MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Comparison of Technologies for Multiplatform Mobile Applications Development

Master's Thesis

BC. JAN CRHA

Brno, Fall 2021

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Comparison of Technologies for Multiplatform Mobile Applications Development

Master's Thesis

BC. JAN CRHA

Advisor: RNDr. Vít Rusňák, Ph.D.

Brno, Fall 2021


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Jan Crha

Advisor: RNDr. Vít Rusňák, Ph.D.

Acknowledgements

I would like to thank my advisor RNDr. Vít Rusňák, Ph.D., for valuable advice and always being available when I need guidance. I would also like to thank my family for being supportive throughout my studies.

Abstract

The thesis deals with the modern cross-platform mobile applications development following the "write once, run everywhere" approach. It focuses on comparing two distinct concepts: development of native cross-platform applications, i.e., those that compile to the native code for given platform and Progressive Web Applications. Besides the overview of used technologies, I designed and implemented three non-trivial applications (having the same functionality) using React Native and Flutter. Next, I performed a series of experiments focusing on objectively measurable metrics such as CPU and memory utilization. I also provided a subjective assessment of the development complexity.

Keywords

mobile application, cross-platform development, Android, iOS, React Native, Flutter, Progressive Web Application

Contents

1 Introduction

1.1 Motivation

Over the last decade, smartphones became an indispensable part of most people's everyday life. More than 6.37 billion people (over 80% of the world population) own a smartphone and that number is expected to grow in the following years [22]. With an audience this wide, it is no surprise that mobile applications are a crucial software development segment, and the market is highly competitive. Currently, there are two dominant mobile operating systems – Android and iOS, having more than 99% market share [30] together.

These platforms are not compatible, and any application developed natively for one of these platforms will not run on the other. Originally, the only option was to develop separate applications for each targeted platform. Such approach is very demanding on human and time resources. The company either needs to find people proficient in development for both platforms, or they need two separate development teams, one for Android and one for iOS. However, in recent years, we saw the rise of many technologies that aim to provide some level of cross-platform functionality and reduce the development and maintenance cost.

This thesis aims to overview cross-platform mobile applications development, with the focus being on two main philosophies: Compile-to-Native Applications, and Progressive Web Applications. In the practical part of the thesis, I selected React Native and Flutter frameworks, to implement three variants of a non-trivial example application. The goal of the implementations was to create as similar versions of the application as possible. These implementations were then thoroughly compared in objective measures such as performance or resource utilization. I also report on personal experience with the application development.

Chapter 2 overviews different possibilities of mobile applications development. The focus is on how they work, what are the development requirements, and what are the benefits and limitations of described methodologies. Chapter 3 presents the technologies used for the practical part of the thesis. The architecture of React Native and Flutter, the approaches to state management, notable specifics, and the size of their communities are discussed. The sample application, its requirement analysis, design, and implementation details are presented in Chapter 4. Chapter 5 presents the evaluation of selected performance metrics and resource utilization and personal assessment with the application development. Finally, Chapter 6 summarizes the thesis and provides recommendations about selecting the suitable mobile development technology for specific use cases.

2 Mobile Applications

The focus of this chapter is to describe different methodologies for mobile applications development.

In the first part of the chapter, I overview the iOS and Android platform specifics. Next, I focus on three main philosophies to cross-platform development: *hybrid*, *compile-to-native*, and *progressive web applications*.

2.1 Platforms

Native applications are created to target specific operating systems. Their development should follow guides created by maintainers of the mobile platform, and these applications are specifically optimized to run in that platform environment. The benefits of native development are that native applications can easily take full advantage of all the *Application Programming Interfaces* (*APIs*) provided by the platform. Both operating systems (iOS and Android) offer APIs that allow developers to access device information, hardware capabilities, system storage, secure storage for sensitive information, and many more. Another benefit is that roughly two-thirds of all mobile application developers are doing native mobile development [21]. This means that the market with native developers is much bigger than with cross-platform developers.

2.1.1 iOS

iOS is the operating system developed by Apple Inc. for their smartphones[1]. To create an application for iOS, the developer must have access to a Mac computer with XCode *Integrated Development Environment* (*IDE*) and build tools installed. Also, if the developers want to publish the application to Apple App Store, they will have to create a paid Apple Developer account.

The two languages that are primarily used for iOS development are Objective-C and Swift. Apple officially recommends these two languages, and most of the APIs available for iOS are implemented for both Objective-C and Swift. Developers can also leverage their knowledge of C++ to create native libraries. Still, for *Graphical User Interface* (*GUI*) developers usually choose either Objective-C or Swift.

That said, Swift is the more modern option getting more promotion from Apple. It is newer, has many contemporary features, and provides full compatibility with Objective-C APIs. The fact that Swift code has full access to the existing Objective-C APIs means that developers can slowly adapt Swift to their existing Objective-C code without rewriting the whole codebase at once [12].

Historically, one of the most significant pain points of iOS development was UI development. The developer had a choice between writing the UI programmatically (using UIKit framework), via the Interface Builder or Storyboards [28] since iOS 5.

In the 2019 Worldwide Developers Conference (WWDC), Apple introduced a new declarative way of building UI called the SwiftUI, which takes inspiration from declarative UI frameworks such as React. SwiftUI have become the officially recommended way of creating a user interface [13].

Another big pain point for iOS developers is that Apple often introduces breaking changes in new iOS versions. Apple is known for being user privacy-oriented which is generally a good thing. Still, it means new privacy permissions are often introduced to the system, and developers need to adjust their applications in the short time window between the announcement and release of the new iOS version. Not only do developers need to update their applications to work with the new permissions, but these new permissions result in serious system bugs from time to time. One example is the Local Network privacy permission introduced in iOS 14. After the system update was released, some developers quickly noticed a scenario that prevented users from connecting to devices on the local network through their application [11]. If users denied the Local Network permissions for the application and reinstalled the app, they did not have the option to enable it again. There was no way to fix this until Apple released a system patch.

1. Until 2019, iOS was also the operating system of iPads. After that, it got re-branded as iPadOS.

2.1.2 Android

Android is an open-source operating system based on a modified version of the Linux kernel. The leading developers of the Android platform are Google and the Open Handset Alliance [23]. Android is primarily used for touchscreen devices such as smartphones or tablets, but it has been adjusted to support various other devices such as televisions [1].

The two main languages used for the development of Android applications are Java and Kotlin (which replaced Java as the preferred language for Android application development in 2019 [23]). That said, some other languages that provide lower-level functionality, such as C++, can also be used to develop specific parts of the application logic.

Unlike iOS development, which requires a Mac computer, Android developers can build their applications on many different operating systems and are not limited to specific IDE.

One of the biggest hurdles in Android development is its fragmentation. There are multiple definitions. The most common interpretation of Android fragmentation include the necessity to support the magnitude of multiple different versions of the Android operating system due to slow adaptation and release cycle of new versions and shorter – compared to iOS devices – guarantee to receive updates on older devices. The second most common interpretation of the Android fragmentation issue is the multitude of different devices and manufacturer modifications to the Android system [4]. This can become a real struggle for developers because code that works on most devices running on the Android system can still produce subtle bugs in some specific device types. To mitigate this issue, developers usually resort to testing their applications in Android emulator[2] on a variety of exotic virtual devices.

2.2 Cross-Platform Development Philosophies

In contrast to native mobile applications that target one specific platform, cross-platform mobile applications target two or more operating systems. Usually, the main targets are iOS and Android. Still, some technologies attempt to cover a broader spectrum of platforms and aim to achieve the "write once, run everywhere" approach by compiling to the web and computer targets (e.g., Windows, macOS, or some Linux distributions) as well. There are many technologies and frameworks that aim to make cross-platform development more accessible and more widely adopted, but most of these frameworks end up only being used by relatively small communities of enthusiasts.

