KIELCE UNIVERSITY OF TECHNOLOGY

Mobile Applications – Lecture 8

Architecture, Testing and Performance

Mateusz Pawełkiewicz

1.10.2025

Introduction: In this talk, we'll cover a modern approach to React Native mobile app architecture, testing strategies (from unit testing to end-to-end testing), and debugging and performance optimization techniques. All advice is based on current best practices, leveraging the latest libraries and patterns.
In particular, we will focus on organizing the project in the style of*feature-first*, logical layering, separating the domain layer from API data (DTOs vs. domain models), writing unit tests, components, hooks, and E2E (Detox), using debugging tools (Flipper, React DevTools), and optimizing lists and animations (FlatList tuning, avoiding unnecessary renders, introduction to Reanimated). All of this will be illustrated with commented code examples to facilitate understanding and practical application of the concepts discussed.

1. React Native Application Architecture

In a well-designed React Native application, the right approach is key.**project architecture**– the organization of files, modules, and layers in code. A good architecture facilitates application scaling, teamwork, testing, and code maintenance. We will discuss the approach.*feature-first*to organize the code, we will explain the division into logical layers (ui, hooks, services, api, types, utils), as well as the concept**DTO vs. Domain Model**and data mapping and error handling from the backend.

1.1 Approach*feature-first*vs. traditional layered architecture

Traditionally, many React Native (and React) projects have been organized*in layers*– files were grouped by type, e.g. global directoriescomponents/, screens/, utils/, services/, types/etc. This structure is initially intuitive, but as the project grows, it can become problematic. As the application grows to dozens or even dozens of screens, a layered approach often leads to:

- **Poor scalability**– Functions related to a single feature are scattered across multiple folders, making it easy to create unwanted dependencies between modules. A change in one place can inadvertently affect another area of the application.
- **Tight coupling**– lack of clear boundaries between business domains. Code for different functionalities is mixed, making it difficult to isolate and modify individual elements.
- **Conflicts in the team**– multiple developers can work on the same “global” files (e.g. adding things to oneutils.jsor by modifying the commonstore), which increases the risk of conflicts and integration errors.
- **Difficulties in testing**– It's difficult to test a module in isolation when the logic of a single feature is spread across multiple layers. Mocking dependencies can be complicated because there are no clear boundaries between components and logic.
- **Maintenance problems**– Adding a new feature increases the "chaos" because it often requires editing many scattered files. The code becomes less readable over time, and refactoring becomes risky.

The solutionthese problems is the approach**feature-first**, which reverses the traditional pattern. Instead of grouping files by type, we group them by*functionality (feature)*Each main functional module of the application has its own subdirectory containing everything related to it: UI components, screens, logic (hooks, services), type definitions, and even a sub-

module for communicating with the API.*feature folder*it's like a mini-application inside the project - it has most of the things needed for a given function to work, which is similar to the microservices approach in the backend, but applied locally in the front-end code.

Example feature-first structure: Let's imagine an application with modules: authentication, home screen, onboarding, settings, and todo list. The project structure might look like this:

```
src/
├── features/
│   ├── auth/
│   │   ├── api/ # code for communicating with the API for the auth module (e.g. login, logout)
│   │   ├── components/ # auth-specific UI components (e.g. login form)
│   │   ├── hooks/ # hooks related to auth (np. useAuth)
│   │   ├── screens/    # ekrany (React components) modułu auth (np. LoginScreen)
│   │   ├── services/ # auth business logic (e.g. token handling, validation)
│   │   ├── store/ # (optional) auth-specific global state (e.g. Redux slice)
│   │ └── types/ # auth-related type definitions (TypeScript) (e.g. UserCredentials)
│   ├── home/
│   │ └── ... # analogous structure for the main screen module
│   ├── onboarding/
│   ├── settings/
│   └── todos/ # task list module
├── navigation/ # application navigation (e.g. stacks, tab navigator – often global)
├── services/ # shared services (e.g. common API client, configuration)
├── store/ # global store (e.g. Redux configuration, persistence)
├── ui/ # common UI components (e.g. a button used in multiple places)
└── utils/ # common helper functions (e.g. date formatting, logging)
```

In the above structure, each folder in`features/`represents relatively**standalone business module**, which *can be developed, tested, and even removed independently of the rest of the application*This division forces loose connections between modules - communication takes place via defined interfaces (e.g. calls in`services/api`) instead of by splitting global variables. This allows:

- The application scales better – when adding a new feature, we add a new folder, minimizing the impact on existing code.
- Developers can work on different functionalities in parallel without getting in each other's way (less conflicts).
- **Testing**becomes simpler – we can more easily run tests for the entire feature module (or mock one module while testing another). Individual functionalities are isolated by definition, which favors the creation of unit and integration tests.
- **Readability and maintenance**– code related to a given issue is grouped together, making it easier for newcomers to understand where to look for the relevant logic. If something isn't working, for example, with logging, they know that most of the relevant code will be in`features/auth/*`.

It is worth noting that the feature-first approach does not exclude having some shared parts of the application – e.g. general-purpose UI components (`src/ui`), global HTTP client configuration (`src/services/api`), or common application state (`src/store`). However, even these elements can be designed to be used by individual features in a way*compositive*, not to replace the feature logic. For example, we can have a global API client, but individual feature

modules create their own API functions (in their own folderapi/), using this client to make network calls.

To sum up: Feature-first architecture improves the modularity and scalability of a React Native project.

1.2 Logical layers:ui, hooks, services, api, types, utils

In the feature-first structure we distinguish**logical layers**inside each module. The listed directories (ui, hooks, services, api, types, utils) perform the following roles:

- **ui(user interface)**– presentation components related to a given feature. These may include, for example, specific controls, buttons, tabs, or forms used only within that module. Often, these are simple components that are aware only of their own props (so-called*dumb components*) that can be reused. If a component becomes necessary in multiple features, it can be moved to a common directorysrc/uiExample: in moduleallwe can have a componentTodoItemrendering a single task.
- **hooks**– Custom hooks with business logic or handling the local state of a feature. Hooks can hide internal implementation details (e.g., how data is retrieved or how changes are reacted to) and provide a convenient interface to components. Example:useTodos() w module allfetching a list of tasks from the API, handling loading status, error, etc. Hooks also enable easy reuse of code in different components (e.g. the same hook used on different screens).
- **services** - layer **business/domain logic**. Here we place code implementing specific use cases (use-cases) not directly related to the UI. Services can call functions from the layerapi, may also include, for example, feature-specific state management (e.g., simple*store*based on Context or global store operations). In some projects this layer is called*use-cases*, *logic*the*controllers*. Example: todos/services/todoService.tsmay have functions such asaddTodo, toggleTodoCompleted– operating on task data by calling the API and updating the local state.
- **api**– an external data access layer. Contains functions/fetchers for communicating with an external API (backend) or a local database. Many modern applications use this layer, for example:React QuerytheRTK Queryfor efficient data retrieval and caching, or classic fetch/axios calls. Layerapishould be**separated from the rest**– so that changing API details (endpoints, data formats) does not affect the UI code. Example:auth/api/authApi.tscan export functionlogin(credentials)which performsPOST /loginand returns an object e.g.AuthTokenIn practice, this can be implemented, for example, using the RTK Query library, which allows defining*endpointy*for mutations and queries.
- **types**– type and interface definitions (if we use TypeScript, which is currently the standard in RN). Separating types makes them easier to reuse between layers (e.g. interfaceEverythingused inapiand inui). Types are often divided into DTOs (API response types) and domain models – more on that in a moment. In the foldertypeswe can also include, for example, types for component props to avoid cluttering the main component code.
- **utils**– auxiliary functions specific to a given module. These may include data formatting, validation functions, parsers, etc. If a utils function proves to be globally

useful, it may be included in`src/utils`At the feature level, we only keep what's unique to a given area. Example:`utils/formatTodoTitle.ts`– a simple function for formatting the task title (e.g. trimming names that are too long).

