POLITECHNIKA ŚWIĘTOKRZYSKA

Aplikacje mobilne – wykład 8

Architektura, testy i wydajność

Mateusz Pawełkiewicz

1.10.2025

Wprowadzenie: W tym wykładzie omówimy nowoczesne podejście do architektury aplikacji mobilnych React Native, strategie testowania (od testów jednostkowych po end-to-end) oraz techniki debugowania i optymalizacji wydajności. Wszystkie porady odpowiadają aktualnym dobrym praktykom – będziemy korzystać z najnowszych bibliotek i wzorców.
W szczególności skupimy się na organizacji projektu w stylu *feature-first*, podziale na warstwy logiczne, odseparowaniu warstwy domenowej od danych z API (DTO vs. modele domenowe), pisaniu testów jednostkowych, komponentów, hooków i E2E (Detox), użyciu narzędzi debugowania (Flipper, React DevTools) oraz na optymalizacji list i animacji (FlatList tuning, unikanie zbędnych renderów, wprowadzenie do Reanimated). Całość zilustrujemy przykładami kodu z komentarzami, aby ułatwić zrozumienie i praktyczne zastosowanie omawianych koncepcji.

1. Architektura aplikacji React Native

W dobrze zaprojektowanej aplikacji React Native kluczowa jest odpowiednia **architektura projektu** – czyli organizacja plików, modułów i warstw w kodzie. Dobra architektura ułatwia skalowanie aplikacji, pracę zespołową, testowanie oraz utrzymanie kodu. Omówimy podejście *feature-first* do organizacji kodu, wyjaśnimy podział na warstwy logiczne (ui, hooks, services, api, types, utils), a także koncepcję **DTO vs. model domenowy** oraz mapowanie danych i obsługę błędów z backendu.

1.1 Podejście *feature-first* vs. tradycyjna architektura warstwowa

Tradycyjnie wiele projektów React Native (a także React) organizowano *warstwowo* – pliki były grupowane według typu, np. katalogi globalne components/, screens/, utils/, services/, types/ itp. Taka struktura jest początkowo intuicyjna, ale wraz z rozrostem projektu może sprawiać problemy. Gdy aplikacja rośnie do kilkunastu czy kilkudziesięciu ekranów, podejście warstwowe często prowadzi do:

- **Słabej skalowalności** – funkcje powiązane z jednym featurem są porozrzucane po wielu folderach, przez co łatwiej o niechciane zależności między modułami. Zmiana w jednym miejscu może niechcący wpłynąć na inny obszar aplikacji.
- **Ścisłego sprzężenia (tight coupling)** – brak jasnych granic między domenami biznesowymi. Kod różnych funkcjonalności miesza się, co utrudnia izolowanie i modyfikację pojedynczych elementów.
- **Konfliktów w zespole** – wielu deweloperów może pracować na tych samych „globalnych" plikach (np. dodając rzeczy do jednego utils.js czy modyfikując wspólny store), co zwiększa ryzyko konfliktów i błędów integracji.
- **Trudności w testowaniu** – ciężko jest przetestować moduł w izolacji, skoro logika jednego feature'u rozsiana jest po różnych warstwach. Mockowanie zależności bywa skomplikowane, bo brak wyraźnych granic między komponentami a logiką.
- **Problemów z utrzymaniem** – dodanie nowej funkcji potęguje „chaos", bo często wymaga edycji wielu rozproszonych plików. Kod z czasem staje się mniej czytelny, a refaktoryzacja – ryzykowna.

Rozwiązaniem tych problemów jest podejście **feature-first**, które odwraca tradycyjny schemat. Zamiast grupować pliki według typu, grupujemy je według *funkcjonalności*

(feature). Każdy główny moduł funkcjonalny aplikacji ma swój własny podkatalog zawierający wszystko, co z nim związane: komponenty UI, ekrany, logikę (hooks, serwisy), definicje typów, a nawet pod-moduł do komunikacji z API. Taki *feature folder* jest jakby mini-aplikacją wewnątrz projektu – ma większość rzeczy potrzebnych do działania danej funkcji, co przypomina podejście mikroserwisów w backendzie, ale zastosowane lokalnie w kodzie front-end.

Przykładowa struktura feature-first: wyobraźmy sobie aplikację z modułami: uwierzytelnianie, ekran główny, onboarding, ustawienia, lista zadań (todos). Struktura projektu może wyglądać następująco:

```
src/
├── features/
│   ├── auth/
│   │   ├── api/       # kod do komunikacji z API dla modułu auth (np. login, logout)
│   │   ├── components/ # komponenty UI specyficzne dla auth (np. formularz logowania)
│   │   ├── hooks/     # hooki związane z auth (np. useAuth)
│   │   ├── screens/   # ekrany (React components) modułu auth (np. LoginScreen)
│   │   ├── services/  # logika biznesowa auth (np. obsługa tokenów, walidacja)
│   │   ├── store/     # (opcjonalnie) stan globalny specyficzny dla auth (np. Redux slice)
│   │   └── types/     # definicje typów (TypeScript) związane z auth (np. UserCredentials)
│   ├── home/
│   │   └── ...        # analogiczna struktura dla modułu ekranu głównego
│   ├── onboarding/
│   ├── settings/
│   └── todos/         # moduł listy zadań
├── navigation/        # nawigacja aplikacji (np. stacki, tab navigator – często globalne)
├── services/          # usługi współdzielone (np. wspólny klient API, konfiguracja)
├── store/             # globalny store (np. konfiguracja Redux, persystencja)
├── ui/                # wspólne komponenty UI (np. przycisk używany w wielu miejscach)
└── utils/             # funkcje pomocnicze wspólne (np. formatowanie dat, logowanie)
```

W powyższej strukturze każdy folder w `features/` reprezentuje względnie **samodzielny moduł biznesowy**, który *może być rozwijany, testowany, a nawet usunięty niezależnie od reszty aplikacji*. Taki podział wymusza luźne powiązania między modułami – komunikacja odbywa się poprzez zdefiniowane interfejsy (np. wywołania w `services/api`) zamiast poprzez dzielenie globalnych zmiennych. Dzięki temu:

- Aplikacja lepiej się skaluje – dodając nowy feature, dodajemy nowy folder, minimalizując wpływ na istniejący kod.
- Deweloperzy mogą pracować równolegle nad różnych funkcjonalnościach bez wchodzenia sobie w drogę (mniej konfliktów).
- **Testowanie** staje się prostsze – możemy łatwiej odpalić testy dla całego modułu feature (lub zamockować jeden moduł podczas testowania innego). Poszczególne funkcjonalności są z definicji izolowane, co sprzyja tworzeniu testów jednostkowych i integracyjnych.
- **Czytelność i utrzymanie** – kod związany z danym zagadaniem jest zgrupowany, co ułatwia nowym osobom zrozumienie, gdzie szukać odpowiedniej logiki. Gdy coś nie działa np. w logowaniu, to wiadomo, że większość istotnego kodu będzie w `features/auth/*`.

Warto zaznaczyć, że podejście feature-first nie wyklucza posiadania pewnych współdzielonych części aplikacji – np. komponentów UI ogólnego użytku (src/ui), globalnej konfiguracji klienta HTTP (src/services/api), czy wspólnego stanu aplikacji (src/store). Jednak nawet te elementy można projektować tak, aby były wykorzystywane przez poszczególne feature'y w sposób *kompozycyjny*, a nie żeby zastępowały logikę feature. Np. możemy mieć globalny klient API, ale poszczególne moduły feature tworzą własne funkcje API (w swoim folderze api/), używając tego klienta do wywołań sieciowych.

Podsumowując: Architektura feature-first poprawia modularność i skalowalność projektu React Native.

1.2 Warstwy logiczne: ui, hooks, services, api, types, utils

W strukturze feature-first wyróżniamy **warstwy logiczne** wewnątrz każdego modułu. Wymienione katalogi (ui, hooks, services, api, types, utils) pełnią następujące role:

- **ui (interfejs użytkownika)** – komponenty prezentacyjne związane z danym featurem. Mogą to być np. specyficzne kontrolki, przyciski, karty, formularze używane tylko w ramach tego modułu. Często są to proste, świadome tylko własnych propsów komponenty (tzw. *dumb components*), które można reużyć. Jeśli jakiś komponent staje się potrzebny w wielu feature'ach, można go przenieść do wspólnego katalogu src/ui. Przykład: w module todos możemy mieć komponent TodoItem renderujący pojedyncze zadanie.
- **hooks** – custom hooki z logiką biznesową lub obsługujące stan lokalny feature'u. Hooki mogą ukrywać wewnętrzne szczegóły implementacji (np. sposób pobierania danych czy reagowania na zmiany) i mieć wygodny interfejs do komponentów. Przykład: useTodos() w module todos pobierający listę zadań z API, obsługujący stan ładowania, błąd itp. Hooki umożliwiają też łatwe ponowne wykorzystanie kodu w różnych komponentach (np. ten sam hook użyty na różnych ekranach).
- **services** – warstwa **logiki biznesowej / domenowej**. Tutaj umieszczamy kod realizujący konkretne przypadki użycia (use-cases) niezwiązany bezpośrednio z UI. Services mogą wywoływać funkcje z warstwy api, mogą też zawierać np. zarządzanie stanem specyficznym dla feature (np. prosty *store* oparty o Context lub operacje na globalnym store). W niektórych projektach ta warstwa bywa nazwana *use-cases*, *logic* lub *controllers*. Przykład: todos/services/todoService.ts może mieć funkcje takie jak addTodo, toggleTodoCompleted – operujące na danych zadań poprzez wywołanie API i aktualizację stanu lokalnego.
- **api** – warstwa dostępu do danych zewnętrznych. Zawiera funkcje/fetchery do komunikacji z zewnętrznym API (backendem) lub bazą danych lokalną. W wielu nowoczesnych aplikacjach stosuje się tu np. React Query lub RTK Query do efektywnego pobierania i cache'owania danych, albo klasyczne wywołania fetch/axios. Warstwa api powinna być **odseparowana od reszty** – tak, by zmiana szczegółów API (endpoints, formaty danych) nie wpływała na kod UI. Przykład: auth/api/authApi.ts może eksportować funkcję login(credentials) która wykonuje POST /login i zwraca obiekt np. AuthToken. W praktyce może to być zaimplementowane np. przy pomocy biblioteki RTK Query, która pozwala definiować *endpointy* dla mutacji i zapytań.

- **types** – definicje typów i interfejsów (jeśli korzystamy z TypeScript, co jest obecnie standardem w RN). Oddzielenie typów ułatwia ich reużycie między warstwami (np. interfejs Todo używany w api i w ui). Często dzieli się typy na DTO (typ odpowiedzi z API) i modele domenowe – o czym więcej za chwilę. W folderze types możemy też umieścić np. typy dla propsów komponentów, aby nie bałaganić głównego kodu komponentu.
- **utils** – pomocnicze funkcje specyficzne dla danego modułu. Mogą to być np. formatowanie danych, funkcje do walidacji, parsery itp. Jeśli jakaś funkcja utils okazuje się globalnie przydatna, może trafić do src/utils. Na poziomie feature trzymamy tylko to, co unikalne dla danego obszaru. Przykład: utils/formatTodoTitle.ts – prosta funkcja formatująca tytuł zadania (np. przycinająca zbyt długie nazwy).

