiguruj Babel (kolejność pluginów!). Uruchom `npx expo prebuild --clean`.
2. Skonfiguruj `app.config.ts` z uprawnieniami kamery (`cameraPermissionText`). Napisz komponent `CameraScreen` z `Camera` (device: 'front', pixelFormat: 'yuv', fps: 30). Sprawdź, że podgląd kamery działa na emulatorze lub fizycznym urządzeniu.
3. Nałóż puste płotno Skia (`Canvas` z `StyleSheet.absoluteFill`) na podgląd kamery. Narysuj testową linię po przekątnej ekranu, aby potwierdzić, że Skia renderuje się nad kamerą.
4. Zaimplementuj FPS counter w `Frame Processorze` (pomiar czasu między klatkami). Wyświetl wartość FPS w rogu ekranu przez `Canvas > Text` w Skia. Upewnij się, że licznik jest widoczny i aktualizuje się.
5. Przetestuj na fizycznym urządzeniu (emulator ma ograniczoną kamerę). Zmierz bazowe FPS samego podglądu bez analizy - powinno być 30 fps.

WERYFIKACJA: Podgląd kamery działa. Linia testowa Skia widoczna nad kamerą. FPS counter wyświetla wartości w czasie rzeczywistym. Na urządzeniu fizycznym bazowe fps ≥ 28 .

Zadanie 2 (25 pkt) - Integracja MediaPipe Holistic

6. Zaimplementuj `holisticService.ts` z funkcjami `initHolistic()` i `detectHolistic()`. Wywołaj `initHolistic()` w `useEffect` przy montowaniu głównego ekranu. Wyświetl komunikat 'Ładowanie modelu...' podczas inicjalizacji.
7. Zdefiniuj stałe `POSE_CONNECTIONS`, `HAND_CONNECTIONS` i `FACE_OUTLINE_CONNECTIONS` w `constants/mediapipe.ts` (zgodnie z treścią sekcji 4.2). Sprawdź poprawność indeksów przez rysowanie testu z jednym połączeniem.
8. Zbuduj `usePoseFrameProcessor` hook zwracający `SharedValues` `poseLandmarks`, `leftHandLMs`, `rightHandLMs`, `faceLandmarks`. Uruchom `detectHolistic` w `Frame Processorze`. Zastosuj `frame skipping` (Holistic co 2. klatkę).
9. Zaimplementuj `PoseSkeleton`, `HandSkeleton` i `FaceOutline` jako komponenty Skia czytające `SharedValues`. Nakładaj szkielety na podgląd kamery. Sprawdź, że kolorowanie lewo/prawo działa poprawnie (czerwony/niebieski).
10. Zmierz FPS z włączonym Holistic na fizycznym urządzeniu. Cel: ≥ 20 fps. Jeśli niższe, zwiększ `HOLISTIC_EVERY_N_FRAMES` lub obniż próg `minTrackingConfidence` do 0.3.

WERYFIKACJA: Szkielet ciała widoczny podczas ruchu użytkownika. Dłonie rysowane zielono (lewa) i pomarańczowo (prawa). Kontur twarzy widoczny. FPS ≥ 20 na urządzeniu fizycznym.

Zadanie 3 (30 pkt) - Iris Landmarker i parametry liczbowe

11. Dodaj śledzenie teczówki: w `Frame Processorze` czytaj punkty 468-477 z tablicy `faceLandmarks` (wymagane włączenie `outputIrisLandmarks: true`). Uzupełnij `SharedValues` `irisLeft` i `irisRight`. Zaimplementuj `IrisOverlay` rysujący okrąg teczówki i punkt centrum.
12. Zaimplementuj `computeGazeVector` z `utils` zgodnie z sekcją 5.2. Oblicz `GazeVector` dla lewego i prawego oka w `Frame Processorze`. Zapisz wyniki do `SharedValue gazeVector`.

13. Zaimplementuj `calculateJointAngle` i `normalizedDistance` w `utils/angles.ts`. Obliczaj w `FrameProcessorze` 6 katów: oba łokcie, oba kolana, oba biodra. Zapisz do `SharedValue jointAngles`.
14. Zbuduj `ParametersPanel` (polprzezroczysty panel z boku ekranu) wyświetlający: FPS, 6 kątów stawów (w stopniach), wektor wzroku lewego i prawego oka (x,y), pozycję nosa (x,y,z). Użyj `Skia Text` z `monospace` fontem.
15. Dodaj etykiety przy kluczowych 13 punktach landmarków ciała: wyświetl ID punktu i jego znormalizowane współrzędne (x.xxx, y.xxx) jako mały żółty tekst przy krawędzi markera.