The benefits of cross-platform development are mainly code sharing and ease of writing. Most of these technologies allow developers to share a significant portion of code between all the target platforms they build the application for [2]. However, developers still need to write platform-specific code when the application requires access to low-level device capabilities. Regarding the ease of writing, most cross-platform frameworks use programming languages widely known among developers, such as C#, JavaScript, or TypeScript. The two latter make it even easier for web developers to develop mobile applications using cross-platform frameworks because these languages are mainstream for web development.

There are three main concepts of cross-platform development: hybrid, native cross-platform and progressive web applications. The details are discussed in the reminder of the section.

2.2.1 Hybrid Mobile Applications

Hybrid mobile applications combine aspects of both native and web applications. Hybrid applications compile to a native container that usually uses a WebView native element to display content to the end-user. The application is installed to the device just like any other native application, but usually, it is simply displaying web pages created through web technologies[3].

In case when the application needs to access the device's hardware features or connect to some external device, it is possible to include native code for each target platform and create a "wrapper layer" to call this code in a unified way. Some device hardware APIs can be called directly through

2. Android emulator is a software that emulates smartphone environment with Android operating system on the desktop and enables the application development without the need to access a physical device.
3. E.g., CSS (Cascading Style Sheets), JavaScript and HTML5

JavaScript APIs using the WebView. For example, to access the device camera or GPS module, existing JavaScript APIs can be used, and no specific native code is required.

Some of the benefits of hybrid mobile applications are:

- Avoiding the necessity to go through Apple App Store and Google Play Store reviews. These reviews are time-consuming, and quite often, the application gets rejected for some bureaucratic reasons. The developer then needs to fix these issues and resubmit the application to go through the process again.
- Reducing the number of developers necessary. One team can work on both iOS and Android versions of the application since they share most of the code.
- It is easier to find developers with web development background than finding iOS and Android developers for two separate teams.

On the other hand, hybrid mobile application development also has many downsides. Most notably:

- A limited user interface. Since the application does not use native system elements but instead displays a web page, it is harder to achieve the "native" application user feeling.
- The speed of development and code-sharing falls short for mobile applications once the application needs to communicate with native hardware, which isn't exposed through standard JavaScript APIs. In that case, the development team needs to write platform-specific code, which can be a problem if the whole team has little native development experience.
- Slower performance. Hybrid applications can only easily achieve a genuinely native feeling for small projects. Once the application grows in size and complexity, it starts to be hard to keep the application performant and on par with native applications.

Currently, hybrid mobile development frameworks seem to be in decline. In 2020, Adobe decided to discontinue the development of the PhoneGap framework and PhoneGap Build service, which was used by the most common hybrid mobile application development framework Ionic[4].

2.2.2 Compile-to-Native Applications

Native cross-platform mobile development frameworks are still a relatively new concept emerging in recent years. These technologies usually generate platform-specific code necessary to build the application for each target platform. This generated code does not need further modifications unless the application needs to access some native libraries or APIs that are not provided by the framework out of the box (e.g., libraries used to connect to external devices such as payment card readers).

In case that the application needs to access some native library or API that is not provided by the framework, the developer can resort to writing platform-specific code and creating a *wrapper* API for unified access, similarly to hybrid frameworks.

Except for the platform-specific generated code, the developer can use one codebase for the whole multi-target project without the need to worry about the compatibility of specific platforms. What differentiates this approach from hybrid applications is that the application is not running in a WebView container but is actually translated to truly native platform-specific components. The fact that the code renders native UI components means that the application appearance and behavior in an ideal scenario is indistinguishable from native applications written for a specific platform.

The main advantages of native cross-platform mobile development worth mentioning are:

- Native look and feel – native cross-platform applications usually translate to truly native components, creating a user interface and user experience that is not distinguishable from a native application is more effortless.
- Flexbox – most of the frameworks support *Flexbox Layout* allowing container elements to adjust dimensions of their children to fill the available space efficiently. In principle, it works similarly to CSS. Using flexbox makes it possible to achieve the same application look on many different screen sizes, which can be a complex issue in native applications.

4. https://ionicframework.com/

- Performance compared to hybrid ones – unlike hybrid mobile applications, native cross-platform applications are not containerised, therefore more performant.

The main disadvantages are:

- Novelty – the majority of these frameworks started to be widely used in production-grade applications only 3–6 years ago. Not all issues are discovered or fixed. Also, the documentation can be missing for some specific low-level parts of the framework.
- Community size – the size of the communities of developers for these frameworks are smaller than those for truly native applications.
- Performance compared to truly native applications – similarly optimised truly native applications will, in most cases, still outperform native cross-platform applications[5].

Compared to hybrid mobile development frameworks, the native cross-platform frameworks seem to be gaining a lot of traction lately. The most known examples of this approach, such as React Native and Flutter, are backed by some of the biggest companies. These frameworks are used to build production applications for many large companies. And furthermore, they have a significant community that is growing fast and helps improve these frameworks at a rapid pace.

2.2.3 Progressive Web Applications

Progressive Web Applications (PWAs) are not built with a single technology. Instead, they represent a new philosophy for building web applications that involve specific patterns and APIs. It is usually not obvious if a web application is PWA or not just by looking at it. The term *Progressive Web Application* is not an official name. Google first used it for the concept of creating flexible and adaptable applications using only web technologies.

For an application to be considered PWA, it has to meet some requirements. For example, it should be installable to the device's home screen and provide some basic offline functionality. At a minimum, it should load even without an internet connection and show some sensible offline fallback screens [7].

To achieve these requirements PWAs leverage specific technologies and architectural patterns, e.g., Service Workers[6], which are most commonly used to provide an offline experience, background synchronization, or push notifications, or Progressive Enhancement, which is a design philosophy that focuses on progressing the user experience up. For users of newer web browsers and devices, the web application may appear differently and more fully featured than for users of older browsers.

Progressive Web Applications are partially responsible for the decline of hybrid cross-platform applications. They provide similar functionality without the need to run in native WebView containers. Adobe specifically mentioned that one of the reasons for shutting down PhoneGap was the rise of PWAs [9].

The main benefits of Progressive Web Applications are:

- Decreased load times after the application has been installed to the user's home screen. PWAs use caching with Service Workers to cut loading times and save resources.
- The ability to update the application's content without the necessity to go through Apple App Store and Google Play Store reviews.
- No experience with native development is required. PWAs are web applications without any native parts. This means that no specialized knowledge for the underlying platform is necessary.

Although PWAs are on the rise and are widely used, there are also some pitfalls and limitations. Most notably:

- PWAs can only access hardware/operating system features that are provided by web browser APIs. PWA will no longer be an option if the application needs to use some specific hardware such as payment card terminals.

5. This is not necessarily always correct but it is a common argument against native cross-platform technologies.
6. Service Worker API is a web API that acts like a proxy server between web applications, the browser, and the network.

- Compatibility with iOS. PWAs are supported since iOS version 11.3. Additionally, Apple only allows PWAs to be installed if the application is accessed through its default web browser Safari. It also does not provide APIs for Bluetooth or NFC (Near-Field Communication) capabilities.
- Lower performance compared to native applications. Since the PWAs run in the web browser, the additional overhead negatively influences the application performance.

3 Used Technologies

This chapter describes the technologies selected for comparison in the assignment of this thesis: React Native, Flutter, and PWA. The focus is on the architectures, state management, framework specifics, and communities around the framework.

3.1 React Native

React Native is maintained by Meta (former Facebook) with strong community support. It was released to the public in 2015. React Native is based on the successful React web framework and uses similar concepts. It is one of the most known and used frameworks for native cross-platform development nowadays [21].