Zalety podziału na warstwy: Porządek ten wymusza czysty podział odpowiedzialności (Separation of Concerns). UI zajmuje się wyłącznie prezentacją, nie musi wiedzieć skąd pochodzą dane – dostaje je od hooka/serwisu. Serwis zna logikę biznesową, ale nie zajmuje się renderowaniem – wywołuje API i np. zarządza stanem. Warstwa API izoluje szczegóły komunikacji z serwerem (adresy URL, struktura JSON-ów) od reszty aplikacji. Dzięki temu zmiany w jednej warstwie minimalnie wpływają na inne. Taka organizacja sprzyja **testowalności** – możemy np. testować logikę serwisu niezależnie, podając mu zamockowaną warstwę API (albo używając tzw. *dummy data* zamiast realnych wywołań). Podobnie, komponenty UI możemy testować, podając im fikcyjne dane i custom hooki, bez faktycznego kontaktu z backendem.

1.3 DTO vs. modele domenowe (Domain Models)

Przy pracy z warstwą danych warto rozróżnić dwa pojęcia: **DTO (Data Transfer Object)** i **model domenowy**. W kontekście aplikacji klienckiej (RN) DTO to zazwyczaj struktura danych dokładnie taka, jak przychodzi z API, natomiast model domenowy to reprezentacja danych, której używamy wewnątrz naszej aplikacji (dopasowana do potrzeb interfejsu i logiki biznesowej).

- **DTO** – to obiekt przenoszący dane pomiędzy systemami (np. między serwerem a naszą aplikacją). Ma taką strukturę, jak zdefiniował backend, często zagnieżdżoną, zawierającą pola potrzebne głównie z perspektywy serwera. DTO może zawierać np. pola techniczne, identyfikatory, relacje, które nie zawsze są bezpośrednio potrzebne w UI. Często API stosują standardy (np. JSON:API), które narzucają pewien kształt odpowiedzi (np. obiekt attributes, relationships itp.).
- **Model domenowy** (czasem zwany też *model biznesowy* lub *encja domenowa*) – to nasza własna, wewnętrzna reprezentacja danych, dopasowana do potrzeb aplikacji. Model domenowy powinien być prostszy, zawierać tylko to, co naprawdę potrzebne aplikacji i w formie wygodnej do użycia w kodzie. Może również zawierać metody czy logikę operującą na danych (choć w JavaScript/TypeScript częściej modele to po prostu interfejsy/typy, a logika jest w funkcjach osobno). Model domenowy jest niezależny od tego, jak dane są przechowywane czy przesyłane – opisuje koncepty biznesowe w sposób zrozumiały dla dewelopera i (teoretycznie) dla analityka biznesowego.

Kluczowa różnica: **DTO istnieje z powodów technicznych**, aby przenieść dane przez sieć lub między warstwami, natomiast **model domenowy istnieje z powodów biznesowych** – odwzorowuje pojęcia, którymi posługuje się domena/problem, który rozwiązujemy. Często model domenowy ma *bogatsze zachowania* (metody biznesowe, walidacje) lub przynajmniej jasno odzwierciedla język dziedziny (np. klasa/typ Order z metodą calculateTotal() w przeciwieństwie do DTO OrderDTO będącego tylko "głupim" zbiorem pól).

W praktyce w aplikacjach React/React Native modele domenowe bywają **spłaszczone** i uproszczone w porównaniu do DTO. Przyjrzyjmy się przykładowi, aby to zilustrować:

Przykład: Załóżmy, że backend zwraca nam użytkownika w formacie zgodnym z JSON:API, np.:

```
// Przykładowy DTO użytkownika z API
{
  "id": "123",
  "type": "user",
  "attributes": {
    "handle": "jan_kowalski",
    "avatar": "https://example.com/avatar.jpg",
    "info": "Bio użytkownika..."
  },
  "relationships": {
    "followerIds": ["u1", "u2", "u3"]
  }
}
```

To jest **DTO** – struktura dokładnie taka, jak w odpowiedzi serwera. W naszym kodzie jednak korzystanie z tak zagnieżdżonego obiektu byłoby uciążliwe (ciągłe odwołania user.attributes.handle itd.), a co gorsza – uzależnia nas to mocno od formatu serwera. Gdyby backend zmienił strukturę (np. zrezygnował z pola attributes), nasz kod UI by się posypał, bo wszędzie odwołujemy się do user.attributes.handle.

Dlatego w warstwie **mapowania danych** przekształcamy DTO na *model domenowy*, np. taki:

```
// Model domenowy User w aplikacji (TypeScript interface)
interface User {
  id: string;
  handle: string;
  avatar: string;
  info?: string;
  followerIds: string[];
}
```

Tutaj mamy wszystkie potrzebne pola, *spłaszczone* i nazwane zgodnie z intuicją. Takim modelem posługuje się reszta aplikacji (UI, logika) – jest prosty i nie zawiera zbędnych zagnieżdżeń. Jak go uzyskać? Poprzez **mapowanie DTO -> model** w warstwie api lub services.

Możemy napisać funkcję mappera, np.:

```
// src/features/user/api/userMapper.ts
import { UserDTO } from '../types';   // zakładamy, że UserDTO to typ odpowiadający strukturze z API
```

```
import { User } from '../types';    // nasz model domenowy

export function mapUserDtoToModel(dto: UserDTO): User {
  return {
    id: dto.id,
    handle: dto.attributes.handle,
    avatar: dto.attributes.avatar,
    info: dto.attributes.info,
    followerIds: dto.relationships.followerIds,
  };
}
```

Taką funkcję wywołujemy zaraz po otrzymaniu danych z API, zanim przekażemy je dalej. Przykładowy fragment serwisu:

```
// src/features/user/services/userService.ts
import { apiClient } from '../../../services/apiClient'; // globalny klient HTTP (np. axios)
import { mapUserDtoToModel } from '../api/userMapper';

export async function fetchUser(handle: string): Promise<User> {
  const response = await apiClient.get<{ data: UserDTO }>(`/user/${handle}`);
  const userDto = response.data.data;
  const user = mapUserDtoToModel(userDto);
  return user;
}
```

Dzięki temu **komponenty UI i logika biznesowa operują na wygodnym modelu User**, np. mogą bezpośrednio pisać user.handle zamiast user.attributes.handle. Zmniejsza to złożoność kodu frontendu i uniezależnia nas od ewentualnych zmian w API. Gdy backend zmieni format, wystarczy zmodyfikować funkcję mapującą, zamiast przeszukiwać i poprawiać użycia w dziesiątkach komponentów.

Ponadto, odseparowanie modeli od DTO pomaga w **obsłudze błędów i walidacji**. Np. jeśli serwer zwraca błąd walidacji z informacjami w specyficznym formacie (np. pola errors zawierające listę komunikatów), to warstwa api/service może taki błąd przechwycić i przekształcić na przyjazny dla aplikacji obiekt błędu lub wyjątek. Można zdefiniować własny typ błędu domenowego, np. ValidationError z polami field i message, i mapować odpowiedź serwera na taki błąd. Dzięki temu komponenty lub hooki dostają już gotowy, zrozumiały obiekt błędu (np. do wyświetlenia komunikatu), zamiast surowej odpowiedzi serwera. Przykład obsługi błędu:

```
try {
  const user = await fetchUser('jan_kowalski');
  setUser(user);
} catch (e: any) {
  if (isValidationError(e)) {
    showToast(e.message);  // nasz własny komunikat z wyjątku domenowego
  } else {
    console.error('Unknown error fetching user', e);
  }
}
```

Funkcja fetchUser wewnątrz może zrobić coś takiego:

```
const response = await apiClient.get(...);
if (response.status === 400 && response.data.errors) {
  throw new ValidationError(mapError(response.data.errors));
}
```

Gdzie `mapError` przekształca strukturę błędu z API na naszą klasę/obiekt błędu. Oczywiście implementacja zależy od API – chodzi o zasadę, by **logikę błędów trzymać w warstwie komunikacji/serwisu, a nie rozrzucać ją po komponentach**. Komponent powinien dostać już przeprocesowaną informację (np. że logowanie nie powiodło się bo złe hasło), zamiast surowego HTTP 401 z ciałem JSON.

Podsumowanie: Oddzielenie DTO od modeli domenowych czyni nasz kod czytelniejszym i odporniejszym na zmiany. UI nie jest *ściśle sprzężone* z formatem danych serwera, co zmniejsza złożoność i poprawia odporność na zmiany backendu. Wprowadzenie warstwy mapowania to krok w kierunku czystszej architektury – *„czystego" frontendu, gdzie dane z API są izolowane w swojej warstwie* (nierzadko określanej jako *Domain Layer* w literaturze). Ten koncept jest inspirowany zasadami **DDD (Domain-Driven Design)**, choć stosujemy go w lżejszy sposób, dopasowany do realiów aplikacji mobilnej.

2. Testy w React Native – od jednostkowych do E2E

Testowanie aplikacji React Native przebiega na kilku poziomach. W nowoczesnym podejściu kładziemy nacisk zarówno na **testy jednostkowe** (pojedyncze funkcje, logika), **testy komponentów i hooków** (czyli testy integracyjne UI w kontrolowanym środowisku) oraz na **testy end-to-end (E2E)** symulujące prawdziwe użycie aplikacji. Omówimy po kolei te rodzaje testów, narzędzia takie jak **Jest**, **React Native Testing Library** i **Detox**, a także sposób konfiguracji środowiska testowego. Pokażemy też przykłady kodu testów komponentu i hooka, aby zademonstrować praktyczne podejście.

2.1 Testy jednostkowe (Jest)

Testy jednostkowe sprawdzają najmniejsze jednostki kodu – zwykle pojedyncze funkcje lub moduły – w izolacji od reszty. W ekosystemie React Native standardem do testów jednostkowych (jak i większości testów frontendowych) jest **Jest** – framework testowy dostarczany od razu przy tworzeniu projektu React Native (np. przez `npx react-native init`). Jest oferuje środowisko uruchomieniowe dla testów JavaScript z wbudowanym **domyślnym wsparciem dla React Native** (preset `react-native` w konfiguracji).

Test jednostkowy w Jest ma postać funkcji `test(…)` lub aliasu `it(…)`, w której opisujemy oczekiwane zachowanie kodu. Możemy używać asercji (matcherów) takich jak `expect(x).toBe(y)` albo `expect(obj).toHaveBeenCalled()` (dla mocków). Przykładowo, mając funkcję w `utils` liczącą sumę, test jednostkowy wyglądałby:

```
import { suma } from '../utils/calc';

test('poprawnie sumuje dwie liczby', () => {
  expect(suma(2, 3)).toBe(5);
});
```

Oczywiście prawdziwa siła testów jednostkowych ujawnia się przy bardziej złożonej logice. W projekcie RN będziemy pisać takie testy głównie dla **logiki biznesowej** (np. funkcje w services/utils) oraz dla *custom hooków* (o ile testujemy je bezpośrednio) czy reduktorów stanu. Testy jednostkowe są szybkie i izolowane – nie dotykają UI ani sieci, dzięki czemu uruchamiają się w milisekundach i dają nam informację zwrotną przy każdym buildu/commitcie.

Mockowanie zależności: Podczas testów jednostkowych często korzystamy z mechanizmu *mocków* w Jest. Pozwala on zastąpić faktyczne moduły lub funkcje „udawanymi" implementacjami, by testować dany kawałek kodu w oderwaniu od reszty. Przykład: testując funkcję fetchUser z poprzedniej sekcji, nie chcemy w teście robić prawdziwego requestu HTTP. Możemy więc zamockować moduł apiClient używając jest.mock('nazwaModułu') i definiując, że apiClient.get zwraca obiekt z przykładowymi danymi (symulacja odpowiedzi serwera). Dzięki temu test jest deterministyczny i szybki.