WERYFIKACJA: Teczówki otoczone jasnoniebieskim (lewe) i magentowym (prawe) okrągiem. Panel parametrów wyświetla kąty i wektor wzroku. Etykiety ID i współrzędnych widoczne przy 13 punktach. FPS ≥ 18 .

Zadanie 4 (25 pkt) - Optymalizacja i eksport danych

16. Zmierz fps przed i po wdrożeniu pełnej optymalizacji. Zastosuj: `frame skipping` (`Holistic/2`, `Iris/3`), `delegate: GPU`, `runningMode: VIDEO`. Porównaj wyniki w tabeli i dołącz do sprawozdania.
17. Dodaj tryb 'LITE' (przełącznik w UI): wyłącza `Face Outline` i etykiety landmarków, zmniejsza `strokeWidth` linii do 1px. Tryb LITE powinien dawać ≥ 25 fps na urządzeniu testowym.
18. Zaimplementuj eksport klatki: przycisk 'Zapisz dane' zapisuje do pliku JSON bieżące wartości wszystkich `SharedValues` (pozycje, kąty, wektor wzroku) wraz ze znacznikiem czasowym. Użyj `expo-file-system` do zapisu.
19. Napisz min. 2 testy jednostkowe dla funkcji `calculateJointAngle`: test dla kąta 180 stopni (wyprostowane ramie - punkty na jednej prostej), test dla kąta 90 stopni (łokiec prosty - punkty prostopadle). Sprawdź dokładność do 1 stopnia.

WERYFIKACJA: FPS ≥ 24 z pełną analizą w trybie normalnym. Tryb LITE daje ≥ 28 fps. Plik JSON zapisany i czytelny. 2 testy `calculateJointAngle` przechodzą bez błędów.

12. Kryteria oceniania

Zadanie	Punkty	Co weryfikuje prowadzący
Zad. 1: Środowisko i kamera	20 pkt	Podgląd kamery z Skia overlay. FPS counter na ekranie. Fizyczne urządzenie ≥ 28 fps.
Zad. 2: Holistic Landmarker	25 pkt	Szkielet ciała, dłoni i twarzy. Kolorowanie lewo/prawo. FPS ≥ 20 na urządzeniu.
Zad. 3: Iris i parametry	30 pkt	Teczowki z okragami. Panel parametrow z katami i wzrokiem. Etykiety ID przy landmarkach.
Zad. 4: Optymalizacja	25 pkt	Tabela FPS przed/po. Tryb LITE. Eksport JSON. 2 testy jednostkowe katow.
RAZEM	100 pkt	

Ocena	Punkty / Wymagania
5.0	90-100 pkt - Wszystkie zadania. Stabilne ≥ 24 fps. Eksport danych. Testy jednostkowe. Tryb LITE.
4.5	80-89 pkt - Zadania 1-3 kompletne. Teczowki działają. Panel parametrow kompletny. FPS ≥ 18 .
4.0	70-79 pkt - Zadania 1-2 + panel z katami i wektorem wzroku. Brak etykiet ID. FPS ≥ 15 .
3.5	60-69 pkt - Zadania 1-2: szkielet ciała, dłoni i twarzy widoczny. FPS ≥ 12 .
3.0	50-59 pkt - Zadanie 1: kamera działa, FPS counter widoczny. Holistic w trakcie implementacji.
2.0	0-49 pkt - Kamera nie działa lub prebuild nie przeszedł pomyślnie.