Advantages of division into layers: This order enforces a clear separation of responsibilities (Separation of Concerns). The UI is solely concerned with presentation and doesn't need to know where the data comes from – it receives it from the hook/service. The service knows the business logic but isn't responsible for rendering – it calls the API and, for example, manages state. The API layer isolates the details of communication with the server (URLs, JSON structure) from the rest of the application. This allows changes in one layer to minimally impact the others. This organization promotes**testability**– we can, for example, test the service logic independently by providing it with a mocked API layer (or using the so-called*dummy data*instead of real calls). Similarly, we can test UI components by providing them with fictitious data and custom hooks, without actual contact with the backend.

1.3 DTO vs. modele domenowe (Domain Models)

When working with the data layer, it is worth distinguishing between two concepts:**DTO (Data Transfer Object)** i **domain model**In the context of a client application (RA), a DTO is typically the data structure exactly as it comes from the API, while a domain model is the data representation we use within our application (tailored to the needs of the interface and business logic).

- **DTO**– is an object that transfers data between systems (e.g., between the server and our application). It has a structure defined by the backend, often nested, containing fields needed mainly from the server's perspective. A DTO can contain, for example, technical fields, identifiers, and relationships that are not always directly needed in the UI. APIs often use standards (e.g., JSON:API) that impose a certain response shape (e.g., object`attributes`, `relationships`etc.).
- **Domain model**(sometimes also called*business model*the*domain entity*) – this is our own, internal data representation, tailored to the application's needs. A domain model should be simpler, containing only what the application truly needs, in a form convenient for use in code. It can also contain methods or logic operating on the data (although in JavaScript/TypeScript, models are more often simply interfaces/types, with the logic contained in separate functions). A domain model is independent of how data is stored or transmitted – it describes business concepts in a way that is understandable to the developer and (theoretically) to the business analyst.

Key difference:**DTO exists for technical reasons**to move data across a network or between layers, while**the domain model exists for business reasons**– maps the concepts used by the domain/problem we are solving. Often, a domain model has*richer behaviors*(business methods, validations) or at least clearly reflects the language of the domain (e.g. class/type`Order`with the method`calculateTotal()`unlike DTO`OrderDTO`being just a "dumb" set of fields).

In practice, in React/React Native applications, domain models are sometimes**flattened**and simplified compared to DTO. Let's look at an example to illustrate this:

Example: Let's assume that the backend returns the user in a JSON:API compliant format, e.g.:

```
// Sample user DTO from API
{
  "id": "123",
  "type": "user",
  "attributes": {
"handle": "jan_smith",
    "avatar": "https://example.com/avatar.jpg",
"info": "User Bio..."
  },
  "relationships": {
    "followerIds": ["u1", "u2", "u3"]
  }
}
```

This is **DTO**– structure exactly as in the server response. However, in our code, using such a nested object would be cumbersome (constant references`user.attributes.handle`etc.), and what's worse - it makes us highly dependent on the server format. If the backend changed its structure (e.g. removed the field`attributes`), our UI code would fall apart because we refer to it everywhere`user.attributes.handle`.

Therefore, in the layer**data mapping**we transform DTO into*domain model*, e.g. like this:

```
// User domain model in the application (TypeScript interface)
interface User {
  id: string;
  handle: string;
  avatar: string;
  info?: string;
  followerIds: string[];
}
```

Here we have all the necessary fields,*flattened*and named intuitively. This model is used by the rest of the application (UI, logic) – it is simple and does not contain unnecessary nesting. How to achieve it? By**DTO mapping -> model**in the layer`api`the`services`.

We can write a mapper function, e.g.:

```
// src/features/user/api/userMapper.ts
import { UserDTO } from '../types'; // we assume that UserDTO is a type corresponding to the structure from the API
import { User } from '../types'; // our domain model

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,
  };
```

```
}
```

We call this function immediately after receiving data from the API, before passing it on. Example service fragment:

```
// src/features/user/services/userService.ts
import { apiClient } from '../../../services/apiClient'; // global HTTP client (e.g. 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;
}
```

Thanks to this **UI components and business logic operate on a convenient model**`User`, e.g. they can write directly`user.handle` instead `user.attributes.handle`This reduces the complexity of the frontend code and makes us independent of potential API changes. When the backend changes format, we simply modify the mapping function instead of searching and correcting usage across dozens of components.

Furthermore, separating models from DTOs helps in**error handling and validation**For example, if the server returns a validation error with information in a specific format (e.g. fields`errors`containing a list of messages), this layer`api/service`can capture such an error and transform it into an application-friendly error object or exception. You can define your own domain error type, e.g.`ValidationError`with fields`field` i `message`, and map the server's response to such an error. This allows components or hooks to receive a ready-made, understandable error object (e.g., to display a message), instead of the server's raw response. An error handling example:

```
try {
  const user = await fetchUser('jan_kowalski');
  setUser(user);
} catch (e: any) {
  if (isValidationError(e)) {
showToast(e.message); // our own message from the domain exception
  } else {
    console.error('Unknown error fetching user', e);
  }
}
```

Function`fetchUser`inside it can do something like this:

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

Where `mapError`transforms the error structure from the API into our error class/object. Of course, the implementation depends on the API - the principle is that**keep error logic in the communication/service layer, rather than scattering it across components**The component

should receive pre-processed information (e.g. that login failed due to incorrect password), instead of raw HTTP 401 with a JSON body.

Summary: Separating DTOs from domain models makes our code more readable and more resistant to change. The UI is not*tightly coupled*with the server data format, which reduces complexity and improves resilience to backend changes. Introducing a mapping layer is a step towards a cleaner architecture –*"pure" frontend, where API data is isolated in its own layer*(often referred to as*Domain Layer*in literature). This concept is inspired by the principles**DDD (Domain-Driven Design)**, although we use it in a lighter way, adapted to the realities of a mobile application.

2. Testing in React Native – from Unit to E2E

Testing React Native applications occurs on several levels. Our modern approach emphasizes both**unit tests**(single functions, logic),**component and hook tests**(i.e. UI integration tests in a controlled environment) and on**testy end-to-end (E2E)**simulating real-world application usage. We'll discuss these types of tests one by one, tools such as**Is**, **React Native Testing Library** i **Detox**, and how to configure a test environment. We'll also show code examples of component and hook tests to demonstrate a practical approach.

2.1 Unit Testing (Yes)

Unit teststhey test the smallest units of code – usually individual functions or modules – in isolation from the rest. In the React Native ecosystem, the standard for unit testing (and most frontend testing) is**Is**– a testing framework provided immediately when creating a React Native project (e.g. by`npx react-native init`). It offers a runtime environment for JavaScript tests with built-in**default support for React Native** (preset `react-native`in configuration).