3.1.1 Architecture

The programming languages available for React Native development are JavaScript and TypeScript. CSS is used to define user interface styles. The core of React Native architecture consists of three core parts – a native part, an asynchronous bridge, and a JavaScript part (Figure 3.1). The native part is where the native code for the application resides, and the JavaScript part is where the JavaScript code (and most of the logic developers write) lives. The bridge provides the ability for these two parts to communicate through asynchronously serialized JavaScript Object Notation (JSON) messages. React Native uses three threads:

- *Main thread* (also called UI thread) which is the only thread that can make changes to the user interface;
- *JavaScript thread*, which is responsible for running the JavaScript code and React code, and is executed by a JavaScript engine such as JavaScriptCore or Hermes, which is an open-source JS engine developer by Facebook and optimized for React Native applications [16]; and
- *Shadow thread* which is a background thread that is responsible for a recalculation of the UI and layout and passing new data to the *Main thread* [14].

Similar to React, React Native uses uni-directional data flow. It maintains a component[2] hierarchy, where each component depends only on a parent component and its state. This means that data flows only from a parent to children. In case that the parent component needs to update based on some changes in a child component, the parent needs to pass a callback function as property to the child.

React Native provides access to a subset of the device hardware and APIs (and support for many other capabilities is available through community-maintained packages). However, the support for

1. Source: `https://collectivemind.dev/blog/react-native-re-architecture`.
2. Component is virtually a UI element or element that provides some extra functionality to other components.

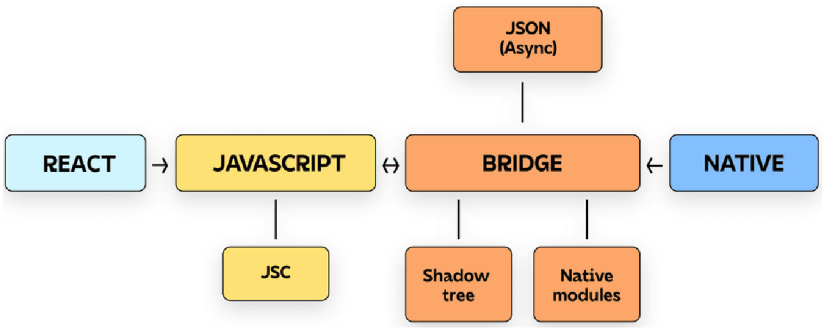

Figure 3.1: Current React Native architecture[1].

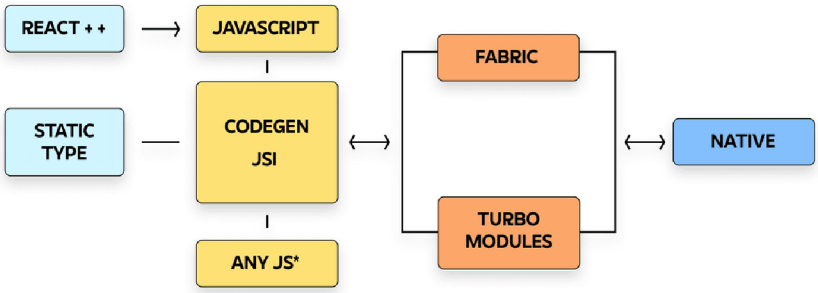

Figure 3.2: React Native upcoming architecture[4].

many low-level device capabilities is not present in the framework. If developers need to work with APIs that are not provided through the framework, they can create a so-called Native Module. Native Modules are React Native mechanisms that allow creating a wrapper around custom native code to be accessed from the JavaScript side of the application [17].

3.1.2 Incoming re-architecture

Over the lifetime of React Native, both the community and the core team realized some limitations in the original architecture. React Native user-written JavaScript code is single threaded[3]. Although not ideal, developers can work around this limitation by creating custom native modules for performance-critical functionality. The bigger problem is that the communication between JavaScript and Native (including native modules) is done asynchronously through the bridge. Having a bridge that serialized messages between two sides creates overhead of copying of data. There are more problems with the bridge though. Since the communication is asynchronous, there can be delays before the data reach the other side. Another issue with data serialization through the bridge is that it can become a bottleneck when data flows faster than it can be serialized and transferred. This can result in JavaScript thread frame drops, issues with UI updates, and user action handling.

In order to get rid of the problems surrounding the bridge, React Native team designed a new architecture. The goals were to improve overall performance and improve type safety between JavaScript and Native parts of the application. The new architecture completely removes the bridge and replaces it with a new component called the JSI.

JSI (JavaScript Interface) is a layer between JavaScript and Native parts of the application responsible for direct communication between these parts without any need to serialize messages. JSI enables JavaScript to hold direct references to C++ objects in the native world and invoke methods on them. A side effect of the JSI introduction is that the JS bundler no longer depends on the JSC engine but allows React Native to use any other JavaScript engine.

Closely related to JSI is a tool called Codegen. It is not directly related to the React Native architecture, but it allows to automate and ensure build-time type compatibility between JavaScript and Native parts of the application.

The two remaining new components of the new architecture are called Fabric and Turbomodules. Fabric is the name of the new UI layer, which through JSI, directly exposes native UI operations as functions to JavaScript. Turbomodules are re-implementation of the old native modules. They bring performance improvements due to direct communication between JS and native, and another big difference is that they are lazily initialized, which, among other things, improves application startup time [14].

React Native developers posted a blog post where mentioned that the new architecture is ready, and they are already using it in Facebook applications. They are currently rolling the changes out to the community and creating migration playbooks for library maintainers [18].

3. Since JavaScript itself is single-threaded.
4. Source: `https://collectivemind.dev/blog/react-native-re-architecture`.

3.1.3 State Management

There are two types of data that control React Native components: *props* and *state*. Props are the implementation of uni-directional data flow in React and React Native. They are set by a parent component and can only be changed by the parent component. For data that can change from within the component itself, developers have to use the state. The state is defined inside the element to which it is relevant. For example, it can be used to hide parts of the UI based on some condition, store data from text inputs, or highlight some screen features in reaction to user actions (cf. Figure 3.3).

```
import React, { useState, useEffect } from "react"
import { StyleSheet, Text } from "react-native"

type Props = {
    text: string
}

export const Blink = (props: Props) => {
    const [isShowingText, setIsShowingText] = useState( initialState: true)

    useEffect( effect: () => {
        const toggle = setInterval( callback: () => {
            setIsShowingText(!isShowingText)
        }, ms: 1000)

        return () => clearInterval(toggle)
    })

    if (!isShowingText) {
        return null
    }

    return <Text style={styles.text}>{props.text}</Text>
}

const styles = StyleSheet.create({
    text: {
        fontWeight: "bold",
    },
})
```

Figure 3.3: Example of React Native component with local state[5].

State management can become a complex topic once the application grows in size. Components can hold a local state but some parts usually need to be shared through the component hierarchy. The naive approach is to pass the state (and function that can modify it) down the component hierarchy. This approach is called *props drilling*, and it can quickly lead to hard to maintain and refactor application architecture where developers pass props through many levels of the component hierarchy.

There are many libraries and approaches developed to simplify state management in applications. React Native provides some mechanisms such as the context API, `useReducer` or `useState` hooks API. However, they are suitable only for small applications. The large projects use some state management library such as Redux[6] or Recoil[7].

5. Source: `https://reactnative.dev/docs/state`
6. `https://redux.js.org/`
7. `https://recoiljs.org/`

3.1.4 Community

React Native's open-source philosophy is highly dependent on its community. The core team focuses primarily on architecture and core React Native design while relying on the community to help with bug fixes, library maintenance, and feature development. This is emphasized by an effort called *Lean Core* which aims to reduce the size of the React Native framework and move responsibility for the maintenance of many modules that were previously included in React Native to the community as external modules.