Konfiguracja Jest: W projektach RN minimalna konfiguracja to zainstalowanie zależności i ewentualnie pliku konfiguracyjnego jest.config.js. Typowa konfiguracja (jeśli korzystamy z React Native Testing Library, o czym za chwilę) może wymagać dodania do setupFiles rozszerzeń matcherów. Przykładowo, aby korzystać z dodatkowych asercji @testing-library/jest-native (np. toBeVisible(), toHaveTextContent()), do pliku konfiguracyjnego dodaje się:

```
module.exports = {
  preset: 'react-native',
  setupFilesAfterEnv: ['@testing-library/jest-native/extend-expect']
};
```

Tyle wystarczy, by zacząć pisać testy. Uruchomienie testów odbywa się poleceniem npm test (lub yarn test), które odpala wszystkie pliki o nazwach zakończonych na .test.js lub .spec.js (domyślnie w konfiguracji).

2.2 Testowanie komponentów i hooków (React Native Testing Library)

Testy jednostkowe pokrywają logikę, ale w aplikacjach UI ważne jest też przetestowanie **komponentów** – czyli czy interfejs renderuje się zgodnie z założeniami i reaguje na interakcje użytkownika. Do tego celu używamy **React Native Testing Library (RNTL)**, która dostarcza narzędzia do renderowania komponentów w środowisku testowym i sprawdzania ich zawartości oraz zachowania. RNTL jest adaptacją popularnej biblioteki Testing Library znanej z React web, dostosowaną do specyfiki React Native (np. obsługuje widoki, Text, itp.).

Instalacja: Zakładamy, że mamy już Jest. Instalujemy paczki: @testing-library/react-native oraz opcjonalnie @testing-library/jest-native (jak wyżej, dla dodatkowych matcherów). Te biblioteki pozwalają w testach używać funkcji takich jak render, fireEvent oraz asercji typu toHaveTextContent. Po instalacji i konfiguracji (jak wspomniano w 2.1), możemy pisać testy komponentów.

Renderowanie komponentu w teście: RNTL dostarcza funkcję render() do wyrenderowania komponentu w kontekście testowym (nie w prawdziwym emulatorze, tylko w wirtualnym środowisku przypominającym drzewo komponentów). render zwraca obiekt zawierający m.in.

metody do wyszukiwania elementów: getByText, getByTestId, getByRole itp. Dzięki temu możemy znaleźć wirtualnie wyrenderowany element i sprawdzić, czy istnieje, ma odpowiednie propsy, style, tekst, itp.

Przykład – przetestujmy prosty komponent SocialLinks, który wyświetla link do profilu użytkownika (np. Twitter):

```
// Komponent SocialLinks (props: label, link, type), np. pokazuje ikonkę i tekst
import React from 'react';
import { Text, TouchableOpacity, Image, Linking } from 'react-native';

const icons = {
  twitter: require('../assets/twitter.png'),
  instagram: require('../assets/instagram.png')
};

export const SocialLinks = ({ type, label, link }) => (
  <TouchableOpacity onPress={() => Linking.openURL(link)}>
    <Image source={icons[type]} accessibilityRole="image" />
    <Text>{label}</Text>
  </TouchableOpacity>
);
```

Chcemy przetestować, czy: (a) odpowiedni tekst label jest renderowany, (b) ikona (Image) jest obecna, (c) kliknięcie wywołuje otwarcie linku. W tym celu piszemy testy z użyciem RNTL:

```
import React from 'react';
import { Linking } from 'react-native';
import { render, fireEvent } from '@testing-library/react-native';
import { SocialLinks } from '../SocialLinks';

// Przygotowanie: zamockujemy Linking.openURL, by nie otwierał prawdziwej przeglądarki:
jest.spyOn(Linking, 'openURL').mockImplementation(() => Promise.resolve());

test('renderuje etykietę linku społecznościowego', () => {
  const { getByText } = render(
    <SocialLinks type="twitter" label="John Doe's Twitter" link="https://twitter.com/johndoe" />
  );
  const labelElement = getByText("John Doe's Twitter");
  expect(labelElement).toBeTruthy();  // sprawdzamy czy tekst się pojawił
});
```

W powyższym teście używamy render do wyrenderowania komponentu z przykładowymi propsami. Następnie getByText wyszukuje element <Text> po zawartości. Oczekujemy, że taki element istnieje (matcher toBeTruthy() sprawdza, czy znaleziony element nie jest null). Ten test upewnia nas, że komponent faktycznie wyświetla przekazany label. Gdyby ktoś przez pomyłkę zmienił komponent tak, że ignoruje prop label (np. wstawił na sztywno tekst), test by się nie powiódł, co wychwyci regresję.

Drugi test – czy ikona jest renderowana:

```
test('renderuje ikonę typu social (np. Twitter)', () => {
```

```
  const { getByRole } = render(
    <SocialLinks type="twitter" label="John Doe's Twitter" link="https://twitter.com/johndoe" />
  );
  const icon = getByRole('image');
  expect(icon).toBeTruthy();  // obecność obrazka (Image)
});
```

Tutaj użyliśmy getByRole('image'). W RNTL role odpowiadają atrybutom dostępności (accessibilityRole). Ponieważ w komponencie <Image> domyślnie ma accessibilityRole="image" (co jest mapowane na rolę), możemy w teście w ten sposób znaleźć obrazek. Test sprawdza tylko czy jest jakikolwiek obrazek – można by doprecyzować, np. sprawdzić czy źródło obrazu jest właściwe, ale często wystarczy sprawdzić obecność elementu (testy nie muszą wnikać we wszystkie detale UI, raczej w kluczowe aspekty).

Trzeci test – **reakcja na interakcję** (kliknięcie):

```
test('otwiera link po naciśnięciu komponentu', () => {
  const { getByText } = render(
    <SocialLinks type="twitter" label="John Doe's Twitter" link="https://twitter.com/johndoe" />
  );
  const labelElement = getByText("John Doe's Twitter");
  fireEvent.press(labelElement);  // symulujemy tap na cały TouchableOpacity (tu na jego tekst)
  expect(Linking.openURL).toHaveBeenCalledWith('https://twitter.com/johndoe');
});
```

W tym teście kluczowe jest przygotowanie: wcześniej użyliśmy jest.spyOn aby uczynić Linking.openURL funkcją mockowaną. Dzięki temu możemy sprawdzić wywołania tej funkcji. Używamy fireEvent.press(...) aby zasymulować naciśnięcie elementu dotykowego (nasz komponent opakowuje <Text> i <Image> w <TouchableOpacity>, więc kliknięcie tekstu też zadziała). Oczekujemy, że Linking.openURL została wywołana z konkretnym URL. Jeśli np. pomylimy i w komponencie wywołamy coś innego, test to wykryje.

Tak przetestowaliśmy kluczową funkcjonalność komponentu: renderowanie poprawnych danych i reagowanie na interakcję. Testy komponentów pozwalają wychwycić wiele błędów zanim uruchomimy aplikację na urządzeniu, np. brak wyświetlania ważnego elementu, złą nazwę pola, brak reakcji na kliknięcie itp. Co ważne, testy te są **deterministyczne** i dość szybkie (choć wolniejsze od czysto jednostkowych, bo muszą wyrenderować komponenty). Nie wymagają emulatora – działają w Node z symulowanym środowiskiem RN.

Testowanie custom hooków: Hooki również można testować, choć wymaga to albo utworzenia komponentu testowego, który wykorzysta hook, albo użycia dedykowanej biblioteki. Istnieje @testing-library/react-hooks (React Hooks Testing Library), która ułatwia testowanie logiki hooków bez komponowania ich w prawdziwy komponent. W przyszłości może zostać zintegrowana, ale na 2025 nadal można z niej korzystać. Przykład – załóżmy prosty hook useCounter(initialValue) zwracający aktualny licznik i funkcję increment:

```
// hooks/useCounter.ts
import { useState } from 'react';
export function useCounter(initialValue: number = 0) {
  const [count, setCount] = useState(initialValue);
  const increment = () => setCount(c => c + 1);
```

```
  return { count, increment };
}
```

Test przy użyciu React Hooks Testing Library:

```
import { renderHook, act } from '@testing-library/react-hooks';
import { useCounter } from '../hooks/useCounter';

test('inicjalizuje licznik i inkrementuje', () => {
  const { result } = renderHook(() => useCounter(5));
  // result.current to { count, increment }
  expect(result.current.count).toBe(5);
  act(() => {
    result.current.increment();
  });
  expect(result.current.count).toBe(6);
});
```

Funkcja renderHook wywołuje nasz hook i zwraca obiekt z polem result zawierającym aktualny wynik hooka (pod result.current). Możemy sprawdzić początkową wartość, następnie użyć act() by wykonać akcję zmieniającą stan (wywołanie increment) i potem asercję, że stan się zmienił zgodnie z oczekiwaniem. Użycie act jest wymagane, aby symulować poprawnie cykl renderów (to taki mechanizm Testing Library mówiący „to jest zmiana stanu, ogarnij prze-render"). Ten test upewnia nas, że nasz hook działa poprawnie izolując logikę od komponentu.

Oczywiście hooki, które korzystają z kontekstu lub innych hooków RN mogą wymagać owinięcia w odpowiedni provider podczas testu (RNTL render ma opcję wrappera). Ale w prostych przypadkach takie podejście jest wystarczające.

2.3 Testy E2E – narzędzie Detox

Testy jednostkowe i komponentów pokrywają nasz kod „wewnątrz" aplikacji, ale nie dają 100% pewności, że aplikacja jako całość działa poprawnie na urządzeniu. Tutaj do gry wchodzą **testy end-to-end (E2E)**, które symulują prawdziwe scenariusze użycia aplikacji na fizycznym urządzeniu lub emulatorze. W ekosystemie React Native de facto standardem do E2E jest biblioteka **Detox** (rozwijana przez Wix).

Czym jest Detox? Detox to framework do automatyzacji testów E2E zaprojektowany specjalnie z myślą o aplikacjach mobilnych (szczególnie React Native). W przeciwieństwie do podejść czysto czarnej skrzynki (np. Appium), Detox jest **narzędziem *gray-box*** – oznacza to, że testy uruchamiane są na prawdziwej aplikacji (tak jak użytkownik by ją używał), ale Detox ma wgląd w wewnętrzny stan aplikacji, co pozwala mu lepiej synchronizować akcje z tym, co się dzieje wewnątrz. Dzięki temu testy są bardziej stabilne: framework **czeka na bezczynność aplikacji** (idle) zanim wykona kolejne kroki – np. poczeka aż zakończą się animacje, requesty sieciowe, render komponentów itp., zanim np. spróbuje kliknąć przycisk.

Jak to działa od strony technicznej? Detox integruje się z natywnymi frameworkami testów UI – na iOS używa XCUITest, na Androidzie Espresso. Różnica jest taka, że normalnie te frameworki są czarną skrzynką (tylko klikają i patrzą na UI), a Detox dodaje warstwę komunikacji z wnętrzem RN. Działa to tak, że nasza aplikacja RN jest uruchamiana ze

specjalną biblioteką Detox, która raportuje do testera stan (np. „jeszcze coś się renderuje, jeszcze działa pętla animacji”). Testy piszemy w JavaScript (uruchamiane przez Jest lub Mocha), one komunikują się z aplikacją przez most Detox. Schemat wygląda mniej więcej tak:

- Uruchamiamy detox test – buduje się aplikacja (wersja testowa) i odpalany jest runner testów.
- Detox odpala aplikację na emulatorze/symulatorze. Testy czekają, aż aplikacja będzie gotowa.
- Następnie każda testowa **akcja** (tap, wpisanie tekstu, scroll) jest wysyłana do natywnej strony (Espresso/XCUITest), która wykonuje ją na urządzeniu.
- Z kolei każda **asercja** (sprawdzenie, czy jakiś element widać) działa tak, że Detox potrafi na podstawie *matcherów* (np. tekstu, id) znaleźć element w hierarchii UI aplikacji.
- **Synchronizacja:** Detox automatycznie przed wykonaniem akcji/asercji upewnia się, że aplikacja jest gotowa – np. nie wykonuje tap, dopóki aplikacja nie jest idle (nie wykonuje animacji, operacji JS). To praktycznie eliminuje konieczność stosowania sleep() czy ręcznych oczekiwań – coś, co często dręczy testy E2E.
- Wyniki (sukces/porażka) przekazywane są do frameworka testów i możemy je zobaczyć w konsoli.