The unit test in Jest is a function`test(...)`or alias`it(...)`, in which we describe the expected behavior of the code. We can use assertions (matchers) such as`expect(x).toBe(y)` or `expect(obj).toHaveBeenCalled()`(for mocks). For example, having a function in`utils`calculating the sum, the unit test would look like:

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

test('correctly sums two numbers', () => {
expect(sum(2, 3)).toBe(5);
});
```

Of course, the real power of unit testing comes with more complex logic. In the RN project, we will be writing such tests mainly for**business logic**(e.g. functions in`services/utils`) and for*custom hooków*(as long as we test them directly) or state reducers. Unit tests are fast and isolated – they don't touch the UI or network, so they run in milliseconds and provide feedback on every build/commit.

Dependency mocking: During unit testing we often use the mechanism*mocks*in Jest. It allows you to replace actual modules or functions with "mock" implementations, so you can test a given piece of code in isolation. Example: when testing a function`fetchUser`from the previous

section, we don't want to make a real HTTP request in the test. So we can mock the module`apiClient`using`jest.mock('modulename')`and defining that`apiClient.get`returns an object with sample data (simulating the server response). This makes the test deterministic and fast.

Configuration Is: In RN projects, the minimum configuration is to install dependencies and possibly a configuration file`jest.config.js`A typical setup (if we're using the React Native Testing Library, which I'll discuss in a moment) might require adding`setupFiles`matcher extensions. For example, to use additional assertions`@testing-library/jest-native` (e.g. `toBeVisible()`, `toHaveTextContent()`), the following is added to the configuration file:

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

This is all you need to start writing tests. To run the tests, use the command`npm test`(or`yarn test`), which runs all files with names ending in`.test.js`the`.spec.js`(default in configuration).

2.2 Testing Components and Hooks (React Native Testing Library)

Unit tests cover the logic, but in UI applications it is also important to test**components**– that is, whether the interface renders as intended and responds to user interactions. For this purpose, we use**React Native Testing Library (RNTL)**, which provides tools for rendering components in a test environment and checking their content and behavior. RNTL is an adaptation of the popular Testing Library from React web, tailored to the specifics of React Native (e.g., it supports views, Text, etc.).

Installation: We assume we already have Jest. Let's install the packages:`@testing-library/react-native`and optionally`@testing-library/jest-native`(as above, for additional matchers). These libraries allow you to use functions like`render`, `fireEvent`and type assertions`toHaveTextContent`After installation and configuration (as mentioned in 2.1), we can write component tests.

Rendering a component in the test:RNTL provides the function`render()`to render the component in a test context (not in a real emulator, but in a virtual environment resembling a component tree).`render`returns an object containing, among others, methods for searching for elements:`getByText`, `getByTestId`, `getByRole`etc. This allows us to find a virtually rendered element and check if it exists, has the appropriate props, styles, text, etc.

Example – let’s test a simple SocialLinks component that displays a link to a user’s profile (e.g. Twitter):

```
// SocialLinks component (props: label, link, type), e.g. shows icon and text
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>
);
```

We want to test whether: (a) the appropriate label text is rendered, (b) the icon (Image) is present, and (c) the link opens when clicked. To do this, we write tests using RNTL:

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

// Preparation: we'll mock Linking.openURL so that it doesn't open a real browser:
jest.spyOn(Linking, 'openURL').mockImplementation(() => Promise.resolve());

test('renders social link label', () => {
  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(); // check if the text has appeared
});
```

In the above test we use`render`to render the component with sample props. Then`getByText`searches for an item`<Text>`by content. We expect that such an element exists (matcher`toBeTruthy()`checks if the found element is not null). This test ensures that the component actually displays the passed label. If someone were to mistakenly change the component so that it ignores the prop`label`(e.g. hard-coded the text), the test would fail, which would catch the regression.

Second test – is the icon rendered:

```
test('renders a social icon (e.g. Twitter)', () => {
  const { getByRole } = render(
<SocialLinks type="twitter" label="John Doe's Twitter" link="https://twitter.com/johndoe" />
  );
  const icon = getByRole('image');
expect(icon).toBeTruthy(); // presence of an image (Image)
});
```

Here we used`getByRole('image')`In RNTL, roles correspond to accessibility attributes (accessibilityRole). Because the <Image> component defaults to accessibilityRole="image" (which maps to a role), we can use this method to find an image in the test. The test only checks for the presence of an image – it could be clarified, for example, by checking whether the image source is correct, but often it's enough to check the element's presence (tests don't need to delve into every UI detail, but rather into key aspects).

Third test –**reaction to interaction**(click):

```
test('opens link after pressing the component', () => {
  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); // simulate a tap on the entire TouchableOpacity (here on its text)
  expect(Linking.openURL).toHaveBeenCalledWith('https://twitter.com/johndoe');
});
```

Preparation is key in this test: we previously used`jest.spyOn`to make`Linking.openURL`mock function. This allows us to check the calls to this function. We use`fireEvent.press(...)`to simulate pressing a touch element (our component wraps <Text> and <Image> in <TouchableOpacity>, so clicking the text will also work). We expect that`Linking.openURL`was called with a specific URL. If, for example, we make a mistake and call something else in the component, the test will detect it.

This is how we tested the component's key functionality: rendering correct data and responding to interactions. Component testing allows us to catch many errors before we launch the app on a device, such as a missing important element, an incorrect field name, no response to a click, etc. Importantly, these tests are**deterministic**and quite fast (although slower than pure unit-level ones, because they have to render components). They don't require an emulator - they run in Node with a simulated RN environment.

Testing custom hooks: Hooks can also be tested, although this requires either creating a test component that uses the hook or using a dedicated library. There is`@testing-library/react-hooks`(React Hooks Testing Library), which makes it easier to test hook logic without composing it into a real component. It may be integrated in the future, but as of 2025, it can still be used. For example, let's assume a simple hook.`useCounter(initialValue)`returning the current counter and function`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 using React Hooks Testing Library:

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

test('initializes the counter and increments it', () => {
  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);
});
```

Function `renderHook` calls our hook and returns an object with a field `result` containing the current hook result (under `result.current`). We can check the initial value, then use `act()` to perform an action that changes the state (call `increment`) and then assert that the state has changed as expected. Use `act` is required to properly simulate the render cycle (it's a Testing Library mechanism that says "this is a state change, get over the re-render"). This test ensures that our hook is working correctly, isolating the logic from the component.

Of course, hooks that use context or other RN hooks may need to be wrapped in the appropriate provider during the test (RNTL `render` has a wrapper option). But in simple cases, this approach is sufficient.

2.3 E2E Testing – Detox Tool

Unit and component tests cover our code "inside" the app, but they don't provide 100% assurance that the app as a whole will run correctly on the device. This is where **testy end-to-end (E2E)**, which simulate real application usage scenarios on a physical device or emulator. In the React Native ecosystem, the de facto standard for E2E is the library **Detox** (developed by Wix).

What is Detox? Detox is an E2E test automation framework designed specifically for mobile applications (especially React Native). Unlike pure black-box approaches (e.g., Appium), Detox is **tool *gray-box*** – This means that tests are run on the real application (just as a user would use it), but Detox has visibility into the internal state of the application, allowing it to better synchronize actions with what's happening internally. This makes the tests more stable: framework **waits for the application to be idle** (idle) before performing subsequent steps – e.g. it waits until animations, network requests, component rendering, etc. are finished, before e.g. trying to click a button.

How does it work from a technical perspective? Detox integrates with native UI testing frameworks – on iOS it uses XCUITest, on Android it uses Espresso. The difference is that these frameworks are normally black boxes (they just click and look at the UI), while Detox adds a layer of communication with the RN backend. Our RN app is launched with a special Detox library that reports status to the tester (e.g., "something is still rendering, the animation loop is still running"). We write tests in JavaScript (run by Jest or Mocha), and they communicate with the app through the Detox bridge. The diagram looks something like this:

- We are launching `detox test` – the application (test version) is built and the test runner is launched.
- Detox launches the app on the emulator/simulator. Tests wait until the app is ready.
- Then each test **share** (tap, text entry, scroll) is sent to the native page (Espresso/XCUITest) which executes it on the device.
- In turn, each **assertion** (checking if any element is visible) works in such a way that Detox can, based on *matchers* (e.g. text, id) find an element in the application UI hierarchy.

- **Synchronization:**Detox automatically makes sure the application is ready before executing an action/assertion – for example, it doesn't execute a tap until the application is idle (does not perform animations, JS operations). This practically eliminates the need to use`sleep()`or manual expectations – something that often plagues E2E testing.
- The results (success/failure) are passed to the test framework and we can see them in the console.

Writing Detox Tests: We create tests in a similar way to regular tests, e.g.`e2e/login.spec.js`. Detox provides global objects for interaction:

- `device`– device control (e.g.`device.reloadReactNative()`to reset the application between tests, or`device.rotateScreen()`).
- `element(by.matcher(...))`– selecting an element on the screen using a matcher (e.g.`by.id('loginButton')`, `by.text('Hello')`, `by.label('Password')`).
- Actions on an item: Once an item is selected, you can chain it.`tap()`, `.typeText()`, `.clearText()`, `.scroll()`etc.
- Assertions:`await expect(element(by.id('something'))).toBeVisible()` or `.toHaveText('...')`etc.

Detox requires that our elements in the application are*identifiable*It's best to give them an attribute`testID`(RN has such props for all basic components.) For example, by defining`<Button testID="loginButton" .../>`we can use in the test`element(by.id('loginButton'))`to find this button. This is crucial – without the appropriate`testID`tests would have to rely on text or view structure, which can be unreliable and susceptible to UI changes.**Therefore, good practice:**all interactive elements (buttons, text fields) and key displayed information should be inserted from`testID`when creating a component to facilitate E2E testing.

Example E2E test scenario: Consider a simple login flow. We have a login screen with username and password fields, and a "Log In" button. After logging in, we move to the "Second Screen." Here's a sample test in Detox (pseudo-code similar to the real thing):