React Native's community is growing every year, and in the time of writing, React Native has more than 99 000 stars and 2257 contributors in its GitHub repository[8]. It has a large Reddit community with over 85 000 followers[9] and a StackOverflow community with close to 120 000 questions asked[10].

3.2 Flutter

Flutter is another open-source UI framework for native cross-platform application development maintained by Google with the help of the community. It is a reactive framework inspired by React and promotes declarative UI composition. Its goal is to enable developers to deliver performant applications that feel natural on different platforms while allowing as to share as much code as possible [26]. Flutter is often compared to React Native since they are the most used native cross-platform frameworks nowadays. In 2021, Flutter has overtaken React Native as the most used native cross-platform technology [21].

3.2.1 Architecture

Flutter uses the Dart language[11]. The core of Flutter architecture is an extensible, layered system where each layer is designed to be optional and replaceable. It has three core parts:

- *Embedder* – provides an entry point for the application, coordinate with the underlying operating system for access to various services such as rendering surfaces or accessibility, and managing of the message event loop. It is platform-specific.
- *Engine* – is responsible for drawing new frames and provides a low-level implementation of system APIs, including graphics, layout, and network I/O operations. It provides the primitives necessary to support all Flutter applications and manages the plugin architecture and Dart runtime. It is written mainly in C++.
- *Framework* – provides a set of platform, layout, and foundational libraries. Its core is relatively small, features such as camera or geolocation are provided as plugins [26]. It is written in the Dart language.

During development, Flutter applications run in a VM that supports a stateful hot reload feature (similar to React Native) without recompiling the whole application for every change. For release builds, Flutter applications are compiled directly to machine code instructions or JavaScript if the target for the application is the web.

Inspired by React, Flutter also follows uni-direction data flow philosophy where widgets (equivalents of React's components) only rely on data provided by the parent widget and optionally their own state [26]. Flutter widgets are represented by immutable Dart classes that, by composition, create the widget hierarchy tree. Flutter takes a step further from React components with its widgets. The philosophy is that everything in Flutter is a widget. Where developers would use styles in React Native, Flutter provides widgets to compose instead. For example, there are widgets such as Center, Align, Padding, Transform, and many others.

8. `https://github.com/facebook/react-native`.
9. `https://www.reddit.com/r/reactnative/`.
10. `https://stackoverflow.com/questions/tagged/react-native?sort=Newest&edited=true`.
11. Dart language is developed by Google and optimized for user interfaces and performance on any platform.
12. Source: `https://docs.flutter.dev/resources/architectural-overview`.

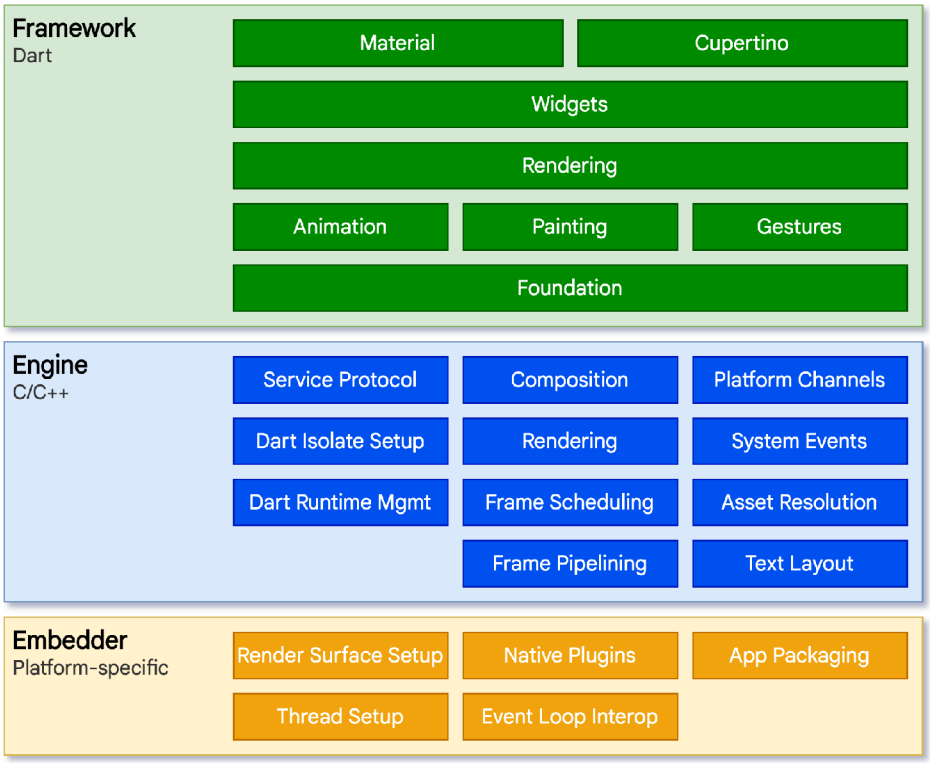

Figure 3.4: Flutter architecture[12].

In contrast to React Native, Flutter widgets are not translated to the underlying system UI elements. Instead, Flutter has its own implementation for each UI element. For example, there is a Dart implementation for both the iOS button and Android button. There are several benefits to this approach. For instance, if developers want to extend some UI elements, they are not limited by the API of the underlying operating system. Additionally, using this approach, Flutter does not need to transition back and forth between Dart code and native code, but it can compose the whole scene at once. One of the most significant benefits of this approach is that Flutter widgets look the same on all supported operating system versions, and developers do not need to worry about their UI changing with the new OS release.

In case that the application needs to integrate with some low-level system library or native SDK, developers have the option to create a *platform channel*. Platform channels are a concept similar to React Native's native modules. They allow exchanging serialized messages between native implementation and Dart code (see Figure 3.5). In addition to platform channels, developers have another option to communicate with C-based APIs (including C APIs that can be generated for code written in languages like Rust or Go). The option is FFI[14]. Dart provides a library for direct binding to native code. The FFI can be considerably more performant than platform channels because no serialization of data is required.

3.2.2 Rendering

Flutter rendering works differently from native mobile technologies or other native cross-platform frameworks. Many cross-platform frameworks create an abstraction on top of system-provided UI elements. This approach adds significant overhead to the rendering phase. Since Flutter uses its own set of UI elements, it bypasses communicating with the system about rendering. Flutter widgets are

13. Source: `https://docs.flutter.dev/resources/architectural-overview`.
14. FFI (Foreign Function Interface) is a mechanism by which a program written in one language can communicate with a program written in another language.

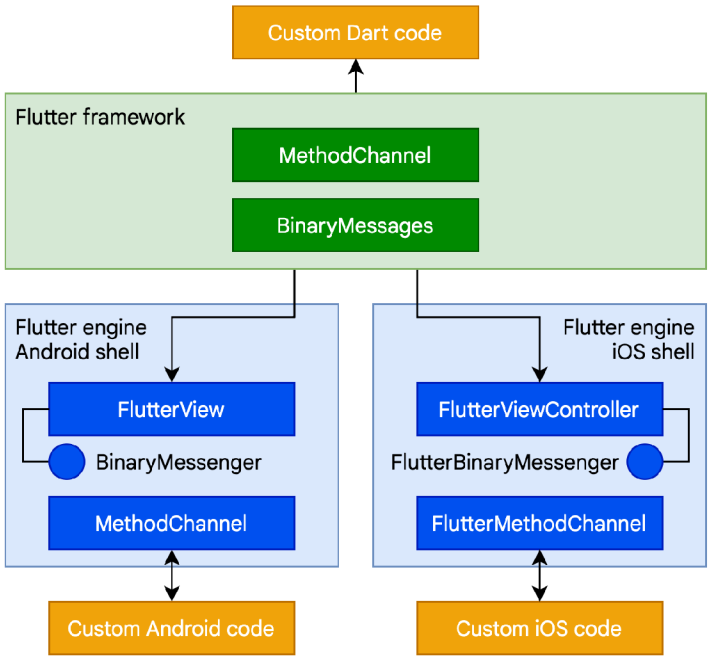

Figure 3.5: Flutter platform channels[13].

compiled into native code that uses Skia graphical engine for rendering, which calls CPU and GPU to draw the content to the device screen. Skia is an open-source 2D graphics engine developed by Google. It is written in C and C++ and powers rendering in Google Chrome, Chrome OS, Android, Mozilla Firefox, and other products [26][25]. The Skia engine focuses on performance and allows Flutter to achieve performance that is on par with native applications.