Pisanie testów z Detox: Testy tworzymy podobnie jak zwykłe testy, np. e2e/login.spec.js. Detox udostępnia globalne obiekty do interakcji:

- device – kontrola urządzenia (np. device.reloadReactNative() aby zresetować aplikację między testami, albo device.rotateScreen()).
- element(by.matcher(...)) – wybór elementu na ekranie za pomocą matchera (np. by.id('loginButton'), by.text('Hello'), by.label('Password')).
- Akcje na elemencie: po wybraniu elementu można chainować .tap(), .typeText(), .clearText(), .scroll() itd.
- Asercje: await expect(element(by.id('something'))).toBeVisible() albo .toHaveText('...') itp.

Detox wymaga, by nasze elementy w aplikacji były *dostępne do zidentyfikowania*. Najlepiej nadawać im atrybut testID (RN ma taki props dla wszystkich podstawowych komponentów). Np. definując <Button testID="loginButton" .../> możemy w teście użyć element(by.id('loginButton')) do znalezienia tego przycisku. To jest kluczowe – bez odpowiednich testID testy musiałyby polegać na tekście lub strukturze widoku, co bywa zawodne i podatne na zmiany UI. **Dlatego dobra praktyka:** wszystkie interaktywne elementy (przyciski, pola tekstowe) i kluczowe wyświetlane informacje wstawiajmy z testID podczas tworzenia komponentu, by ułatwić testy E2E.

Przykładowy scenariusz testu E2E: Rozważmy prosty przepływ logowania. Mamy ekran logowania z polami login, hasło i przyciskiem "Log In". Po zalogowaniu przechodzimy do ekranu "Second Screen". Przykładowy test w Detox (pseudo-kod zbliżony do prawdziwego):

```
describe('Flow logowania', () => {
  beforeAll(async () => {
    await device.launchApp(); // uruchomienie aplikacji
```

```
 });

 beforeEach(async () => {
  await device.reloadReactNative(); // reset stanu przed każdym testem, jeśli potrzebne
 });

 it('powinien zalogować użytkownika poprawnymi danymi', async () => {
  await expect(element(by.text('Log In'))).toBeVisible();      // sprawdź czy jesteśmy na ekranie logowania
  await element(by.id('LoginInput')).typeText('Admin');       // wpisz login (pole tekstowe ma
testID="LoginInput")
  await element(by.id('PasswordInput')).typeText('password');   // wpisz hasło
  await element(by.text('Log In')).tap();                    // naciśnij przycisk "Log In"
  await expect(element(by.text('Second Screen'))).toBeVisible(); // oczekuj przejścia do drugiego ekranu
 });

 it('powinien wyświetlić błąd przy pustych polach', async () => {
  await element(by.text('Log In')).tap(); // bez wpisywania niczego, klikamy logowanie
  await expect(element(by.id('LoginInputError'))).toBeVisible();    // pole login powinno wyświetlić błąd
(zakładamy, że komponent pola ma <Text testID="LoginInputError"> przy błędzie)
  await expect(element(by.id('PasswordInputError'))).toBeVisible(); // to samo dla hasła
 });
});
```

Powyższy kod (w uproszczeniu) pokazuje, jak test E2E loguje się i sprawdza nawigację oraz walidację. Zwróć uwagę na asercje: `expect(element(by.text('Log In'))).toBeVisible()` – Detox sprawdza, czy istnieje element z tekstem "Log In" i czy jest widoczny na ekranie (co oznacza, że ekran logowania jest aktywny). Wpisywanie tekstu `typeText` symuluje klawiaturę na urządzeniu – w tle do aplikacji lecą zdarzenia naciskania klawiszy. `tap` symuluje tapnięcie w ekran w miejscu elementu. Dzięki `await` i wewnętrznej synchronizacji Detox automatycznie czeka na zakończenie każdej akcji. Np. `await element(by.text('Log In')).tap()` poczeka, aż akcja tap zostanie wykonana i ewentualna nawigacja zakończona, zanim przejdzie dalej.

Detox – konfiguracja i uruchomienie: Aby użyć Detox, musimy doinstalować paczkę `detox` oraz dostosować projekt natywny (zwłaszcza iOS) – np. dodając do `Podfile` bibliotekę Detox. W pliku `package.json` konfigurujemy różne *konfiguracje* Detox, np. dla iOS simulator, dla Android emulator. Przykład konfiguracji (fragment):

```
"detox": {
 "testRunner": "jest",
 "configurations": {
  "ios.sim.debug": {
   "binaryPath": "ios/build/Build/Products/Debug-iphonesimulator/MyApp.app",
   "build": "xcodebuild -workspace ios/MyApp.xcworkspace -scheme MyApp -configuration Debug -sdk
iphonesimulator -derivedDataPath ios/build",
   "type": "ios.simulator",
   "device": { "type": "iPhone 14" }
  }
 }
}
```

Analogicznie dla Androida (Gradle build i apk path). Następnie uruchamiamy: `detox build -c ios.sim.debug` (buduje aplikację testową) i `detox test -c ios.sim.debug` (uruchamia testy). To dość złożony temat, ale szczegóły są w dokumentacji Detox. Warto wspomnieć, że można

integrować Detox z CI (np. popularne jest użycie go na platformach ciągłej integracji, a Codemagic ma nawet wsparcie wprost).

Zalety Detox:

- Jest szybki i stabilny w porównaniu do np. Appium, bo jest dostosowany do RN i korzysta z *gray-box* podejścia.
- Testy odpalają się bez konieczności ręcznej instrumentacji (wszystko skryptowo).
- Wspiera zarówno iOS jak i Android w jednym frameworku (piszemy test raz, działa na obu).
- Pozwala testować autentyczne scenariusze: od startu appki, przez przechodzenie między ekranami, po integrację z native (np. można symulować powiadomienia push, różne stany aplikacji).
- Integruje się z Jest – więc można w jednym repo mieć i testy jednostkowe, i E2E, korzystając ze znajomego runnera.

Wyzwania i dobre praktyki: Testy E2E są najwolniejsze i najbardziej złożone w utrzymaniu, więc zwykle nie piszemy ich bardzo dużo – pokrywamy kluczowe ścieżki (happy pathy i kilka edge case'ów). Trzeba dbać o unikanie „flakiness" – czyli sytuacji, gdzie test raz przechodzi, raz nie. Detox w dużej mierze to rozwiązuje synchronizacją, ale programista musi pamiętać o testID i unikalnych selektorach, a czasem o czyszczeniu stanu (stąd np. device.reloadReactNative() aby każda scena startowała od czystego stanu). Ważne jest też stubowanie ewentualnych zewn. zależności – np. jeśli aplikacja podczas E2E trafia na niestabilne API, można w testach zastosować tzw. *mock server* albo włączać specjalny tryb, gdzie aplikacja zamiast prawdziwych requestów używa lokalnych danych (Detox umożliwia np. nadpisanie fetch, lub można stawiać serwer stubów na localhost).

Podsumowując, Detox jest potężnym narzędziem, które automatyzuje testy z perspektywy użytkownika i **zapewnia, że aplikacja faktycznie działa poprawnie jako całość**. Użycie go znacząco poprawia pewność co do jakości.

3. Debugowanie React Native – Flipper i React DevTools

Nawet najlepsza architektura i testy nie uchronią nas przed koniecznością **debugowania** podczas tworzenia aplikacji. Debugowanie w React Native bywa wyzwaniem, ponieważ mamy do czynienia zarówno z kodem JavaScript, jak i natywnym oraz ich komunikacją przez most. Na szczęście istnieją narzędzia, które znacznie ułatwiają diagnozowanie problemów, podgląd stanu aplikacji oraz analizę wydajności. Skupimy się na dwóch kluczowych: **Flipper** – rozbudowane, oficjalnie wspierane narzędzie debugowe od Meta, oraz **React DevTools** – narzędzie do inspekcji drzewka komponentów React i profilowania renderowania.

3.1 Flipper – wszechstronne narzędzie debugowania aplikacji mobilnych

Flipper to desktopowa aplikacja stworzona przez Facebook/Meta, która służy jako platforma do debugowania aplikacji mobilnych (Android, iOS) – z naciskiem na React Native, ale obsługuje też inne technologie. Od React Native 0.62 Flipper jest domyślnie zintegrowany z RN (w trybie debug). Oznacza to, że uruchamiając naszą aplikację w trybie deweloperskim,

możemy podłączyć się do niej przez Flipper bez dodatkowej konfiguracji (czasem wymaga to tylko zainstalowania paczki react-native-flipper i odpalenia flippera w tle).

Kluczowe możliwości Flippera: Flipper ma architekturę wtyczek, gdzie każda wtyczka oferuje podgląd innego aspektu aplikacji. Najważniejsze w kontekście debugowania RN to:

- **Layout Inspector (Inspektor widoku)** – wizualizacja hierarchii widoków aplikacji i ich właściwości. Działa to podobnie do narzędzia "Inspect" w przeglądarce czy w Android Studio: możemy podejrzeć strukturę UI (widoki, komponenty, ich wzajemne ułożenie), zaznaczać elementy na ekranie urządzenia i przeglądać ich style/propsy. Jest to niezwykle przydatne, gdy np. jakiś element nie wyświetla się lub ma złe wymiary – w Flipperze zobaczymy cały widok i łatwiej zlokalizujemy problem (np. zero-height view, overlapping element etc.). Flipper umożliwia nawet edycję niektórych wartości na żywo, co pomaga eksperymentalnie znaleźć poprawkę.
- **Network Inspector (Podgląd sieci)** – monitorowanie wszystkich zapytań HTTP wychodzących z aplikacji. W zakładce Network Flipper przechwytuje requesty i responsy (wraz z nagłówkami, ciałem) wysyłane przez aplikację. Dzięki temu możemy sprawdzić np. czy zapytanie do API faktycznie się wysyła, jaki jest adres, co wraca z serwera, ile trwało, jaki był status. To wybawia od konieczności dodawania logów w kodzie czy używania proxy – Flipper działa jak swoisty *sniffer* ruchu sieciowego aplikacji RN. Można filtrować po URLach, metodach, itp. W kontekście debugowania problemów z komunikacją z backendem, ta wtyczka jest bezcenna.
- **Log Viewer (Konsola logów)** – zbiorczy podgląd logów natywnych i JavaScript. Zamiast patrzeć na logcat (Android) czy Xcode console oraz dodatkowo na console.log z debuggera RN, Flipper pokazuje wszystko w jednym miejscu. Widzimy logi z urządzenia (np. błędy natywne, wyjątki w JNI) jak i te wypisane przez console.log w JavaScripcie. Działa to bez dodatkowej konfiguracji. Jeśli aplikacja nam się wysypuje bez wyraźnego komunikatu w RN, warto spojrzeć do Flipper Logs – może tam widać stacktrace z native.
- **React DevTools** – Flipper ma wbudowaną integrację z React DevTools (pod spodem uruchamia najnowszą wersję DevToolsów Reactowych). Dzięki temu w Flipperze możemy przełączyć się na zakładkę React i tam dostać widok drzewa komponentów React (tych naszych, wysokopoziomowych, a nie niskopoziomowych widoków native jak w Layout Inspector). To pozwala debugować np. stan komponentów, propsy przekazywane do nich, hierarchię komponentów. Mamy też dostęp do **React Profiler** – czyli możliwość nagrania przebiegu renderowania i zobaczenia, które komponenty się renderują i ile to trwa. To świetne narzędzie do optymalizacji – można wykryć niepotrzebne ponowne rendery, zobaczyć gdzie aplikacja spędza czas podczas renderingu. (React DevTools i profiler są identyczne jak w aplikacjach webowych React – po prostu podłączone do kontekstu RN).
- **Crash Reporter** – Flipper może przechwycić raporty crashy aplikacji (zwłaszcza na Androidzie). Jeżeli aplikacja ulegnie nagłemu zamknięciu, Flipper wyświetli stacktrace i informacje o wyjątku. To pomaga zidentyfikować np. problem w kodzie natywnym lub w module RN (np. wołamy coś co powoduje NullPointerException w Javie – Flipper pokaże to).
- **Database & Preferences** – Wbudowane wtyczki pozwalają podejrzeć lokalną bazę danych (SQLite) i AsyncStorage w urządzeniu. W trybie debug możemy zajrzeć np. do