```
describe('Login Flow', () => {
  beforeAll(async () => {
await device.launchApp(); // launch the application
  });

  beforeEach(async () => {
await device.reloadReactNative(); // reset state before each test, if needed
  });

it('should log the user in with the correct credentials', async () => {
await expect(element(by.text('Log In'))).toBeVisible(); // check if we are on the login screen
    await element(by.id('LoginInput')).typeText('Admin');        // wpisz login (pole tekstowe ma
testID="LoginInput")
await element(by.id('PasswordInput')).typeText('password'); // enter password
await element(by.text('Log In')).tap(); // press the "Log In" button
await expect(element(by.text('Second Screen'))).toBeVisible(); // wait for transition to second screen
  });

it('should display an error on empty fields', async () => {
await element(by.text('Log In')).tap(); // without entering anything, click login
```

```
await expect(element(by.id('LoginInputError'))).toBeVisible(); // the login field should display an error (we assume the field component has <Text testID="LoginInputError"> on error)
await expect(element(by.id('PasswordInputError'))).toBeVisible(); // same for password
  });
});
```

The above code (simplified) shows how the E2E test logs and checks navigation and validation. Note the assertions:`expect(element(by.text('Log In'))).toBeVisible()`– Detox checks if there is an element with the text "Log In" and if it is visible on the screen (meaning the login screen is active). Typing text`typeText`simulates a keyboard on the device – keystroke events are sent to the application in the background.`tap`simulates tapping the screen at the location of the element. Thanks`await`and internal synchronization Detox automatically waits for each action to complete. E.g.`await element(by.text('Log In')).tap()`will wait until the tap action is completed and any navigation is complete before continuing.

Detox – setup and launch: To use Detox, we need to install the package`detox`and adapt the native project (especially iOS) – e.g. by adding to`Podfile`Detox library. In the file`package.json`we configure different*configurations*Detox, e.g., for iOS simulator, for Android emulator. Configuration example (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" }
    }
  }
}
```

Similarly for Android (Gradle build and apk path). Then run:`detox build -c ios.sim.debug`(builds a test application) and`detox test -c ios.sim.debug`(runs tests). This is a rather complex topic, but details are available in the Detox documentation. It's worth mentioning that Detox can be integrated with CI (for example, it's popular for continuous integration platforms, and Codemagic even has direct support).

Detox Advantages:

- It is fast and stable compared to e.g. Appium, because it is adapted to RN and uses*gray-box*approaches.
- Tests run without the need for manual instrumentation (everything is scripted).
- Supports both iOS and Android in one framework (write the test once, it works on both).
- It allows you to test authentic scenarios: from app startup, through switching between screens, to native integration (e.g. you can simulate push notifications, various application states).

- Integrates with Jest – so you can have both unit and E2E tests in one repo, using a familiar runner.

Challenges and good practices:E2E tests are the slowest and most complex to maintain, so we usually don't write very many of them – we cover the key paths (happy paths and a few edge cases). We need to be careful to avoid "flakiness" – situations where a test sometimes passes, sometimes fails. Detox largely solves this with synchronization, but the programmer must remembertestIDand unique selectors, and sometimes about clearing the state (hence e.g.device.reloadReactNative()so that each scene starts from a clean state). It is also important to stub out any external dependencies – for example, if the application encounters an unstable API during E2E, you can use the so-called*mock server*or enable a special mode where the application uses local data instead of real requests (Detox allows, for example, overriding fetch, or you can set up a stub server on localhost).

In summary, Detox is a powerful tool that automates testing from the user's perspective and**ensures that the application actually works correctly as a whole**Using it significantly improves quality assurance.

3. Debugowanie React Native – Flipper i React DevTools

Even the best architecture and testing will not protect us from the necessity**debugging**when building an application. Debugging in React Native can be challenging because we're dealing with both JavaScript and native code, and their communication via a bridge. Fortunately, there are tools that make diagnosing problems, viewing application status, and analyzing performance much easier. We'll focus on two key areas:**Flipper**– an extensive, officially supported debugging tool from Meta, and**React DevTools**– a tool for inspecting the React component tree and profiling rendering.

3.1 Flipper – a versatile mobile app debugging tool

Flipperis a desktop application created by Facebook/Meta that serves as a platform for debugging mobile applications (Android, iOS) – with a focus on React Native, but also supports other technologies. Since React Native 0.62, Flipper is integrated with RN by default (in debug mode). This means that when running our application in developer mode, we can connect to it via Flipper without additional configuration (sometimes this only requires installing a packagereact-native-flipperand launching a pinball machine in the background).

Key features of Flipper: Flipper has a plugin architecture, with each plugin offering a preview of a different aspect of the application. The most important aspects for debugging RN are:

- **Layout Inspector**– visualization of the application's view hierarchy and properties. This works similarly to the "Inspect" tool in a browser or Android Studio: you can preview the UI structure (views, components, and their relative arrangement), select elements on the device's screen, and view their styles/props. This is extremely useful when, for example, an element isn't displayed or has incorrect dimensions – in Flipper, you'll see the entire view and can more easily pinpoint the problem (e.g.,

zero-height view, overlapping element, etc.). Flipper even allows you to edit certain values live, which helps you experimentally find a fix.