3.2.3 State Management

As mentioned in Section 3.2.1, Flutter applications have uni-directional data flow. Child widgets accept context from their parent. This context can not be mutated from the child. For data that changes inside the widget, developers use state. In Flutter, each user-defined widget extends either StatelessWidget class or StatefulWidget class[15]. The clear separation by inheritance makes it easier to reason about code, and it allows developers to see at first glance whether a widget has some internal state or not (see Figure 3.6).

For simple application state management, Flutter documentation recommends the usage of setState API or ChangeNotifier and Consumer widgets in combination with the Provider package. However, for larger applications, there is a variety of different solutions and libraries. The Flutter community is fragmented between these solutions, and there is no single best library for state management. The most known state management approaches include the BLoC pattern[16], Redux, GetX, and Riverpod community libraries.

15. StatelessWidget and StatefulWidget are base classes provided by the Flutter framework.
16. BLoC (Business Logic Components) is a design pattern that separates UI components from logic.

```dart
class TapboxA extends StatefulWidget {
  const TapboxA({Key? key}) : super(key: key);

  @override
  _TapboxAState createState() => _TapboxAState();
}

class _TapboxAState extends State<TapboxA> {
  bool _active = false;

  void _handleTap() {
    setState(() {
      _active = !_active;
    });
  }

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: _handleTap,
      child: Container(
        child: Center(
          child: Text(
            _active ? 'Active' : 'Inactive',
            style: const TextStyle(fontSize: 32.0, color: Colors.white),
          ),
        ),
        width: 200.0,
        height: 200.0,
        decoration: BoxDecoration(
          color: _active ? Colors.lightGreen[700] : Colors.grey[600],
        ),
      ),
    );
  }
}
```

Figure 3.6: Example of Flutter widget with local state[17].

3.2.4 Community

Flutter, similarly to React Native, heavily relies on the community. Although Flutter accepts bug fixes and improvements to the core from the community, more internal functionality is maintained by the core team than in React Native's case. For example, Flutter developers maintain official plugins for the camera and image picker, whereas React Native leaves it to the community.

The community around Flutter is growing at a fast pace. At the time of writing, Flutter GitHub repository has more than 132 000 stars and 941 contributors[18]. It has a thriving Reddit community with close to 82 000 followers[19] and a StackOverflow community with over 107 000 questions asked[20].

17. Source: `https://docs.flutter.dev/development/ui/interactive#creating-a-stateful-widget`.
18. `https://github.com/flutter/flutter`.
19. `https://www.reddit.com/r/FlutterDev/`.
20. `https://stackoverflow.com/questions/tagged/flutter?tab=Newest`.

3.3 Progressive Web Applications

Progressive web applications are a concept, not a specific technology. There are multiple frameworks for the PWA implementation part, for example, React[21], Vue[22], Svelte[23] or Angular[24]. However, I decided to use Flutter, which supports the web as one of its primary build targets, in order to see if it would be possible to use one codebase for both native and web applications.

3.4 Summary

This section briefly summarizes the similarities and differences of the presented frameworks. The most notable difference is that the JavaScript part of React Native communicates with the native part through an asynchronous message bridge to render native UI components. On the other hand, Flutter uses the Skia graphics engine to paint custom UI elements. The table below provides a high-level overview of essential aspects of React Native and Flutter:

Table 3.1: Summary of similarities and differences of used technologies.

	React Native	Flutter
Main Developer	Meta (formerly Facebook)	Google
Programming Language	JavaScript/TypeScript	Dart
Rendering	Platform specific rendering	Skia graphics engine
Component Communication	Parent-child only	Parent-child only
Ability to use System APIs	Yes, Native Modules[25]	Yes, Platform Channels[26]

21. https://reactjs.org/
22. https://vuejs.org/
23. https://svelte.dev/
24. https://angular.io/
26. https://reactnative.dev/docs/native-modules-intro
26. https://docs.flutter.dev/development/platform-integration/platform-channels

4 Example Application *FavoriteRoutes*

The chapter describes the design and implementation details of an example application used for the evaluation. In the first section, I identify functional and non-functional system requirements for the system and create a use-case diagram. Then I describe the design of the application, focusing on the application architecture and the user interface of the client application. The last section focuses on the implementation, specific technologies, file structure, and the differences between individual implementations.

4.1 Requirement Analysis

The initial requirements on the application included communication with a cloud backend and use of device hardware functionality. After considering the general assignment and discussions with the thesis advisor, I decided that a suitable example application would be a location and route tracking application, with similar tracking options to Strava[1] or Calimoto[2]. The working name for the application is FavoriteRoutes. The main use cases of FavoriteRoutes are presented in Figure 4.1.

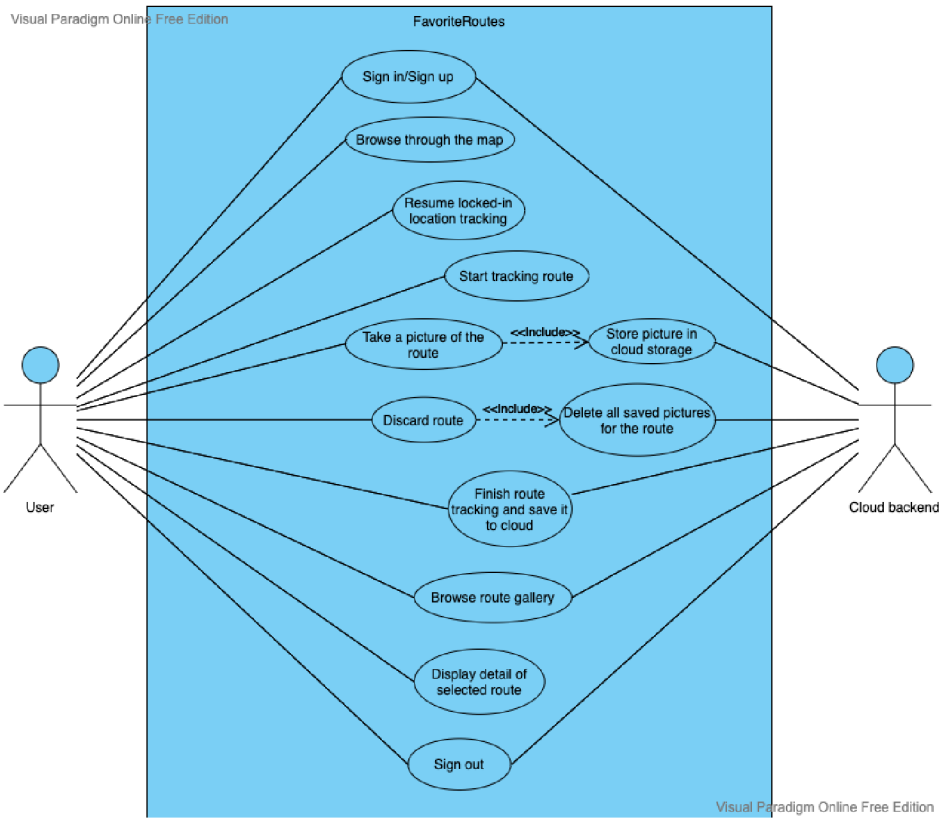

Figure 4.1: The use case diagram presenting the main functionality of FavoriteRoutes application.