bazy SQLite używanej przez aplikację (przeglądać tabele, rekordy), a także do AsyncStorage (klucze i wartości). To przyspiesza debugowanie np. problemów z zapisem ustawień lub cachowaniem danych lokalnie.
- **Performance** – Domyślnie Flipper posiada prosty **FPS monitor** (w prawym górnym rogu aplikacji jest przełącznik perf, który pokazuje aktualne FPS). Jednak pełniejsze dane wydajnościowe dostarcza wtyczka **React Native Performance** (znana też jako *Flashlight*). Trzeba ją doinstalować (z marketplace Flippera). Daje ona wykresy użycia CPU (całościowo i per wątek), liczbę klatek na sekundę, zużycie pamięci itp.. Możemy zatem profilować wydajność natywną aplikacji w czasie rzeczywistym – np. czy podczas przewijania listy CPU skacze do 100%, czy spada FPS (co by wskazywało na lagi). Wtyczka Performance może pomóc w identyfikacji wąskich gardeł wydajnościowych.

Flipper jest stale rozwijany i ma mnóstwo dodatkowych pluginów tworzonych przez społeczność (dostępnych w marketplace Flippera). Przykładowo pluginy do debugowania połączeń WebSocket, do podglądu bieżącego stanu Redux store, czy do symulowania różnych warunków sieciowych.

Jak korzystać z Flippera? W praktyce: instalujemy aplikację Flipper na komputerze (ze strony fbflipper.com). Upewniamy się, że nasza aplikacja RN ma włączony Flipper (w RN 0.62+ jest to domyślne w Debug, jeśli nie, to `npm i react-native-flipper` i drobna konfiguracja w native jak w dokumentacji). Odpalamy aplikację na emulatorze lub urządzeniu *w trybie debug (Metro)*. Flipper automatycznie wykryje aplikację (o ile jesteśmy na tej samej sieci) i pokaże ją na liście. Po kliknięciu, uzyskamy dostęp do pluginów (część może wymagać doinstalowania paczek – Flipper poinformuje). Potem już normalnie używamy: klikamy „Layout", by zbadać UI, itp.

Flipper w znacznym stopniu zmniejsza potrzebę używania np. Xcode Instruments czy Android Studio profilerów dla wielu typowych zadań, agregując debugowanie w jednym miejscu. To oszczędza czas – nie musimy np. w przypadku problemów sieciowych budować własnego logowania, tylko patrzymy do Network plugina. Dodatkowo minimalizuje *context switching*: wszystko (UI, logi, sieć, stan) jest w jednym oknie.

Przykład użycia: Załóżmy, że przycisk w naszym UI nie reaguje na tapnięcia. Co możemy zrobić:

- W Flipper -> Logs sprawdzimy, czy w ogóle event dotarł (może w logach `onPress` coś logujemy).
- W Layout Inspector zobaczymy hierarchię – być może inny przezroczysty widok nakrywa przycisk i przechwytuje dotyk (częsty bug z overlayami). Inspektor pokaże, czy przycisk jest klikalny czy zakryty.
- Możemy też w React DevTools sprawdzić, czy props `onPress` faktycznie został przekazany do komponentu.
- Jeśli nic nie pomaga, w Flipper można też wykorzystać funkcję przechwytywania logów dotyku (plugin Touch Events – doinstalowany plugin społeczności).
- W efekcie, diagnoza problemu jest dużo szybsza niż metodą `alert('x')` czy zgadywaniem.

Podsumowując, **Flipper** to “szwajcarski scyzoryk” debugowania RN: *layout, network, logi, stan, wydajność – wszystko pod ręką*. W 2025 jest to podstawowe narzędzie dla React Native devów, wspierane oficjalnie. Jak mówi dokumentacja, Flipper umożliwia **inspekcję hierarchii widoków, monitorowanie zapytań sieciowych, przeglądanie logów z urządzenia i integrację z React DevTools** w czasie rzeczywistym. Dzięki temu debugowanie staje się dużo wygodniejsze i skuteczniejsze.

3.2 React DevTools – inspekcja komponentów i profilowanie

Choć React DevTools jest zintegrowane we Flipperze, warto omówić jego możliwości bardziej szczegółowo, bo dotyczy stricte *warstwy Reactowej* aplikacji.

React DevTools to narzędzie, które wielu zna z debugowania aplikacji webowych React (np. jako rozszerzenie do Chrome). W przypadku RN, od wersji 0.62 wzwyż, możemy korzystać z DevTools poprzez Flipper lub poprzez React Native Debugger (osobna aplikacja). DevTools pozwala:

- **Przeglądać drzewo komponentów React** – widzimy jakie komponenty (funkcyjne, klasowe) są zamontowane, ich hierarchię (co jest dzieckiem czego). Dla każdego wybranego komponentu możemy podejrzeć **propsy** jakie dostał, **stan** (dla komponentów klasowych lub hook useState), a także **wartości hooków** (DevTools pokaże np. że nasz hook useCounter ma wartość count = 5, it = function etc.). To bardzo pomaga zrozumieć co się dzieje: np. czy rodzic przekazał właściwą wartość prop, czemu dziecko ma stan X, itp. Możemy dynamicznie edytować propsy lub stan w DevTools, co może pomóc testować różne scenariusze bez przebudowy aplikacji.
- **Wykonywać funkcje hooków debugowo** – nowa wersja DevTools umożliwia np. wywołanie manualnie funkcji aktualizującej stan (w hooks). Np. jeśli mamy useState counter, możemy w DevTools kliknąć aby zwiększyć/zmniejszyć wartość. To drobnostka, ale czasem przydatna.
- **Profilować renderowanie** – React DevTools posiada zakładkę **Profiler**, gdzie możemy nagrać przebieg renderów podczas wykonywania pewnych akcji w aplikacji. Po zatrzymaniu nagrywania otrzymujemy timeline z zaznaczonymi renderami komponentów i informacją, ile milisekund zajęło renderowanie danego poddrzewa. Komponenty, które renderują się często lub długo, będą zaznaczone (np. ciepłe kolory jeśli dużo czasu). W profilowaniu RN jest to cenne do optymalizacji: np. wykryjemy, że przy przewijaniu listy, cały ekran się renderuje 60 razy na sekundę – może brakuje `React.memo` na jakimś dużym komponencie? Albo że po zmianie jednego pola, niepotrzebnie 10 innych niezmienionych komponentów się przerysowuje. Profiler wskaże nam *co* i *jak długo*. Potem można użyć takich informacji, by zastosować memoizację, podział na mniejsze komponenty lub przesunięcie obliczeń poza render.
- **Debugging performance w hooks** – DevTools jest też świadomy Hooków, więc np. pokaże w jakiej kolejności hooki były wywoływane, co może pomóc w zrozumieniu czy np. `useEffect` nie jest odpalany zbyt często.

Warto zaznaczyć, że aby React DevTools działało, nasza aplikacja musi być w trybie debug (połączona z Metro). W trybie release nie mamy tego narzędzia (chyba że budujemy specjalnie z DevSettings). Z Flipperem to proste, bo Flipper sam zadba o połączenie.

Inne narzędzia debugowania: Wspomnijmy krótko, że istnieją też alternatywne lub uzupełniające narzędzia:

- **React Native Debugger** – osobna aplikacja (open-source) integrująca DevTools, debugger JS i Redux DevTools. Można jej używać zamiast Flippera, zwłaszcza jeśli korzystamy z Redux – bo ma wbudowany podgląd stanu Redux z możliwością time-travel debuggingu. Jednak Flipper ostatnio zyskuje większą popularność, bo jest oficjalny i pluginowy.
- **Chrome Debugger** (stare podejście) – RN kiedyś debugowało JS przez wbudowany mechanizm z Chrome. Obecnie w dobie Hermes (silnik JS) to się zmieniło – debugowanie JS odbywa się inaczej (Remotely w Hermes debugging), ale można też używać Chrome devtools do debuggera kodu (wbijania w breakpointy JS).
- **LogBox** – to nie narzędzie zewnętrzne, ale wbudowany mechanizm RN pokazujący elegancko błędy i warningi w UI aplikacji podczas dev. Wspominam, bo debugowanie to też reagowanie na warny/errory – RN LogBox (od RN 0.63) bardzo to ułatwia (konsola w appce z możliwością filtrowania i ignorowania komunikatów).

Konkludując sekcję debugowania: **dobry zestaw narzędzi debugowych** to Flipper + React DevTools, które razem pozwalają zajrzeć w niemal każdy aspekt aplikacji:

- od strony natywnej (layout, logi, zasoby, sieć) – Flipper,
- od strony React (stan komponentów, cykl renderów) – DevTools.

Z takim „arsenałem" typowe problemy (UI się nie układa, requesty nie dochodzą, coś się wiesza) rozwiążemy znacznie szybciej i pewniej niż metodą prób i błędów.

4. Wydajność aplikacji React Native

Wydajność to ważny temat – chcemy, aby nasza aplikacja działała płynnie (60 klatek na sekundę lub więcej), reagowała natychmiast na interakcje i efektywnie gospodarowała pamięcią, nawet na słabszych urządzeniach. Omówimy techniki optymalizacji związane głównie z renderowaniem list (FlatList i jego konfiguracja), unikaniem zbędnych renderów/layoutów oraz usprawnianiem animacji (wprowadzenie do biblioteki Reanimated, animacje natywne vs. „klatkowe").

4.1 Optymalizacja list – wykorzystanie FlatList i konfiguracja wydajnościowa

W aplikacjach mobilnych częstym wyzwaniem są **długie listy** elementów (np. feed aktualności, lista kontaktów, itp.). Nie możemy pozwolić sobie na renderowanie setek komponentów na raz – powodowałoby to ogromne obciążenie pamięci i spadki płynności. React Native oferuje komponent **FlatList** (oraz pokrewny SectionList) jako wydajne rozwiązanie do wyświetlania list poprzez *wirtualizację* (virtualized list). Główna idea: FlatList renderuje tylko te elementy, które mieszczą się w aktualnym widoku (plus pewien bufor

wokół), a resztę „usuwa” lub nie tworzy ich wcale, dopóki użytkownik nie scrolluje w ich kierunku.