13. Najczęstsze błędy i ich rozwiązania

Błąd / Objaw	Przyczyna i rozwiązanie
Brak klatki na podglądzie - czarny ekran po prebuild	Uprawnienie kamery nie zostało przyznane lub konfiguracja kamery jest nieprawidłowa. Sprawdź czy w app.config.ts dodano plugin react-native-vision-camera z cameraPermission-Text. Na urządzeniu: Ustawienia > Aplikacje > PoseTracker > Uprawnienia > Kamera.
'worklet' nie jest rozpoznawane - błąd kompilacji Babel	Plugin react-native-worklets-core/plugin nie jest pierwszym pluginem w babel.config.js, lub brakuje go całkowicie. Kolejność musi być: worklets-core, potem reanimated (ostatni). Uruchom npx expo start --clear po zmianie konfiguracji Babel.
MediaPipe: 'Model file not found' lub timeout ładowania	URL modelu jest niedostępny lub sieć jest zbyt wolna. Pobierz plik .task i dołącz do assets/models/. Zmień modelAssetPath na: require('./assets/models/holistic_landmarker.task'). Dodaj pole plugins w app.config.ts.
Szkielet odwrócony (lewo/prawo zamienione)	Kamera przednia (selfie) odwraca obraz poziomo. MediaPipe zwraca współrzędne dla oryginalnego (nieodwróconego) kadru. Rozwiązanie: przy rysowaniu zastosuj transformację: $x_screen = (1 - lm.x) * frameWidth$ dla kamery przedniej.
FPS spada do 5-8 z Holistic włączonym	delegate jest ustawiony na CPU zamiast GPU, lub runningMode to IMAGE zamiast VIDEO. Sprawdź konfigurację createFromOptions. Jeśli GPU nie jest dostępne (starszy emulator), spróbuj delegate: 'NNAPI'. Zwiększ HOLISTIC_EVERY_N_FRAMES do 3 lub 4.
Punkty teczówki są puste (irisLeft.value.length === 0)	Wynik Face Landmarker nie zawiera punktów iris gdy outputIrisLandmarks jest false lub model nie jest face_landmarker (a holistic_landmarker). Iris jest dołączone do wyników Face Landmarker jako punkty 468-477. Sprawdź długość tablicy faceLandmarks - musi być ≥ 478 .
Skia Canvas nie wyświetla nic ponad kamera	Canvas musi mieć position: 'absolute' i identyczne wymiary jak Camera. Sprawdź że zIndex Canvas jest wyższy niż Camera. Użyj StyleSheet.absoluteFill dla obu komponentów i owin je w View z position: 'relative'.
Tekst Skia nie wyświetla się (useFont zwraca null)	Font nie został załadowany przed pierwszym renderem. useFont jest asynchroniczne. Dodaj warunek: if (!font) return null przed użyciem font w Canvas. Sprawdź że plik .ttf jest w katalogu assets/fonts/ i dodany do app.config.ts w polu fonts.
calculateJointAngle zwraca NaN lub 0 dla	Jeden z landmarków ma współrzędne (0,0) co

wszystkich pozycji	oznacza ze nie zostal wykryty. Sprawdz pole visibility: points z visibility < 0.5 nalezy pominiac. Dodaj guard: if (!a !b !c) return 0; przed obliczeniami.
Frame Processor wyrzuca 'Cannot access SharedValue from non-Reanimated thread'	SharedValue jest uzywana poza kontekstem Reanimated (np. w useEffect lub setTimeout). Dostep do SharedValue.value w Frame Processorze jest poprawny. Poza Frame Processorem uzyj runOnUI() lub odczytuj wartosc w useDerivedValue.

Komendy diagnostyczne

```
# Sprawdz czy prebuild przeszedl pomyslnie:
ls android/app/src/main/AndroidManifest.xml # Powinien istniec
grep -i 'CAMERA' android/app/src/main/AndroidManifest.xml # Uprawnienie kamery

# Logi natywne z urzadzenia (filtruj po nazwie biblioteki):
adb logcat | grep -E '(VisionCamera|MediaPipe|Skia|Reanimated)'

# Sprawdz dostepne formaty kamery na podlaczonym urzadzeniu:
adb shell dumpsys media.camera | grep -A 5 'Front-facing'

# Sprawdz czy urzadzenie ma NPU/GPU dostepne dla MediaPipe:
adb shell getprop ro.hardware.chipname

# Weryfikacja ladowania modelu MediaPipe (powinien pojawic sie log):
adb logcat | grep -i mediapipe | grep -i 'model'

# Pomiar zuzycia CPU i RAM podczas dzialanego PoseTracker:
adb shell top | grep 'com.pam.posetracker'

# Reset cache Metro gdy Babel nie laduje pluginow:
npx expo start --clear

# Sprawdz czy font jest poprawnie dolaczony do APK:
unzip -l android/app/build/outputs/apk/debug/app-debug.apk | grep fonts
```

Instrukcja Laboratoryjna React Native 5

Przetwarzanie obrazu w czasie rzeczywistym - MediaPipe Holistic i Iris, React Native Skia

Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki

Nastepne cwiczenie: React Native 6 - Offline-First, GraphQL i Push Notifications