The use-case diagram models the expected functionality of the example application. The functionality was further specified by analyzing the functional and non-functional requirements and incorporating results of follow-up discussions with the thesis advisor.

1. `https://www.strava.com/mobile`
2. `https://calimoto.com/en`

4.1.1 Functional Requirements

Functional requirements describe how the system should react to specific inputs and situations. In some cases, functional system requirements can also specifically define how the system should not behave [29].

From the thesis assignment, initial analysis, and the requirements to use device functionality, I came up with these functional requirements:

- The application provides email and password authentication.
- If authentication fails, the application shows a sensible error message.
- The application shows a current location on a map for authenticated users, and the location is updated as they move around.
- The application allows authenticated users to start/stop tracking a route at any moment.
- While users track a route, they can take an unlimited amount of pictures that will be stored together with the route.
- At any moment, users can discard or save and stop tracking the currently tracked route along with all already taken pictures.
- Authenticated users can access an in-application gallery with all their previously saved routes.
- While in the route gallery, users have the ability to navigate to a detail page of each route where they can see an interactive map and images.
- If the application goes offline when the user is already authenticated, it continues to function without severe limitations, and the data is synchronized when the internet connection is restored.

4.1.2 Non-functional Requirements

Non-functional requirements describe requirements that are not directly related to specific system functions provided to end-users. They specify criteria that can be used to reason about the operation of a system rather than its concrete behavior. For instance, non-functional system requirements can describe system reliability, system reaction time, or structure of APIs [29].

The example application has the following non-functional system requirements:

- There are three versions of the application, one written using the React Native framework, one implemented through the Flutter framework, and one implemented as a PWA.
- Supported platforms are iOS and Android for the native cross-platform implementations and the web for PWA.
- All implementations of the application look and behave the same, except for situations where selected technology does not support some behavior.
- All implementations of the application are connected to the same cloud backend and provide real-time data synchronization.
- The communication with the backend is performed through client-installed SDKs.
- The implementations follow best practices and principles for the selected technology.
- When possible and not in contradiction to the best practices for the selected technology, all implementations follow the same architectural decisions in order to provide as similar conditions for comparison as possible.

4.2 Design

This section first describes the architecture of the system, next it focuses on the user interface of the client applications.

4.2.1 Architecture

The architecture of the implemented system can be split into two parts – the cloud backend architecture and the client application architecture.

4.2.1.1 Cloud Backend

Considering the requirements for the system backend, such as real-time data synchronization, storage for both data and images objects, and running in the cloud, I decided to integrate with Firebase cloud backend service.

Firebase can be used for application releases, monitoring, performance tracking, and analytics. Developers can use Firebase to host their web application, use machine learning, or implement and host serverless cloud functions – small pieces of backend code that can be triggered directly by an HTTP request or through some events such as time period passing. The backend services are encapsulated in modular installable SDKs providing methods for API calls and can be installed in the client applications.

I also considered other alternatives, such as open-source projects Supabase[3] or Appwrite[4] but the main reasons for choosing Firebase for FavoriteRoutes were threefold: the ease of setup, the ability for automated offline synchronization, and previous experience of the thesis author. Since FavoriteRoutes is a prototype meant for technology comparison, not the production-ready application, I could use Firebase's free tier pricing with no additional costs. The used services from Firebase are:

- Firebase Authentication – a user management and authentication service.
- Firebase Firestore – a real-time document database.
- Firebase Cloud Storage – a storage for large files.

4.2.1.2 Client Application

The client application uses a subset of SDKs for mobile clients provided by the Firebase service. It is responsible for authenticating the user in cooperation with the backend service, keeping the user signed in for further sessions, storing and uploading data about the currently tracked route, and displaying data about previously saved routes. Specific technologies that were used are described in the implementation section (see 4.3).

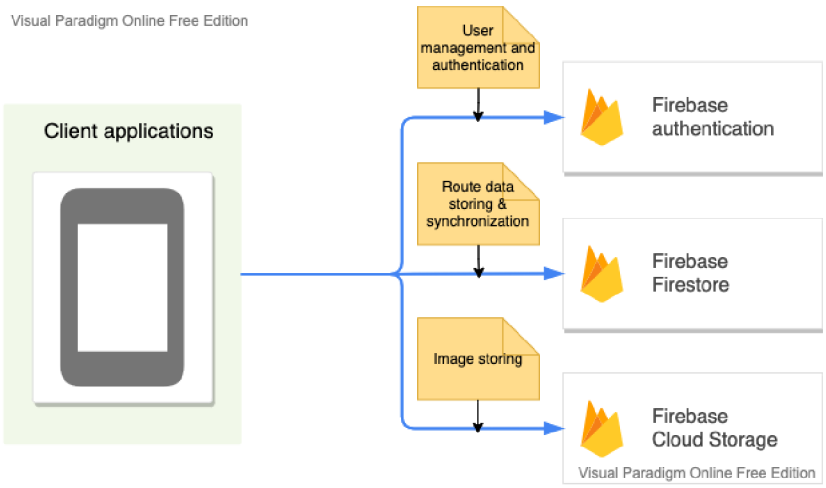

Figure 4.2: High-level overview of the application architecture.

3. `https://supabase.com`
4. `https://appwrite.io`

4.2.2 User Interface

The application is structured into four main user interface pages as shown in Figure 4.3 (screenshots of all possible UI states are presented in Appendix C). Upon initial application startup, the Sign In page is presented to the user. The application persists user data. Consequently, the next time the application is opened, the sign-in screen is skipped. After successful sign-in, the user is redirected to the application's main page with a map that tracks the user's location. The third screen contains a gallery of saved routes for the signed-in user, and it is accessed from the map screen. The last page displays the detail of the screen selected in the gallery.

Figure 4.3: FavoriteRoutes screens from iOS emulator. From left to right: Login, Map, Gallery, Route Detail with sample images.

4.2.2.1 Sign In

The Sign In screen is a plain page with only a small number of UI elements (Figure 4.3, leftmost). The topmost visual component of this page is a welcome text. Below, there are two text inputs for user email address and password. The last two elements are the sign-in and sign-up buttons. An error message is displayed in case of wrong credentials entered as shown in Figure C.2.

Figure 4.4: Sign In page with an error message (cropped).

4.2.2.2 Map

The dominant visual element of this page is a map that fills the whole device screen (Figure 4.3, second from left). There are four floating action buttons: one in the bottom-left, three in the bottom-right corners.

The button on the left side takes the user to a gallery page. The topmost button on the right side starts (or stops if the route is currently being tracked) route tracking. The middle button provides the functionality to re-center the map. The last button on the right side is the sign-out button. When the route tracking is in progress, an additional button appears in the bottom-left corner above the gallery button (Figure 4.5). It allows showing a modal with a camera for taking pictures (Figure 4.6, left).

Figure 4.5: Buttons on the map screen during route tracking (cropped).

When the user presses the button to stop the route tracking, a system alert modal with options to save or discard the route is presented (Figure 4.6, right).

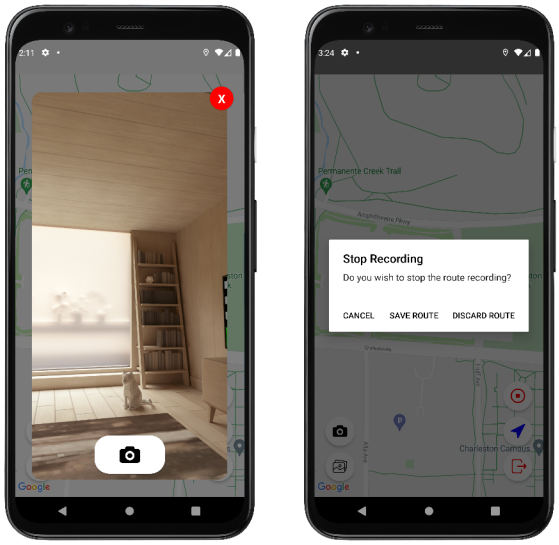

Figure 4.6: Left – camera modal over the main map page[5], right system alert shown when route tracking is stopped.