Zasada działania FlatList (virtualized list): Pod spodem FlatList korzysta z mechanizmu VirtualizedList. Dzieli całą listę na tzw. *okno (window)*, które przemieszcza się wraz ze scrollowaniem. Elementy poza oknem mogą być odłączone (unmount) lub nawet nigdy nie zrenderowane, dopóki nie wejdą do okna. To drastycznie zmniejsza liczbę jednocześnie aktywnych elementów. Oczywiście FlatList wymaga pewnych informacji (np. wysokości elementów) by sprawnie to robić – stąd też ma wiele opcji konfiguracji.

Kluczowe właściwości FlatList wpływające na wydajność: (należy je dostosować do kontekstu użycia):

- **keyExtractor** – funkcja generująca unikalny klucz dla elementu listy, jeśli nasze dane nie mają właściwości key. Dlaczego to ważne? React używa kluczy by zoptymalizować elementy – unikalne klucze zapobiegają niepotrzebnym re-renderom, pozwalają śledzić elementy przy zmianach kolejności. Zwróćmy uwagę, by klucz był stabilny (np. item.id) i unikalny. Jeśli nie ustawimy keyExtractor, RN spróbuje użyć domyślnie item.key lub indeksu – używanie indeksu jest niewskazane, bo zmiana kolejności elementów spowoduje, że React potraktuje je jako inne i przemaluje całą listę. Dlatego **dobre klucze to podstawa wydajności listy**.
- **initialNumToRender** – ile elementów wyrenderować na starcie (domyślnie 10). Możemy to dostosować tak, by pokryć cały ekran na najpopularniejszych urządzeniach, ale nie za dużo ponad to. Np. jeśli na ekranie mieści się ~8 elementów, można dać initialNumToRender = 8 lub 12 (z zapasem). Zbyt mała wartość grozi pustym miejscem („white flash”) przy starcie listy, zbyt duża – niepotrzebnie obciąża start (np. ładuje 50 elementów, gdy ekran mieści 8).
- **windowSize** – rozmiar okna w jednostkach ekranu (domyślnie 21, co oznacza 10 ekranów powyżej i 10 poniżej aktualnie widocznego + bieżący). To parametr określający, ile elementów przed i za widocznym obszarem ma być utrzymywanych w stanie zrenderowanym. Większy windowSize zmniejsza ryzyko pojawienia się pustego obszaru przy szybkim scrollu (bo elementy już czekają tuż poza ekranem), ale zwiększa zużycie pamięci i czas renderowania (bo więcej elementów istnieje). Mniejszy windowSize oszczędzi pamięć, ale np. przy bardzo szybkim scrollowaniu może być widać doczytywanie elementów (puste przestrzenie). Optymalna wartość zależy od charakteru listy – jeżeli listę scrolluje się powoli, można zmniejszyć windowSize by zaoszczędzić zasoby; dla list scrollowanych energicznie (np. social feed) lepiej zostawić większe, by zachować płynność.
- **removeClippedSubviews** – ustawienie (domyślnie true na Androidzie, false na iOS) określające, czy usuwać (odłączać) widoki, które wyszły poza ekran (tzw. *clipped*, zaklippowane). Włączenie tej opcji (na iOS manualnie, bo na Android jest on już true by default) powoduje, że elementy które wyszły daleko poza widok są usuwane z hierarchii natywnej, co redukuje obciążenie GPU i CPU (nie są one uwzględniane w renderowaniu i dotyku). Zysk to mniejsze zużycie głównego wątku (bo mniej widoków do liczenia layoutu i rysowania). Wadą bywa to, że czasem potrafi powodować drobne bugi – np. w przeszłości raportowano, że na iOS potrafiły znikać niektóre elementy w niesprzyjających warunkach, jeśli były transformacje lub absolutne

pozycjonowanie. Ogólnie jednak warto włączyć removeClippedSubviews dla bardzo długich list, zwłaszcza jeżeli elementy listy są „ciężkie” w renderowaniu.

- **maxToRenderPerBatch** oraz **updateCellsBatchingPeriod** – te parametry kontrolują, jak FlatList dokonuje *dorzucania* elementów przy scrollowaniu. maxToRenderPerBatch (domyślnie 10) oznacza ile maksymalnie nowych elementów wyrenderować na jedno przebudzenie listy (czyli gdy zbliżamy się do końca aktualnie wyrenderowanego obszaru, ile kolejnych elementów doczepić). updateCellsBatchingPeriod (domyślnie 50ms) to opóźnienie między takimi porcjami renderów. Zwiększenie maxToRenderPerBatch zmniejszy szanse na puste miejsca (bo doczepiamy więcej naraz), ale może spowodować dłuższy jednorazowy lag (bo nagle 20 elementów się buduje). Zmniejszenie sprawi, że scroll jest bardziej responsywny (mniejsze bloki renderów), ale może pojawić się chwilowo pustka, jeśli użytkownik przewija szybciej niż te batch’e się doczepiają. updateCellsBatchingPeriod z kolei – mniejsza wartość = częściej dobudowujemy elementy (bardziej płynne, ale większe obciążenie CPU stale), większa = rzadziej (oszczędniej, ale ryzyko lagów). W praktyce rzadko się zmienia te wartości od domyślnych, chyba że profilowanie wykaże konkretny problem.
- **getItemLayout** – funkcja, którą możemy dostarczyć, jeśli nasze elementy listy mają stały, z góry znany rozmiar (wysokość). Dzięki temu FlatList może z góry wyliczyć pozycję scrolla i offsety, nie musi mierzyć elementów podczas renderowania. Ustawienie getItemLayout przy stałych wysokościach *znacząco* poprawia wydajność listy, bo RN nie musi każdego elementu renderować choćby raz żeby znać jego wysokość (eliminuje to tzw. *layout pass* dla offscreen elementów). W definicji getItemLayout wskazujemy: dla index -> {length, offset, index}. Np. jeśli każdy element ma wysokość 50, offset to index*50, length=50. FlatList użyje tego, by np. szybko przewinąć do elementu (scrollToIndex) bez błędu estimations.

Podsumowując, **FlatList** daje sporo możliwości tuningu. Dobrą praktyką jest profilowanie (np. w Flipper Perf monitor) scrollowania listy i ewentualna regulacja powyższych parametrów. Dokumentacja RN oficjalnie wymienia te propsy jako pomocne w poprawie wydajności. W skrócie:

- *Masz lagi przy scrollu?* – upewnij się, że removeClippedSubviews = true (szczególnie Android) i ewentualnie zmniejsz maxToRenderPerBatch.
- *Masz „blank areas” (puste dziury) przy szybkim scrollu?* – zwiększ windowSize lub initialNumToRender, ewentualnie maxToRenderPerBatch.
- *Aplikacja zjada za dużo pamięci przy liście?* – zmniejsz windowSize, removeClippedSubviews na true i postaraj się zmniejszyć złożoność renderowanych elementów.

Wskazówki co do elementów listy: Nawet najlepiej skonfigurowana FlatList może zwalniać, jeśli pojedyncze elementy listy (komponenty list item) są „ciężkie”. Kilka porad:

- Upewnij się, że twoje komponenty listy są **jak najprostsze i lekkie**. Unikaj w nich skomplikowanych poddrzew z wieloma warunkami renderowania. Każdy element listy renderowany jest wiele razy (przy scrollowaniu), więc ich optymalność jest krytyczna.

- Rozważ użycie React.memo dla komponentu elementu listy, aby nie re-renderował się, jeśli propsy się nie zmieniły. W połączeniu z odpowiednimi key to spowoduje, że przy dodawaniu nowych elementów, już wyrenderowane nie będą bez potrzeby odświeżane.
- Jeśli lista ma obrazy, stosuj **miniaturki** (thumbnails) lub mechanizmy lazy-load dla obrazków. Duże obrazki spowalniają scroll (zajmują czas dekodowania i miejsce w pamięci). Lepiej wyświetlać mniejsze wersje, a dopiero po wejściu np. w szczegóły ładować duży obrazek.
- Jeżeli można, **paginacja**: zamiast trzymać 1000 elementów naraz, ładuj je w porcjach (np. 50) i doładowuj kolejne, gdy użytkownik zbliża się do końca (FlatList oferuje props onEndReached do tego celu). To zmniejsza rozmiar listy w jednym momencie.

4.2 Unikanie zbędnych renderów i przesunięć układu (layout shifts)

Layout shifts to określenie znane z web (Cumulative Layout Shift) – chodzi o nagłe przeskoki układu gdy elementy się zmieniają. W kontekście RN nie liczymy punktów za stabilność layoutu, ale również chcemy unikać sytuacji, gdzie interfejs „skacze" lub gdzie wykonujemy kosztowne operacje layoutu bez potrzeby. Kilka rad:

- **Stałe wymiary lub użycie <FlatList> zamiast manualnego mapowania.** Gdy korzystamy z FlatList, RN wie, że to lista i zarządza layoutem wydajnie. Jeśli sami w komponencie robimy data.map(item => <MyItem ...>) wewnątrz ScrollView, to tracimy wirtualizację, a także możemy powodować częste przeliczanie layoutu (ScrollView renderuje wszystko naraz). Więc podstawowa rada: używaj FlatList dla większych list. Jeśli z jakiegoś powodu nie możesz (np. potrzebujesz nietypowego layoutu), rozważ <VirtualizedList> lub sekcje.
- **Unikaj niepotrzebnego stanu powodującego globalny re-render.** Np. jeśli mamy duży ekran z listą i drobny toggle, starajmy się, by zmiana toggle nie powodowała przerysowania całej listy. Można to osiągnąć np. wydzielając listę do osobnego komponentu i stosując React.memo lub używając dedykowanego stanu (np. Redux slice) tak, by zmiana niezwiązana z listą nie dotykała jej. Ogólnie, trzymanie globalnego stanu minimalnego i raczej lokalnych stanów dla UI elementów pomoże ograniczyć zakres renderów.
- **Animacje układu:** Jeżeli dodajemy/usuwamy elementy z DOM, to oczywiście układ się zmieni. Można to złagodzić używając np. *Layout Animations*, które płynnie przeprowadzą zmianę, albo planując UI tak, by duże zmiany zachodziły w momentach, gdy użytkownik się ich spodziewa (np. przejście na inny ekran, zamiast dynamicznie na tym samym ekranie).
- **Re-render a listy:** Bardzo częsty powód spowolnień – komponent rodzica listy renderuje się często (np. z powodu zmiany stanu zegara, albo czegokolwiek) i za każdym razem dostaje nową data listy (np. nowa referencja tablicy), co sprawia że FlatList myśli, że dane są „inne" i przerysowuje część elementów. Aby temu zapobiec:
    - Jeżeli generujemy data w renderze, np. const data = items.filter(...).map(...);, to warto albo użyć useMemo do memoizacji wyniku, albo przenieść to wyżej i przekazywać gotowe data z rodzica, który nie renderuje się tak często.

    - FlatList ma też prop extraData do śledzenia dodatkowych zmian – jeśli nie potrafimy powiedzieć, co się zmieniło, ale wiemy, że np. state X wpływa na listę, można to tam włożyć. Jednak lepiej kontrolować referencje danych.
- **Wielkość widoku**: Upewnij się, że stylujesz listy i elementy tak, by nie wymagały skomplikowanego *layout calculation* przez Yoga (silnik layout RN). Np. zbyt zagnieżdżone układy flex z komponentami którymi sterują zmiany dynamiczne mogą obciążać CPU. W profilowaniu (Flipper Perf -> CPU usage) widać, gdy po jakiejś akcji CPU rośnie bo wątek UI liczy layout – to sygnał, że może albo zbyt duża część ekranu się zmienia, albo styl jest suboptymalny (np. dużo shadowOffset i cieni może obciążać GPU/CPU przy przesuwaniu elementów).
- **Usuwanie elementów z DOM**: Jeżeli coś jest niewidoczne, można to usunąć zamiast chować (np. usuwaj ekrany modali gdy są zamknięte, zamiast trzymać je wszystkie w DOM z display:none). RN co prawda nie ma display:none – jak coś nie renderujesz, to jest usunięte. Ale w nawigacji np. stos pamięta ekrany. W przypadku np. TabNavigatora – ekrany zakładek mogą być domyślnie „lazy" czyli montowane na żądanie, co jest ok.