- **Network Inspector**– monitoring all HTTP requests outgoing from the application. In the Network tab, Flipper captures requests and responses (including headers and body) sent by the application. This allows us to check, for example, whether the API request is actually being sent, what the address is, what is being returned from the server, how long it took, and what the status was. This saves us from having to add logs in the code or use a proxy – Flipper acts as a kind of*sniffer*RN application network traffic. You can filter by URL, method, etc. When it comes to debugging backend communication issues, this plugin is invaluable.
- **Log Viewer (Log Console)**– a combined view of native and JavaScript logs. Instead of looking at logcat (Android) or Xcode console, and additionally at console.log from the RN debugger, Flipper shows everything in one place. We can see logs from the device (e.g., native errors, JNI exceptions) as well as those written byconsole.login JavaScript. This works without any additional configuration. If your application crashes without a clear message in RN, it's worth checking Flipper Logs – you might see a stacktrace from native there.
- **React DevTools**– Flipper has built-in integration with React DevTools (it runs the latest version of React DevTools underneath). This allows us to switch to the React tab in Flipper and get a tree view of React components (our high-level ones, not the low-level native views like in Layout Inspector). This allows us to debug, for example, the state of components, props passed to them, and the component hierarchy. We also have access to**React Profiler**– the ability to record rendering progress and see which components are rendering and how long it takes. This is a great optimization tool – you can detect unnecessary re-renders and see where the application is spending time during rendering. (React DevTools and the profiler are identical to those in React web apps – simply connected to the RN context).
- **Crash Reporter**– Flipper can capture application crash reports (especially on Android). If the app crashes, Flipper will display the stacktrace and exception information. This helps identify, for example, a problem in native code or in an RN module (e.g., calling something that causes a NullPointerException in Java – Flipper will show it).
- **Database & Preferences**– Built-in plugins allow you to view the local database (SQLite) and AsyncStorage on the device. In debug mode, you can look into the SQLite database used by the application (viewing tables and records), as well as AsyncStorage (keys and values). This speeds up debugging, for example, problems with saving settings or caching data locally.
- **Performance**– By default, Flipper has a simple**FPS monitor**(There's a perf switch in the top right corner of the app that shows the current FPS.) However, more complete performance data is provided by the plugin**React Native Performance**(also known as*Flashlight*). It needs to be installed (from the Flipper marketplace). It provides graphs of CPU usage (overall and per thread), frames per second, memory usage, etc. This allows us to profile the application's native performance in real time – for example, whether the CPU spikes to 100% while scrolling through a list, or whether the FPS drops (which would indicate lag). The Performance plugin can help identify performance bottlenecks.

Flipper is constantly being developed and has a wealth of additional community-created plugins (available in the Flipper marketplace). These include plugins for debugging WebSocket connections, viewing the current state of the Redux store, and simulating various network conditions.

How to use Flipper?In practice: we install the Flipper application on the computer (from fbflipper.com). We make sure that our RN application has Flipper enabled (in RN 0.62+ it is default in Debug, if not, then`npm i react-native-flipper`and minor configuration in native as in the documentation). We run the application on the emulator or device*in debug mode (Metro)*Flipper will automatically detect the application (provided we're on the same network) and show it in the list. Clicking it will give us access to plugins (some may require additional packages to be installed – Flipper will inform you). Then, use it as usual: click "Layout" to explore the UI, etc.

Flipper significantly reduces the need to use profilers like Xcode Instruments or Android Studio for many common tasks by consolidating debugging in one place. This saves time – for example, in the case of network problems, we don't have to build our own logging; we just look at the Network plugin. Additionally, it minimizes*context switching*: everything (UI, logs, network, status) is in one window.

Example of use: Let's say a button in our UI doesn't respond to taps. What can we do?

- In Flipper -> Logs we can check if the event has arrived at all (maybe in the logs`onPress`we log something).
- In the Layout Inspector, you'll see the hierarchy—perhaps another transparent view is covering the button and capturing touch (a common bug with overlays). The inspector will show whether the button is clickable or covered.
- We can also check if props in React DevTools`onPress`was actually passed to the component.
- If nothing helps, you can also use the touch log capturing function in Flipper (Touch Events plugin – installed community plugin).
- As a result, the diagnosis of the problem is much faster than by the method`alert('x')`or guessing.

To sum up, **Flipper**is the “Swiss Army knife” of RN debugging:*layout, network, logs, status, performance – everything at your fingertips*In 2025, it's the go-to tool for React Native developers, officially supported. As the documentation says, Flipper allows**view hierarchy inspection, network query monitoring, device log viewing, and React DevTools integration**in real time. This makes debugging much more convenient and effective.

3.2 React DevTools – Component Inspection and Profiling

Although React DevTools is integrated in Flipper, it is worth discussing its capabilities in more detail, as it concerns strictly*React layers*applications.

React DevTools This tool is familiar to many for debugging React web applications (e.g., as a Chrome extension). For RN, from version 0.62 onwards, you can use DevTools via Flipper or via the React Native Debugger (a separate application). DevTools allows you to:

- **Browse the React component tree**– we can see what components (functional, class) are installed, their hierarchy (what is the child of what). For each selected component we can preview**propsy**what he got,**stan**(for class components or useState hook), and also**hook values**(DevTools will show, for example, that our useCounter hook has the value count = 5, it = function, etc.) This is very helpful in understanding what's going on: e.g., whether the parent passed the correct prop value, why the child has state X, etc. We can dynamically edit props or state in DevTools, which can help test different scenarios without rebuilding the application.
- **Execute hook functions debuggably**– The new version of DevTools allows, for example, manual calling of a function that updates state (in hooks). For example, if we have a useState counter, we can click in DevTools to increase/decrease the value. It's a small feature, but sometimes useful.
- **Profile rendering**– React DevTools has a tab**Profiler**, where we can record the rendering process while performing certain actions in the application. After stopping the recording, we get a timeline with selected component renders and information on how many milliseconds it took to render a given subtree. Components that render frequently or take a long time will be highlighted (e.g., warm colors if it takes a long time). In RN profiling, this is valuable for optimization: for example, we will detect that when scrolling a list, the entire screen is rendered 60 times per second – perhaps we are missingReact.memoon some large component? Or that after changing one field, 10 other unchanged components are redrawn unnecessarily. The profiler will show us*co* i *How long*You can then use this information to memoize, split into smaller components, or shift computations off-render.
- **Debugging performance w hooks**– DevTools is also aware of Hooks, so for example it will show in what order the Hooks were called, which can help in understanding whether, for example,useEffectit is not fired very often.

It's worth noting that for React DevTools to work, our app must be in debug mode (connected to Metro). In release mode, we don't have this tool (unless we're building specifically with DevSettings). With Flipper, it's simple, because Flipper takes care of the connection itself.

Other debugging tools: Let us briefly mention that there are also alternative or complementary tools:

- **React Native Debugger**– a separate (open-source) application that integrates DevTools, a JS debugger, and Redux DevTools. It can be used instead of Flipper, especially if you use Redux, as it has a built-in Redux state viewer with time-travel debugging capabilities. However, Flipper has recently gained more popularity because it's official and plugin-based.
- **Chrome Debugger**(old approach) – RN used to debug JS through Chrome's built-in mechanism. Now, with Hermes (the JS engine), this has changed – JS debugging is done differently (Remotely in Hermes debugging), but you can also use Chrome devtools to debug your code (hitting JS breakpoints).
- **LogBox**– this isn't an external tool, but a built-in RN mechanism that elegantly displays errors and warnings in the app's UI during development. I mention this because debugging also involves responding to warnings/errors – RN LogBox (since

RN 0.63) makes this much easier (an in-app console with the ability to filter and ignore messages).

To conclude the debugging section:**a good set of debugging tools**is Flipper + React DevTools, which together allow you to look into almost every aspect of the application:

- from the native side (layout, logs, resources, network) – Flipper,
- from the React side (component state, render cycle) – DevTools.

With such an "arsenal" we can solve typical problems (UI not working, requests not arriving, something hangs) much faster and more reliably than by trial and error.

4. React Native Application Performance

Performance is a key issue – we want our app to run smoothly (60 frames per second or more), respond instantly to interactions, and utilize memory efficiently, even on low-end devices. We'll discuss optimization techniques primarily related to list rendering (FlatList and its configuration), avoiding unnecessary renders/layouts, and improving animations (introduction to the Reanimated library, native vs. frame-based animations).

4.1 List optimization – FlatList usage and performance configuration

In mobile applications, a common challenge is**long lists**elements (e.g. news feed, contact list, etc.). We cannot afford to render hundreds of components at once - it would cause huge memory usage and performance drops. React Native offers a component**FlatList**(and the related SectionList) as an efficient solution for displaying lists via*virtualization*(virtualized list). The main idea: FlatList renders only those items that fit in the current view (plus some buffer around them), and "removes" the rest or doesn't create them at all until the user scrolls towards them.