4.2.2.3 Routes Gallery

Pressing the gallery button opens the gallery page (Figure 4.3, second from right). There is a list of UI elements representing their saved routes. Each route is visualized as a small non-interactive map[6] (Figure 4.7). The date and time of the route are shown in the upper-left corner of the map. The last UI component of the route representation is a button placed on the right side of the map. Its purpose is to navigate to the route detail page.

The last component of the gallery screen is a floating back button which is situated in the left bottom corner of the page.

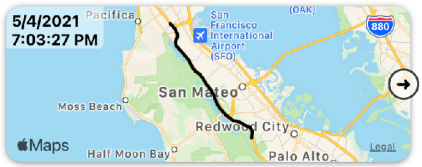

Figure 4.7: Recorded route element in the gallery screen (cropped).

5. The screenshots are taken from Android and iOS simulators that do not show actual camera screens but only a mocked one for android and black screen for iOS.
6. The decision to make maps in the gallery non-interactive was made in order to avoid issues with colliding gestures when scrolling and to provide a superior user experience.

4.2.2.4 Route Detail

The screen is divided vertically into two sections. The upper section shows an interactive map of the route with a re-center floating button placed in the bottom right corner. The lower section shows either a list of route images in a two-column layout or a text informing the user that there are no photos associated with the route (Figure 4.3, rightmost). The associated images start to load in the background immediately when the user enters the route detail page.

4.3 Implementation

Besides React Native and Flutter, I used several other tools, libraries, and frameworks that are presented further. Notably, the Firebase platform, the Redux state management framework, streams and redux-epics libraries, the Logz.io platform, and the libraries that provide map components and enable React Native and Flutter to render SVG elements.

4.3.1 Used Tools and Libraries

4.3.1.1 Firebase

As mentioned in section 4.2.1.1, I used the Firebase Authentication, Firestore, and Cloud Storage SDKs. The Authentication SDK was used for user management and authentication. The Firestore SDK was used to store route data and references to the locations of stored images. It provides real-time synchronization of the data, and it can be configured to work offline and synchronize data when the internet connection is restored. The Cloud Storage SDK was used to store images taken during the route tracking.

Both React Native and Flutter have community implementations of the Firebase API[7], distributed as client SDKs. These implementations do not cover the whole Firebase API yet but are reasonably full-featured.

4.3.1.2 Redux

Nowadays, mobile applications are mainly viewed as event-driven systems where events are triggered by users' actions. Redux library[8], which conforms with the event-driven application architecture methodology, is used to handle the state management of FavoriteRoutes. The core principle of Redux is that the entire shared application state that needs to be accessed from multiple places is kept in a centralized *Redux Store*. Each component (or widget in Flutter) can hook to the store and access some part of the stored state. This approach mitigates the need for extensive props drilling, and it helps to keep the application state management more maintainable.

The two main parts of Redux are actions and reducers. Actions are events that flow through the application, and they are the only way to pass data to the Redux store. Actions are sent to the store using the store.dispatch() method. They are virtually just plain objects (classes in Dart) that must have a *type* property and optionally can carry additional data (Figure 4.8). Reducers are pure functions (i.e. functions returning always the same result for the same input without side effects) that usually consist of a single switch statement on the action type and, based on it, return a new state (Figure 4.9). The most important concept in reducers is that they never modify the existing state. If the state is to be modified in reaction to some action, the reducer always returns a new state object [10].

Redux allows developers to intercept actions before they reach the reducer function. This interception is done through *Redux middleware*. Each middleware is a function that can process action or perform some side effects (e.g., in the example application logging, see Figure 4.11) and then call the `next` function received in an argument to pass data to the next middleware (or the reducer if all middlewares already run).

In general, the use of Redux makes debugging and application maintenance easy. There are tools that allow for time travel between actions and show a diff output between different actions. The

7. React Native implementation: `https://rnfirebase.io/`, Flutter implementation: `https://firebase.flutter.dev/`.
8. `https://redux.js.org`

separation of actions, reducers, selectors[9], and middleware makes it very easy for developers to test each of these components of the system independently.

```
export const SCREEN_CHANGED = "SCREEN_CHANGED"
export const createActionScreenChanged = (
    newScreen: ScreenEnum,
    routeId?: Maybe<string>
) => ({
    type: SCREEN_CHANGED,
    payload: {
        newScreen: newScreen,
        routeId: routeId,
    },
})
```

```
class ScreenChangedAction {
  final ScreenEnum newScreen;
  final String? routeId;

  const ScreenChangedAction({required this.newScreen, this.routeId});

  Map<String, dynamic> toJson() {
    return {
      'type': 'ScreenChangedAction',
      'newScreen': newScreen.string,
      'routeId': routeId,
    };
  }
}
```

Figure 4.8: Example of Redux action (left React Native, right Flutter).

```
export function locationReducer(
    state: LocationState = createDefaultLocationStateObject(),
    action: StateAction = { type: null }
) {
    switch (action.type) {
        case INITIAL_LOCATION_RESOLVED:
            return {
                ...state,
                initialLocation: action.payload.position,
            }

        case CURRENT_LOCATION_CHANGED:
            return {
                ...state,
                currentLocation: action.payload.newPosition,
            }

        case LOCATION_START_RECORDING:
            return {
                ...state,
                isRecordingRoute: true,
            }

        case LOCATION_STOP_RECORDING:
        case ON_DISCARD_ROUTE_CLICKED:
            return {
                ...state,
                isRecordingRoute: false,
            }

        default:
            return state
    }
}
```

```
LocationState locationReducer(
    LocationState state,
    dynamic action
) {
  if (action is InitialLocationResolvedAction) {
    return state.copyWith(
        initialLocation: action.position,
        currentLocation: action.position
    );
  }

  else if (action is CurrentLocationChangedAction) {
    return state.copyWith(currentLocation: action.position);
  }

  else if (action is LocationStartRecordingAction) {
    return state.copyWith(isRecordingRoute: true);
  }

  else if (
    action is LocationStopRecordingAction ||
    action is OnDiscardRouteClickedAction
  ) {
    return state.copyWith(isRecordingRoute: false);
  }

  return state;
}
```

Figure 4.9: Example of Redux reducer (left React Native, right Flutter).

9. Functions which take the Redux state as an argument, optionally run some transformations, and return a sub-state used by components.

4.3.1.3 Streams and Redux-Epics

Streams in computer science refer to possibly infinite sequences of data elements that become available over some time period [27]. The whole concept of the reactive programming paradigm, widely used for front-end technologies, is based on data or event streams and the propagation of changes [8].

Most languages used in frontend development have some implementation of streams and usually also implement the observer pattern via the *Reactive Extensions* (*ReactiveX*). "The observer pattern is a software design pattern in which an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods." [31] In JavaScript, the implementation is provided by the RxJS library[10] and in Dart by the RxDart library[11].