Ogólnie, *unikanie zbędnych renderów* sprowadza się do stosowania wspomnianych technik: **memoizacji** (React.memo, useMemo, useCallback) – by nie generować nowych referencji gdy nie trzeba, **separacji stanów** – by lokalna zmiana nie wpływała globalnie, oraz **profilowania** – by wiedzieć gdzie jest problem. React DevTools Profiler tutaj jest najlepszym przyjacielem: pozwoli zobaczyć np. że komponent lista renderuje się 5 razy podczas jednej akcji, co nie jest potrzebne.

4.3 Animacje klatkowe vs animacje natywne – wprowadzenie do Reanimated

Animacje w aplikacjach mobilnych mogą łatwo stać się wąskim gardłem wydajności, jeśli nie są wykonane prawidłowo. W React Native tradycyjnie mieliśmy bibliotekę **Animated** (API Animated API) oraz możliwość użycia LayoutAnimation czy InteractionManager. Jednak duża zmiana zaszła wraz z pojawieniem się biblioteki **React Native Reanimated** (szczególnie w wersji 2 i wyższych), która pozwala tworzyć płynne animacje wykonywane po stronie natywnej.

Najpierw wyjaśnijmy pojęcia z tematu: **animacje klatkowe** vs **animacje natywne**. Można to rozumieć tak:

- *Animacje klatkowe (frame-by-frame)* – tu rozumiem to jako animacje sterowane na każdej klatce przez JavaScript. Czyli nasz kod JS oblicza, gdzie powinien być obiekt w danej klatce i ustawia styl (np. zmienia left co 16ms). Takie animacje obciążają **wątek JavaScript** i są podatne na gubienie klatek, jeśli JS jest zajęty innymi rzeczami (np. obliczenia, renderowanie). Przykładem animacji klatkowej może być użycie setInterval do zmiany stanu, co powoduje re-render z lekko zmienionym położeniem – to skrajnie niewydajne, bo każda klatka to nowy render Reacta. Lub użycie Animated bez opcji natywnego drivera (w starszym RN useNativeDriver: false powoduje, że animacja dzieje się w JS – styl jest aktualizowany poprzez bridge co klatkę).
- *Animacje natywne* – animacje, które są wykonywane po stronie natywnej (na wątku UI lub innym natywnym), bez angażowania wątku JS na każdą klatkę. W praktyce znaczy to: z góry określamy przebieg animacji (np. „przesuń obiekt z X do Y w 500ms z

krzywą ease-out”), przekazujemy to do warstwy natywnej i tam to się odbywa płynnie, nawet jeśli JS się zatrzyma. W RN Animated klasyczny sposób to `useNativeDriver: true` – ale to miało ograniczenia (tylko niektóre właściwości, brak animacji koloru czy layoutu). Reanimated idzie dalej – pozwala definiować całą logikę animacji i reakcji na gesty jako *worklety* wykonywane na wątku UI.

React Native Reanimated (2.x i 3.x) – co czyni go wyjątkowym?
Jak podaje dokumentacja: *„Reanimated pozwala definiować animacje w czystym JavaScript, które **domyślnie uruchamiają się natywnie na wątku UI**”*. Oznacza to, że piszemy kod animacji w JS, ale dzięki mechanizmowi *workletów* (specjalnych funkcji z dopiskiem `'worklet'`) kod ten zostaje wysłany do natywnego środowiska i tam wykonywany co klatkę, bez obciążania mostu czy JS thread. To daje **gładkie animacje do 60fps, a nawet 120fps** na ekranach je wspierających, niezależnie od obciążenia JS.

Klatkowe vs natywne – efekty praktyczne: Animacja klatkowa (JS-driven) może zacząć gubić klatki, jeśli w tym samym czasie JS robi coś ciężkiego. Np. wyobraźmy sobie animację wysuwania panelu, a jednocześnie obsługujemy duży JSON z API na JS thread – animacja może przycinać, bo JS nie nadąża co 16ms wysłać nowej pozycji. W animacji natywnej (Reanimated) taka sytuacja nie wpływa – panel się wysunie płynnie, bo logika ruchu jest odseparowana. Dlatego wszystkie istotne animacje w UI powinny być natywne, by zapewnić płynność. Dotyczy to także reakcji na gesty – np. płynne przeciąganie elementu palcem: Reanimated w połączeniu z React Native Gesture Handler potrafi przenieść obsługę gestu i animacji całkowicie na natywną stronę, eliminując lagi.

Przykład różnicy: Weźmy animację prostą – przesunięcie kwadratu 100px w prawo. W czystym RN Animated (do wersji RN 0.71 około):

```
Animated.timing(position, {
  toValue: 100,
  duration: 500,
  useNativeDriver: false // (JS-driven)
}).start();
```

Ta animacja co klatkę (co ~16ms) wyśle nową wartość `position` przez Bridge do native. Jeśli Bridge się zapcha albo JS spóźni – będzie skok. Gdy damy `useNativeDriver: true`:

```
Animated.timing(position, {
  toValue: 100,
  duration: 500,
  useNativeDriver: true
}).start();
```

to RN prześle do natywnego modułu Animated informację "animuj tę wartość od 0 do 100 w 500ms", a natywny kod (Core Animation na iOS / Android Animator) zrobi resztę. To już jest natywna animacja i powinna być płynna. Problem w tym, że stare Animated z native driver działało tylko dla niektórych stylów (głównie translacje, skala, opacity). Nie można nim animować np. koloru tła czy położenia zależnego od układu Flexbox.

Reanimated nie ma takich ograniczeń, bo działa inaczej: mamy *shared values* i *worklets*, można animować dowolne style, bo tak naprawdę animacja to po prostu funkcja zmieniająca wartość, a potem przypisanie jej do stylu w natywnym Shadow Tree. Reanimated 2+ integruje się z mechanizmem UI runtime. W efekcie, możemy np. animować pozycję zależnie od wartości z czujników (accelerometr), możemy robić złożone sekwencje i zagnieżdżone animacje – to wszystko w natywnym kontekście.

W kontekście wydajności:

- Animacje natywne (Reanimated, lub Animated native driver) praktycznie nie obciążają JS podczas trwania. Obciążają natywny wątek UI, ale ten jest w C/ C++ i bardzo wydajny, plus może wykorzystywać optymalizacje platformy (np. iOS zrobi to w CoreAnimation).
- Animacje klatkowe (JS) obciążają JS i Bridge – dwa wąskie gardła RN. Lepiej ich unikać, bo nawet jeśli jedna działa, to przy wielu jednoczesnych będą się zawieszać.

W 2025, **Reanimated stał się de-facto standardem** dla skomplikowanych animacji i interakcji. Wiele bibliotek buduje na nim (np. znany bibliotek do dolnych arkuszy "react-native-bottom-sheet" używa Reanimated). Warto chociaż znać podstawy:

- Pojęcia *useSharedValue* (wartość współdzielona animowana),
- *useAnimatedStyle* (hook który pozwala powiązać styl komponentu z animowaną wartością),
- *zestawy animacji* typu withSpring, withTiming – funkcje do aktualizacji shared value z efektami animacji (sprężyna, timing).
- Worklety 'worklet' – czyli funkcje, które wykonują się na UI thread. Np. callback w .onChange gestu zaopatrzony w 'worklet' może bezpośrednio sterować sharedValue i to będzie natywne.

Krótki przykład Reanimated (wersja 3+):

```
import Animated, { useSharedValue, useAnimatedStyle, withSpring } from 'react-native-reanimated';
import { View, Button } from 'react-native';

export default function Box() {
 const offset = useSharedValue(0);
 const animatedStyle = useAnimatedStyle(() => {
  return {
   transform: [{ translateX: offset.value }]
  };
 });

 return (
  <View>
   <Animated.View style={[{ width: 100, height: 100, backgroundColor: 'red' }, animatedStyle]} />
   <Button title="Move" onPress={() => {
    // to 'animuje' natywnie, nie blokując JS
    offset.value = withSpring(offset.value + 100);
   }} />
  </View>
 );
```

}

Tutaj kliknięcie przycisku powoduje zmianę offset.value za pomocą withSpring. To nie wywoła od razu re-renderu React (Animated.View samo odbierze zmianę), lecz spowoduje zapoczątkowanie natywnej animacji sprężynowej – czerwona kostka przesunie się płynnie o 100px. Wątek JS tylko zainicjował animację, dalej dzieje się to w natywnym (UI) wątku. Gdy animacja się zakończy, Reanimated może zasygnalizować JS (ale nie musi, zależy). Co ważne, podczas ruchu możemy nawet zatrzymać JS (np. włączmy dev menu) – animacja i tak dokończy.

Animacje a klatki (FPS):

- Płynna animacja to 60 FPS (w 60Hz ekranach) albo 120 FPS (na iPad Pro np.).
- Animacja klatkowa jeśli JS nie nadąży, może spaść do 30 FPS lub mniej (widać wtedy "szarpanie").
- Reanimated stara się zapewnić 60 FPS nawet w trudniejszych scenariuszach. Oczywiście, jeśli wykonujemy *skrajnie* ciężkie rzeczy w worklecie (np. pętlę 1e6 iteracji co klatkę – co raczej się nie zdarza), to i natywnie można klatki gubić. Ale normalne transformacje, sprężyny – to jest nic dla nowoczesnych CPU, i do tego natywne animacje często korzystają z GPU do finalnego renderu.

Złożone animacje: Reanimated oprócz animacji pojedynczych stylów oferuje też *Layout Animations* (pozwala animować elementy podczas mount/unmount), integruje się z Gesture Handler (gesty też na UI thread – zero laga w drag & drop), i umożliwia tworzenie własnych wątków (tzw. *worklet runtime*) do dużych obliczeń w tle. To ostatnie to ciekawostka – można np. w worklecie liczyć coś ciężkiego (jak w przykładzie sum 1..1000000) i potem przekazać wynik do JS bez zamrażania aplikacji.

W ramach tego wykładu jest to **wstęp** do Reanimated – zachęcam do dalszej nauki, bo temat jest obszerny. Najważniejsze, co warto zapamiętać: **dążymy do animacji natywnych dla lepszej wydajności**. Reanimated to narzędzie, które to umożliwia w bardzo elastyczny sposób (możemy animować prawie wszystko co stylowe, reagować na gesty płynnie). W nowoczesnych aplikacjach RN, Reanimated często zastępuje starą bibliotkę Animated, choć RN nadal utrzymuje Animated API (być może w przyszłości zimplementowane w oparciu o Reanimated pod spodem – już teraz jest dyskusja, by spiąć te światy).

Podsumowanie tej sekcji: Animacje „klatkowe” (czyli zależne od JS co klatkę) powinny być ograniczone do minimum. Używamy mechanizmów natywnych – czy to starego useNativeDriver dla prostych przypadków, a najlepiej Reanimated dla pełnej kontroli – by zapewnić **płynność 60fps** niezależnie od obciążenia logiką aplikacji. Dzięki temu interfejs będzie odbierany jako nowoczesny i responsywny. A przecież nic tak nie psuje UX mobilnego jak animacje/skrollowanie szarpiące jak pokaz slajdów – tego chcemy uniknąć.