How FlatList (virtualized list) works: Below, FlatList uses a mechanismVirtualizedList. Divides the entire list into so-called*window*, which moves with scrolling. Elements outside the window can be unmounted or even never rendered until they enter the window. This drastically reduces the number of simultaneously active elements. Of course, FlatList requires certain information (e.g., element heights) to do this efficiently – hence its many configuration options.

Key performance features of FlatList:(they should be adapted to the context of use):

- **keyExtractor**– a function that generates a unique key for a list item if our data does not have a propertykeyWhy is this important? React uses keys to optimize elements – unique keys prevent unnecessary re-renders and allow you to track elements when reordering. Make sure the key is stable (e.g.item.id) and unique. If we don't set keyExtractor, RN will try to use the defaultitem.keyor index - using the index is not recommended, because changing the order of elements will cause React to treat them as different and repaint the entire list. Therefore**good keys are the foundation of list performance**.

- **initialNumToRender**– how many elements to render at startup (default 10). We can adjust this to cover the entire screen on the most popular devices, but not too much more. For example, if the screen only holds ~8 elements, we can set initialNumToRender to 8 or 12 (plus a margin). Too low a value can result in white flash at startup, while too high a value can unnecessarily burden startup (e.g., loading 50 elements when the screen only holds 8).
- **windowSize**– window size in screen units (default 21, meaning 10 screens above and 10 below the currently visible area + the current one). This parameter specifies how many elements in front of and behind the visible area should be kept rendered. A larger windowSize reduces the risk of empty space appearing during fast scrolling (because elements are already waiting just offscreen), but increases memory consumption and rendering time (because more elements exist). A smaller windowSize saves memory, but, for example, very fast scrolling may result in visible element loading (empty spaces). The optimal value depends on the nature of the list – if the list scrolls slowly, you can reduce the windowSize to save resources; for lists scrolling rapidly (e.g., social feeds), it's better to leave a larger value to maintain smoothness.
- **removeClippedSubviews**– setting (defaulttrueon Android,falseon iOS) specifying whether to remove (detach) views that have gone off-screen (so-called*clipped*, clipped). Enabling this option (manually on iOS, because on Android it's already true by default) causes elements that have moved far out of view to be removed from the native hierarchy, which reduces GPU and CPU load (they aren't considered in rendering and touch). The benefit is lower main thread usage (because there are fewer views to calculate layout and draw). The downside is that it can sometimes cause minor bugs – for example, in the past, it was reported that on iOS, some elements could disappear under unfavorable conditions if there were transformations or absolute positioning. Generally, however, it's worth enabling removeClippedSubviews for very long lists, especially if the list items are "heavy" to render.
- **maxToRenderPerBatch** and **updateCellsBatchingPeriod**– these parameters control how FlatList performs*throwing in*elements when scrolling.maxToRenderPerBatch(default 10) means the maximum number of new elements to render per one list wake-up (i.e. when we approach the end of the currently rendered area, how many more elements to attach).updateCellsBatchingPeriod(default 50ms) is the delay between such render batches. IncreasingmaxToRenderPerBatchwill reduce the chance of empty spaces (because we're attaching more at once), but may cause a longer one-time lag (because suddenly 20 elements are being built). Reducing this will make the scroll more responsive (smaller render blocks), but may cause a momentary blankness if the user scrolls faster than these batches are attaching.updateCellsBatchingPeriodIn turn, a smaller value = we add elements more often (more fluid, but a constant higher CPU load), a larger value = less often (more economical, but the risk of lag). In practice, these values are rarely changed from the defaults, unless profiling reveals a specific problem.
- **getItemLayout**– a function we can provide if our list items have a fixed, known size (height) in advance. This allows FlatList to calculate scroll position and offsets in advance, eliminating the need to measure items during rendering. Setting getItemLayout with fixed heights*significantly*improves the performance of the list,

because RN does not have to render each element at least once to know its height (this eliminates the so-called*layout pass*for offscreen elements). In the getItemLayout definition we specify: for index -> {length, offset, index}. For example, if each element has a height of 50, the offset is index*50, length=50. FlatList will use this to, for example, quickly scroll to the element (scrollToIndex) without estimation error.

To sum up, **FlatList**offers a lot of tuning options. It's good practice to profile the scrolling list (e.g., in the Flipper Perf monitor) and adjust the above parameters if necessary. RN documentation officially lists these props as helpful in improving performance. In short:

- *Do you have lag when scrolling?*– make sure removeClippedSubviews = true (especially Android) and possibly reducemaxToRenderPerBatch.
- *Do you have blank areas when fast scrolling?*– increasewindowSizetheinitialNumToRender, alternatively maxToRenderPerBatch.
- *Is the app consuming too much memory on the list?*– reduce windowSize, removeClippedSubviews to true and try to reduce the complexity of rendered elements.

List Item Tips: Even the best-configured FlatList can become slow if individual list items (list item components) are "heavy." A few tips:

- Make sure your list components are**as simple and light as possible**Avoid complex subtrees with multiple rendering conditions. Each list item is rendered multiple times (when scrolling), so their optimality is critical.
- Consider usingReact.memofor the list item component to not re-render if the props haven't changed. In conjunction with the appropriatekeythis will prevent already rendered elements from being refreshed unnecessarily when new ones are added.
- If the list has images, use**thumbnails**(thumbnails) or lazy-load mechanisms for images. Large images slow down scrolling (they take up decoding time and memory space). It's better to display smaller versions and only load the large image after entering details, for example.
- If you can,**pagination**: instead of holding 1000 items at once, load them in chunks (e.g. 50) and load more as the user nears the end (FlatList offers propsonEndReachedfor this purpose). This reduces the size of the list at once.

4.2 Avoiding unnecessary renders and layout shifts

Layout shifts This is a term familiar from the web (Cumulative Layout Shift) – it refers to sudden jumps in layout when elements change. In the context of RN, we don't count points for layout stability, but we also want to avoid situations where the interface "jumps" or where we perform expensive layout operations unnecessarily. A few tips:

- **Fixed dimensions or use<FlatList>instead of manual mapping.**When we use FlatList, RN knows it's a list and manages the layout efficiently. If we do it ourselves in the componentdata.map(item => <MyItem ...>)inside a ScrollView, we lose virtualization and can also cause frequent layout recalculations (ScrollView renders everything at once).

So, basic advice: use FlatList for larger lists. If for some reason you can't (e.g., you need a non-standard layout), consider<VirtualizedList>or sections.

- **Avoid unnecessary state causing global re-render.**For example, if we have a large screen with a list and a small toggle, try to ensure that changing the toggle doesn't cause the entire list to be redrawn. This can be achieved, for example, by separating the list into a separate component and usingReact.memoor using dedicated state (e.g., Redux slice) so that changes unrelated to the list don't affect it. In general, keeping a global minimal state and preferably local states for UI elements will help limit the scope of renders.
- **Layout animations:**If we add/remove elements from the DOM, the layout will obviously change. This can be mitigated by using, for example:*Layout Animations*that will smoothly implement the change, or by planning the UI so that major changes occur when the user expects them (e.g. switching to another screen, instead of dynamically on the same screen).
- **Re-render and leaves:**A very common reason for slowdowns is that the parent component of the list renders frequently (e.g. due to a clock state change or something) and gets a new one each timedatalist (e.g., a new array reference), which makes FlatList think the data is "different" and redraws some elements. To prevent this:
    - If we generatedatain the renderer, e.g.const data = items.filter(...).map(...);, it's worth usingyou are in Memoto memorize the result, or move it higher and pass it on to the finisheddatafrom a parent that doesn't render as often.
    - FlatList also has propextraditedto track additional changes – if we can't tell what changed, but we know that, for example, state X affects the list, we can put it there. However, it's better to control the data references.
- **View size**: Make sure you style your lists and items so they don't require complicated*layout calculation*by Yoga (RN layout engine). For example, too nested flex layouts with components controlled by dynamic changes can burden the CPU. Profiling (Flipper Perf -> CPU usage) shows when after some action the CPU increases because the UI thread calculates the layout – this is a signal that either too much of the screen is changing, or the style is suboptimal (e.g., a lotshadowOffsetand shadows can burden the GPU/CPU when moving elements).
- **Removing elements from the DOM**: If something is invisible, you can remove it instead of hiding it (e.g. remove modal screens when they are closed, instead of keeping them all in the DOM withdisplay:none). RN doesn't really existdisplay:none– if you don't render something, it's removed. But in navigation, for example, the stack remembers screens. In the case of TabNavigator, for example, tab screens can be "lazy" by default, meaning they're mounted on demand, which is fine.

Generally, *avoiding unnecessary renders*comes down to using the above-mentioned techniques:**memoization**(React.memo, useMemo, useCallback) – to avoid generating new references when not necessary,**separation of states**– so that local change does not have a global impact, and**profiling**– to know where the problem is. React DevTools Profiler is your best friend here: it will let you see, for example, that the list component is rendering 5 times during a single action, which is unnecessary.

4.3 Frame-based vs. Native Animation – An Introduction to Reanimated

Animations in mobile apps can easily become a performance bottleneck if not done correctly. In React Native, we've traditionally had a library**Animated**(Animated API) and the ability to useLayoutAnimation Whether InteractionManagerHowever, a major change occurred with the advent of the library.**React Native Reanimated**(especially in version 2 and higher), which allows you to create smooth animations performed on the native side.

First, let's explain the concepts in the topic:**frame animations** vs **native animations**. This can be understood as follows:

- *Frame-by-frame animations*– here I understand it as animations controlled on each frame by JavaScript. That is, our JS code calculates where the object should be in a given frame and sets the style (e.g. changesleftevery 16ms). Such animations burden**JavaScript thread**and are prone to dropped frames if JS is busy with other things (e.g. calculations, rendering). An example of frame animation would be usingsetIntervalto change state, which causes re-rendering with a slightly changed position - this is extremely inefficient, because each frame is a new React render. Or using Animated without the native driver option (in older RNuseNativeDriver: falsecauses the animation to happen in JS - the style is updated via bridge every frame).
- *Native animations*– animations that are executed on the native side (on the UI thread or another native thread), without engaging the JS thread for each frame. In practice, this means: we define the animation flow in advance (e.g., "move an object from X to Y in 500ms with an ease-out curve"), pass it to the native layer, and it runs smoothly there, even if the JS stops. In RN Animated, the classic way isuseNativeDriver: true– but it had limitations (only some properties, no color or layout animation). Reanimated goes further – it allows you to define the entire animation logic and gesture response as*worklets*executed on the UI thread.

React Native Reanimated(2.x and 3.x) – what makes it special?
As the documentation states:*"Reanimated allows you to define animations in pure JavaScript that**By default, they run natively on the UI thread**"*This means that we write the animation code in JS, but thanks to the mechanism*worklets*(special functions with a note'worklet') this code is sent to the native environment and executed there every frame, without burdening the bridge or JS thread. This gives**smooth animations up to 60fps and even 120fps**on screens that support them, regardless of the JS load.

Frame vs. Native – Practical Effects:A frame-based (JS-driven) animation can start dropping frames if the JS is doing something heavy at the same time. For example, imagine animating a panel slide-out while simultaneously serving large JSON from an API on a JS thread – the animation may stutter because the JS can't keep up with sending a new position every 16ms. In a native (Reanimated) animation, this situation doesn't affect the panel – the slide-out will be smooth because the movement logic is separated. Therefore, all important UI animations should be native to ensure smoothness. This also applies to gesture responses – for example, smoothly dragging an element with a finger: Reanimated, combined with the React Native Gesture Handler, can completely transfer gesture and animation handling to the native page, eliminating lag.

Example of difference: Let's take a simple animation – moving a square 100px to the right. In pure RN Animated (up to RN 0.71 or so):

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

This animation will send a new value every frame (every ~16ms)`position`through Bridge to native. If Bridge gets clogged or JS is late – there will be a jump. When we give`useNativeDriver: true`:

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

The RN will send the message "animate this value from 0 to 100 in 500ms" to the native Animated module, and the native code (iOS/Android Animator Core Animation) will do the rest. This is native animation and should be smooth. The problem is that the old Animated with the native driver only worked for certain styles (mainly translations, scale, opacity). It can't animate, for example, background color or positioning dependent on the Flexbox layout.

Reanimatedthere are no such limitations, because it works differently: we have*shared values* i *worklets*, you can animate any style, because animation is really just a function that changes a value and then assigns it to a style in the native Shadow Tree. Reanimated 2+ integrates with the UI runtime engine. As a result, we can, for example, animate position based on sensor values (accelerometer), and we can create complex sequences and nested animations – all within a native context.

In terms of performance:

- Native animations (Reanimated, or Animated Native Driver) impose virtually no JS overhead during execution. They do burden the native UI thread, but this is C/C++ and very efficient, plus it can leverage platform optimizations (e.g., iOS does this with CoreAnimation).
- Frame-based animations (JS) burden JS and Bridge – two bottlenecks in RN. It's best to avoid them, because even if one works, they'll crash when multiple simultaneous animations are running.

In 2025,**Reanimated has become the de-facto standard**for complex animations and interactions. Many libraries build on it (e.g., the well-known bottom-sheet library "react-native-bottom-sheet" uses Reanimated). It's worth knowing at least the basics:

- Concepts*useSharedValue*(animated shared value),

- *useAnimatedStyle*(hook that allows you to bind the component style to an animated value),
- *animation sets*typewithSpring, withTiming– functions for updating shared values with animation effects (spring, timing).
- Worklets'worklet'– i.e. functions that execute on the UI thread. For example, callback in.onChangegesture equipped with a 'worklet' can directly controlsharedValueand it will be native.

A quick example of Reanimated (version 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={() => {
// this 'animates' natively, without blocking JS
        offset.value = withSpring(offset.value + 100);
      }} />
    </View>
  );
}
```

Here clicking the button changes theoffset.valueusingwithSpringThis won't trigger a React re-render immediately (Animated.View will pick up the change itself), but it will initiate a native spring animation – the red cube will move smoothly by 100px. The JS thread only initiated the animation; everything else happens in the native (UI) thread. When the animation finishes, Reanimated can signal the JS (but it doesn't have to, it depends). Importantly, we can even stop the JS during the movement (e.g., enable the dev menu) – the animation will complete anyway.

Animations and frames (FPS):

- Smooth animation is 60 FPS (on 60Hz screens) or 120 FPS (on iPad Pro, e.g.).
- Frame animation, if JS cannot keep up, may drop to 30 FPS or less (you will then see "jerking").
- Reanimated strives to deliver 60 FPS even in the most challenging scenarios. Of course, if we perform*extremely*Heavy worklet stuff (e.g., a 1e6 iteration loop every frame—which is unlikely to happen) can result in dropped frames natively. But normal transformations and springs are nothing for modern CPUs, and native animations often use the GPU for final rendering.

Complex animations: In addition to single style animations, Reanimated also offers*Layout Animations*(allows you to animate elements during mount/unmount), integrates with Gesture Handler (gestures also on UI thread – zero lag in drag & drop), and allows you to create your own threads (so-called*worklet runtime*) for large background calculations. The latter is interesting – for example, you can calculate something heavy in a worklet (like in the example sum 1..1000000) and then pass the result to JS without freezing the application.

In this lecture it is**entry**to Reanimated – I encourage you to continue learning, as the topic is extensive. The most important thing to remember:**we are moving towards native animations for better performance**Reanimated is a tool that allows for this in a very flexible way (we can animate almost anything stylish, respond to gestures smoothly). In modern RN applications, Reanimated often replaces the old Animated library, although RN still maintains the Animated API (perhaps implemented in the future based on Reanimated underneath – there's already discussion about bridging these worlds).

Summary of this section: Frame-wise animations (i.e., those dependent on JS on a frame-by-frame basis) should be kept to a minimum. We use native mechanisms – whether olduseNativeDriverfor simple cases, and preferably Reanimated for full control – to ensure**smoothness 60fps**regardless of the application's logic load. This will make the interface appear modern and responsive. And nothing ruins mobile UX like jerky animations/scrolling like a slideshow – that's what we want to avoid.

5. Demo – List Refactoring and List Component Test

Finally, let's run a small demo that combines several of the concepts discussed: we'll show**refactoring an existing list**to use FlatList with performance optimization, then**we will write a test**for a list component, checking rendering and interactions.

Scenario: Let's assume we have a task application (TODO list). So far, the task list has been implemented naively – as a simple<ScrollView> With .mapor even in the parent component. We want to improve this, use FlatList, and apply some performance parameters. Additionally, our list has the ability to mark a task as completed when clicking the "Done" button next to the task. We'll test whether clicking actually calls the appropriate function (e.g., passed from props).

Code before refactoring (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>
 );
}
```