5. Demo – refaktoryzacja listy i test komponentu listy

Na koniec przeprowadźmy małe demo, które łączy kilka omawianych konceptów: pokażemy **refaktoryzację istniejącej listy** do użycia FlatList z optymalizacją wydajności, a następnie **napiszemy test** dla komponentu listy, sprawdzający renderowanie i interakcje.

Scenariusz: Załóżmy, że mamy aplikację do zadań (TODO list). Dotychczas lista zadań była zaimplementowana naiwnie – jako zwykły <ScrollView> z .map lub nawet w komponencie rodzicu. Chcemy to poprawić, użyć FlatList i zastosować pewne parametry wydajnościowe. Dodatkowo, nasza lista ma możliwość oznaczania zadania jako ukończonego po kliknięciu przycisku "Done" obok zadania. Przetestujemy, czy kliknięcie faktycznie wywołuje odpowiednią funkcję (np. przekazaną z props).

Kod przed refaktoryzacją (fragment):

```
// components/TaskListBefore.tsx
import { ScrollView, Text, View, Button } from 'react-native';

export function TaskListBefore({ tasks, onTaskDone }) {
  return (
    <ScrollView>
      {tasks.map(task => (
        <View key={task.id} style={styles.taskItem}>
          <Text style={{ textDecorationLine: task.done ? 'line-through' : 'none' }}>
            {task.title}
          </Text>
          {!task.done && (
            <Button title="Done" onPress={() => onTaskDone(task.id)} />
          )}
        </View>
      ))}
    </ScrollView>
  );
}
```

Ten kod działa dla małych list, ale jeśli tasks będzie duże, to ScrollView wyrenderuje wszystko na raz. Użyjmy FlatList:

```
// components/TaskList.tsx
import React from 'react';
import { FlatList, Text, View, Button, ListRenderItem } from 'react-native';

interface Task { id: string; title: string; done: boolean; }
interface TaskListProps { tasks: Task[]; onTaskDone: (id: string) => void; }

export function TaskList({ tasks, onTaskDone }: TaskListProps) {
  const renderItem: ListRenderItem<Task> = ({ item }) => (
    <View style={styles.taskItem}>
      <Text style={{ textDecorationLine: item.done ? 'line-through' : 'none' }}>
        {item.title}
      </Text>
      {!item.done && (
        <Button title="Done" onPress={() => onTaskDone(item.id)} testID={`done-btn-${item.id}`} />
```

```
    )}
  </View>
 );

 return (
  <FlatList
   data={tasks}
   keyExtractor={(item) => item.id}
   renderItem={renderItem}
   initialNumToRender={10}
   windowSize={5}  // 5 ekranów (2 przed, 2 za, 1 obecny)
   maxToRenderPerBatch={5}
   removeClippedSubviews={true}
  />
 );
}
```

Co się zmieniło:

- Używamy <FlatList> z data={tasks} i renderItem do określenia, jak renderować pojedyncze zadanie.
- Dodaliśmy keyExtractor={(item) => item.id}, aby FlatList używał id zadania jako klucza. Mamy pewność unikalności kluczy (ID zadania jest unikalne).
- Ustawiliśmy kilka propsów: initialNumToRender={10} (renderuj 10 zadań od razu – zakładamy, że to starczy by pokryć ekran typowego telefonu), windowSize={5} (to oznacza 5 okien wysokości ekranu – po 2 do góry i do dołu, co w sumie da ok. 2*10 + 10 = 30 elementów w buforze; zmniejszyliśmy nieco z domyślnego 21 aby zaoszczędzić pamięć, bo lista zadań raczej nie jest przewijana ultraszybko), maxToRenderPerBatch={5} (żeby nie dokładać więcej niż 5 elementów na raz, co ograniczy ewentualny lag przy szybkim scrollu), removeClippedSubviews={true} (aby usuwać z widoku elementy poza ekranem – szczególnie przydatne, jeśli nasza lista ma np. dużo obrazków, choć tu tylko teksty i buttony).
- Wewnątrz renderItem dodaliśmy testID dla przycisku Done, co ułatwi nam selekcję w testach (np. testID="done-btn-42" dla zadania o id 42). To praktyka warta stosowania – generować testID zawierające identyfikator elementu, jeśli będziemy klikać konkretny element na liście podczas testów.

W kwestii stylów i layout – styles.taskItem pewnie definiuje coś jak flex row: Text + Button obok siebie. Ważne, że każdy <View> ma nadany key przez FlatList (on sam to robi na podstawie keyExtractor) i że dbamy, aby klucz była stabilny i nie zmieniał się gdy task zmieni np. nazwę.

Zalety po refaktoryzacji: Teraz nawet jeśli tasks ma 1000 pozycji, FlatList nie wyrenderuje wszystkich – tylko ~30 (zależnie od wysokości pojedynczego elementu, initialNumToRender i windowSize). Scroll będzie płynny, bo nie trzymamy w pamięci wszystkich elementów na raz. Mamy też pewność unikalnych kluczy dzięki keyExtractor.

Pisanie testu komponentu listy (render + interakcja):

Chcemy przetestować, że TaskList wyświetla poprawną liczbę elementów i że kliknięcie "Done" przy konkretnym zadaniu wywołuje funkcję onTaskDone z prawidłowym argumentem (id zadania). Możemy to zrobić za pomocą React Native Testing Library:

```
import React from 'react';
import { render, fireEvent } from '@testing-library/react-native';
import { TaskList } from '../components/TaskList';

const sampleTasks = [
  { id: '1', title: 'Kup mleko', done: false },
  { id: '2', title: 'Zapłać rachunki', done: false },
  { id: '3', title: 'Zadanie ukończone', done: true }
];

test('renderuje wszystkie zadania na liście', () => {
  const { getByText } = render(<TaskList tasks={sampleTasks} onTaskDone={() => {}} />);
  // Sprawdzamy obecność tytułów zadań:
  expect(getByText('Kup mleko')).toBeTruthy();
  expect(getByText('Zapłać rachunki')).toBeTruthy();
  expect(getByText('Zadanie ukończone')).toBeTruthy();
  // Sprawdzamy, że dla ukończonego zadania nie ma przycisku "Done":
  expect(() => getByText('Done')).not.toThrow();
  // UWAGA: powyższa linia jest podchwytliwa - getByText('Done') znajdzie *pierwszy* pasujący,
  // a mamy dwa nieukończone zadania, więc będą dwa przyciski "Done".
  // Lepiej użyć queryAllByText:
  const doneButtons = queryAllByText('Done');
  expect(doneButtons.length).toBe(2);
});
```

Powyższy test renderuje listę z trzema zadaniami. Sprawdzamy, że teksty wszystkich tytułów są obecne (czyli że lista renderuje każdy element). Następnie chcemy upewnić się, że element oznaczony jako done (id 3) nie wyświetla przycisku "Done". W naszym kodzie komponentu warunkowo renderujemy przycisk tylko jeśli !item.done. Więc dla 2 zadań done=false przycisk będzie, dla jednego done=true – nie. W teście moglibyśmy próbować znaleźć Done dla każdego, ale lepiej zebrać wszystkie przyciski "Done" i sprawdzić ich liczbę. Używamy queryAllByText('Done') – to zwróci tablicę pasujących elementów lub pustą tablicę jak brak. Powinniśmy dostać 2 elementy (dla id 1 i 2). Sprawdzamy .length === 2. Alternatywnie, moglibyśmy zrobić:

```
expect(queryByText('Done', { selector: ...something for id 3})).toBeNull();
```

ale to skomplikowane. Liczenie jest proste.

Następnie test interakcji:

```
test('wywołuje onTaskDone z poprawnym ID po naciśnięciu przycisku Done', () => {
  const mockOnTaskDone = jest.fn();
  const { getByTestId } = render(<TaskList tasks={sampleTasks} onTaskDone={mockOnTaskDone} />);
  // Bierzemy np. pierwsze zadanie (id '1') i klikamy jego przycisk
  const button = getByTestId('done-btn-1');
```

```
  fireEvent.press(button);
  expect(mockOnTaskDone).toHaveBeenCalledWith('1');
  expect(mockOnTaskDone).toHaveBeenCalledTimes(1);
});
```

Tutaj tworzymy mockOnTaskDone jako funkcję testową (spy). Renderujemy TaskList z sampleTasks. Dzięki temu, że w TaskList nadaliśmy testID dla każdego przycisku zawierające id zadania (done-btn-${item.id}), możemy w teście łatwo odwołać się np. do done-btn-1 dla pierwszego zadania. Pobieramy ten element przez getByTestId i symulujemy fireEvent.press. Następnie sprawdzamy, że mockOnTaskDone został wywołany raz, i to z argumentem '1'. W ten sposób upewniamy się, że mechanizm przekazywania ID zadziałał. Można ewentualnie powtórzyć to dla innego elementu (np. id '2').

Ten test weryfikuje integrację komponentu z logiką – czyli czy UI poprawnie wywołuje przekazaną funkcję z odpowiednim parametrem. W realnej aplikacji onTaskDone mogłoby np. ustawiać stan lub wysyłać akcję Redux. Tutaj tylko sprawdzamy, że został zawołany.

Uruchomienie testów: Po napisaniu powyższych testów (np. w pliku TaskList.test.js), używamy npm test. Zakładamy, że konfiguracja jest ustawiona (Jest, Testing Library). Testy powinny przejść, o ile komponent działa zgodnie z założeniami. Gdybyśmy np. zapomnieli dodać testID do Buttona, to getByTestId('done-btn-1') by nie znalazł elementu i test by failował – to sygnał, że trzeba poprawić implementację (dodać testID lub znaleźć inny sposób identyfikacji, np. getByText + within konkretnego itemu, ale testID jest najwygodniejsze).

Podsumowanie dema: Pokazaliśmy, jak przejść od nieoptymalnej implementacji listy do wydajniejszej używając FlatList z odpowiednimi ustawieniami. Jednocześnie zwróciliśmy uwagę na testowalność: dodaliśmy testID by ułatwić wybieranie elementów w testach, a całą logikę kliknięcia zostawiliśmy w props (onTaskDone), co wpisuje się w zasadę, by komponenty UI były jak najbardziej *czyste* i sterowane z zewnątrz – co ułatwia mockowanie i testowanie. Nasz test komponentu listy potwierdził zarówno renderowanie danych jak i poprawną obsługę interakcji.

Literatura:

1. **https://reactnative.dev/docs/optimizing-flatlist-configuration** (Data dostępu: 1.10.2025) – Oficjalna dokumentacja React Native dotycząca optymalizacji list, szczegółowo opisująca parametry takie jak windowSize, initialNumToRender oraz techniki poprawy płynności przewijania.
2. **https://wix.github.io/Detox/docs/introduction/getting-started** (Data dostępu: 1.10.2025) – Przewodnik po frameworku Detox, wyjaśniający architekturę testów end-to-end typu gray-box oraz mechanizmy automatycznej synchronizacji z aplikacją React Native.
3. **https://fbflipper.com/docs/features/react-native/** (Data dostępu: 1.10.2025) – Dokumentacja narzędzia Flipper, opisująca funkcje debugowania, w tym Layout Inspector, Network Inspector oraz integrację z wtyczką Performance dla aplikacji mobilnych.
4. **https://testing-library.com/docs/react-native-testing-library/intro/** (Data dostępu: 1.10.2025) – Dokumentacja React Native Testing Library (RNTL), prezentująca dobre praktyki testowania komponentów i hooków w oparciu o interakcje użytkownika.