This code works for small lists, but if`tasks`will be large, then ScrollView will render everything at once. Let's use 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 screens (2 before, 2 after, 1 present)
   maxToRenderPerBatch={5}
   removeClippedSubviews={true}
  />
 );
}
```

What has changed:

- We use`<FlatList>` With `data={tasks}` i `renderItem`to determine how to render a single task.
- We added`keyExtractor={(item) => item.id}`for FlatList to use`id`task as a key. We are sure that the keys are unique (the task ID is unique).
- We set some props:`initialNumToRender={10}`(render 10 tasks at once – we assume this is enough to cover the screen of a typical phone),`windowSize={5}`(this means 5 windows in the height of the screen – 2 at the top and 2 at the bottom, which in total gives about 2*10 + 10 = 30 items in the buffer; we reduced it a bit from the default 21 to save memory, because the task list is not scrolled ultra-

fast),maxToRenderPerBatch={5}(to not add more than 5 elements at a time, which will reduce possible lag when scrolling quickly),removeClippedSubviews={true}(to remove off-screen elements from view – especially useful if our list has, for example, a lot of images, although here only texts and buttons).

- InsiderenderItemwe addedtestIDfor the Done button, which will make it easier for us to select in tests (e.g.testID="done-btn-42"for the task with ID 42). It's a practice worth using – generate testID containing the element ID if we click on a specific element in the list during testing.

In terms of style and layout –styles.taskItemprobably defines something like flex row: Text + Button next to each other. The important thing is that each <View> has akeyby FlatList (it does it itself based on keyExtractor) and that we make sure that the key is stable and does not change when the task changes e.g. name.

Advantages after refactoring: Now even iftaskshas 1000 items, FlatList will not render all of them – only ~30 (depending on the height of a single element, initialNumToRender and windowSize). Scrolling will be smooth because we don't keep all the elements in memory at once. We are also sure of unique keys thanks tokeyExtractor.

Writing a list component test (render + interaction):

We want to test that the TaskList displays the correct number of items and that clicking "Done" on a specific task calls the functiononTaskDonewith the correct argument (task id). We can do this using the 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: 'Buy milk', done: false },
{ id: '2', title: 'Pay bills', done: false },
{ id: '3', title: 'Task completed', done: true }
];

test('renders all tasks in the list', () => {
  const { getByText } = render(<TaskList tasks={sampleTasks} onTaskDone={() => {}} />);
// We check for the presence of task titles:
expect(getByText('Buy milk')).toBeTruthy();
expect(getByText('Pay bills')).toBeTruthy();
expect(getByText('Task completed')).toBeTruthy();
// We check that there is no "Done" button for a completed task:
  expect(() => getByText('Done')).not.toThrow();
// NOTE: the above line is tricky - getByText('Done') will find the *first* match,
// and we have two unfinished tasks, so there will be two "Done" buttons.
// It's better to use queryAllByText:
  const doneButtons = queryAllByText('Done');
  expect(doneButtons.length).toBe(2);
```

```
});
```

The above test renders a list with three tasks. We check that all title texts are present (meaning that the list renders every item). Next, we want to make sure that the item marked as done (id 3) does not display a "Done" button. In our component code, we conditionally render the button only if`!item.done`So for 2 tasks done=false the button will be there, for one done=true it won't. In the test we could try to find`Done`for everyone, but it's better to collect all the "Done" buttons and check their number. We use`queryAllByText('Done')`– this will return an array of matching elements or an empty array if none. We should get 2 elements (for id 1 and 2). We check`.length === 2`. Alternatively, we could do:

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

but it's complicated. Counting is simple.

Then the interaction test:

```
test('calls onTaskDone with the correct ID after pressing the Done button', () => {
const mockOnTaskDone = jest.fn();
  const { getByTestId } = render(<TaskList tasks={sampleTasks} onTaskDone={mockOnTaskDone} />);
// For example, we take the first task (id '1') and click its button
  const button = getByTestId('done-btn-1');
  fireEvent.press(button);
  expect(mockOnTaskDone).toHaveBeenCalledWith('1');
  expect(mockOnTaskDone).toHaveBeenCalledTimes(1);
});
```

Here we create`mockOnTaskDone`as a test function (spy). We render a TaskList from sampleTasks. Thanks to the fact that in the TaskList we have assigned a testID for each button containing the task ID (`done-btn-${item.id}`), we can easily refer in the test, for example, to`done-btn-1`for the first task. We retrieve this element by`getByTestId`and we simulate`fireEvent.press`. Then we check that`mockOnTaskDone`was called once, and with the argument '1'. This ensures that the ID passing mechanism worked. This can be repeated for another element (e.g., id '2').

This test verifies the component's integration with the logic—that is, whether the UI correctly calls the passed function with the appropriate parameter. In a real application`onTaskDone`could, for example, set state or send a Redux action. Here we just check that it was called.

Running tests: After writing the above tests (e.g. in the file`TaskList.test.js`), we use`npm test`. We assume that the configuration is set (Jest, Testing Library). The tests should pass as long as the component works as intended. For example, if we forgot to add`testID`to Button, that`getByTestId('done-btn-1')`that it does not find the element and the test fails – this is a signal that the implementation needs to be improved (add testID or find another way of identifying it, e.g. getByText + within a specific item, but testID is the most convenient).

Demo Summary: We showed how to move from a suboptimal list implementation to a more efficient one using FlatList with the right settings. At the same time, we paid attention to testability: we added`testID`to make it easier to select elements in tests, and we left the entire

click logic in props (onTaskDone), which is in line with the principle of making UI components as simple as possible*clean*and externally controlled – making mocking and testing easier. Our list component test confirmed both data rendering and correct interaction handling.

Literature:

1. **https://reactnative.dev/docs/optimizing-flatlist-configuration**(Accessed: 1/10/2025) – Official React Native documentation on list optimization, detailing parameters such as`windowSize`, `initialNumToRender`and techniques to improve scrolling smoothness.

2. **https://wix.github.io/Detox/docs/introduction/getting-started**(Access date: 1/10/2025) – A guide to the Detox framework, explaining the end-to-end gray-box testing architecture and automatic synchronization mechanisms with a React Native application.

3. **https://fbflipper.com/docs/features/react-native/**(Accessed: 1/10/2025) – Flipper documentation, describing debugging features including Layout Inspector, Network Inspector, and integration with the Performance plugin for mobile apps.

4. **https://testing-library.com/docs/react-native-testing-library/intro/**(Access date: 1/10/2025) – React Native Testing Library (RNTL) documentation, presenting good practices for testing components and hooks based on user interactions.