In the client application, the *Redux-epics* library provides an asynchronous action processing Redux middleware. An epic is a function that accepts a stream of actions as its input and, in most cases, returns a stream of actions as its output. The actions produced by epics are dispatched using the regular `store.dispatch()` method. The core idea behind epics is that they are processed after reducers process the action. This reduces the risk of race conditions because developers can rely on the fact the state has been already updated when they write their asynchronous epics. Epics are usually composed of ReactiveX values and operators and allow to separate a big amount of logic from UI components to the client application backend layer (Figure 4.10). In the example application, all of the logic related to Firebase and location tracking is performed inside epics.

```
const deleteImagesEpic = (action$: ActionsObservable<StateAction>) =>
    action$.pipe(
        ofType(DELETE_IMAGES_FOR_ROUTE_REQUESTED),
        mergeMap( project: (action : A ) => {
            const pathToDelete = action.payload.pathToDelete
            const images: RouteImage[] = action.payload.images

            if (images.length === 0) {
                return [createActionImagesForRouteDeleted( result: [])]
            }

            return from(images).pipe(
                mergeMap( project: (image : RouteImage ) => {
                    const fullPath = pathToDelete + image.id + ".jpg"
                    const ref = storage().ref(fullPath)
                    return defer( observableFactory: () => ref.delete()).pipe(
                        switchMap( project: () => {
                            return of( args: {
                                success: true,
                                path: fullPath,
                                imageId: image.id,
                            })
                        }),
                        catchError( selector: (err) => {
                            return of( args: {
                                success: false,
                                path: fullPath,
                                error: err,
                                imageId: image.id,
                            })
                        })
                    )
                }),
                toArray(),
                switchMap( project: (res: ImageDeleteResult[]) => {
                    return [createActionImagesForRouteDeleted(res)]
                })
            )
        })
    )
```

```
Epic<AppState> _startLocationTrackingEpic() =>
    (Stream<dynamic> action$, EpicStore<AppState> store) =>
        action$.whereType<InitialLocationResolvedAction>().switchMap((_) {
          Location location = Location();
          location.onLocationChanged.listen((LocationData event) {
            Globals.dispatch(CurrentLocationChangedAction(position: event));
          });
          return Stream.value(const LocationTrackingStartedAction());
        })
        .handleError((err) {
          logger.error('Error in _startLocationTrackingEpic', {
            'err': err,
          });
          return Stream.value(GetLocationFailedAction(error: err));
        });
```

Figure 4.10: Example of epic in the React Native (left) and Flutter (right) implementations.

10. `https://github.com/ReactiveX/rxjs`.
11. `https://github.com/ReactiveX/rxdart`.

4.3.1.4 Logz.io

Logz.io is an end-to-end cloud monitoring platform. It provides many features, such as alerting, metrics, and tracing, but the sample application uses only its ELK functionality. ELK is an acronym for three open-source projects – Elasticsearch, Logstash, and Kibana. Elasticsearch is a search and analytics engine. Logstash is a server-side data processing platform that ingests data, transforms it, and finally sends it to a stash service (such as Elasticsearch). The final component of ELK is Kibana, which is a frontend service that lets users visualize their data from Elasticsearch [3].

The reasons to integrate Logz.io were its ability to help investigate issues with the application when it is running in production builds on physical devices, and my previous experience with the tool. Both React Native and Flutter implementations use a simple Redux middleware, which sends logs for each Redux action emitted in the system (Figure 4.11). The logger plugin also allows sending logs from any other critical place in code outside the Redux actions. It buffers all log records using RxJS/RxDart respectively and then sends the buffered batch to the cloud every second (Figure 4.12).

```
createReduxMiddleware(): Middleware {
    // The proper type for 'api is MiddlewareAPI<D, S> where D extends Dispatch and S is any
    // since it is not used in the middleware I decided to omit the type in favor of readability
    return (api: any) => (next: Dispatch<AnyAction>) => (action: any) => {
        if (!this.configured) {
            return next(action)
        }
        const entry = this.createEmptyLog()
        entry.level = "info"
        entry.message = action.type
        entry.redux_action = JSON.stringify(action)
        entry.log_type = "redux-action"

        this.log$.next(entry)
        return next(action)
    }
}
```

```
static dynamic loggerMiddleware(
    Store<AppState> store, action, NextDispatcher next) {
  if (!configured) {
    return next(action);
  }

  Map<String, String> entry = createEmptyLog();
  entry['level'] = 'info';
  entry['message'] = (action).toString();
  entry['redux_action'] = jsonEncoder.convert(action);
  entry['log_type'] = 'redux-action';

  log$.add(entry);
  return next(action);
}
```

Figure 4.11: Redux logger middleware (left React Native, right Flutter).

```
configure(options: LogzIoConfig) {
    const url = options.url
    const token = options.token
    const fullUrl = url + "?token=" + token

    this.log$
        .pipe(
            bufferTime( bufferTimeSpan: 1000),
            concatMap( project: (batch: any) => sendHttpRequest(fullUrl, batch).pipe(retry( count: 1))),
            catchError( selector: () => EMPTY)
        )
        .subscribe( next: () => {})

    this.configured = true
}
```

```
static void configure(LogzIoConfig options) {
  String fullUrl = options.url + '?token=' + options.token;

  log$
      .bufferTime(const Duration(milliseconds: 1000))
      .flatMap((batch) => sendHttpRequest(fullUrl, batch))
      .handleError((err) {
    return const Stream.empty();
  }).listen((event) {});

  configured = true;
}
```

Figure 4.12: Logger configuration, which shows the usage of RxJS and RxDart to buffer logs and send them to the cloud (left React Native, right Flutter).

4.3.1.5 Maps

Neither React Native nor Flutter provides a build-in component for maps. However, there are solutions maintained by the community.

The *react-native-maps*[12] is the mainstream library that exposes a `MapView` component and additional functionality in React Native. It supports Google Maps on both Android and iOS and Apple Maps on iOS only. The library provides many `MapView` properties, such as gesture callbacks, a lite mode for Google Maps on Android phones, or the ability to add polylines as children to the MapView. However, some of the properties are platform-specific or map-specific.

12. `https://github.com/react-native-maps/react-native-maps`.

The situation with Flutter map widgets is more complicated. Many libraries show meager download numbers, and there does not seem to be one best solution. Furthermore, there are not many options for cross-platform usage. In the end, I decided to use the *platform_maps_flutter* package[13] which is not as feature-complete as the library used for React Native, but it seemed like the best cross-platform option available. It also provides an implementation of Google Maps and Apple Maps, but in the case of this package, Google Maps are only available for Android. I experienced a couple of issues with this package, mainly with missing or unexposed properties for Android and iOS, respectively (further discussed in 5.6.1).

4.3.1.6 SVG

Most of the buttons in all the implementation variants of the example applications include icons. These icons are SVG elements. Neither React Native nor Flutter has build-in support for SVG rendering. However, there are community solutions available.

React Native community maintains a widely used SVG library called `react-native-svg`[14]. It provides a set of SVG components (which correspond to the XML-based SVG syntax) that can be used inside .jsx files.

Flutter has the package `flutter_svg`[15]. It works in all the target platforms and supports rendering from both SVG files and raw strings. The only issue is only partial API documentation.

4.3.2 Database

The database for the example application is provided by the integration with Firebase Firestore and Cloud Storage services. All route data are stored in the document-based Firestore database, except for images that are uploaded to the Cloud Storage database. Since Firestore is a document database, and data are stored as collections of JSON documents, there is no database schema. Instead, I will describe the hierarchy of the collections and the structure of the JSON documents our implementations use.

The top-level collection is called *users*. Documents inside *users* collection are identified by users' IDs. Each user document holds a reference to a collection of saved routes called *routes*. The documents in this collection store individual routes identified by route IDs. The structure of a *route* document is:

- `id` – a unique identifier of the route,
- `images` – an array of JSON objects representing the images stored with the route,
- `routeCoordinates` – an array of JSON objects consisting of *latitude* and *longitute*,
- `timestamp` – a timestamp representing the time when the route was saved.

The structure of an object representing a route image is:

- `id` – a unique identifier of the image,
- `url` – an address pointing to the location in Firebase Cloud Storage from where the image is downloadable,
- `coordinates` – JSON object consisting of *latitude* and *longitude* representing the coordinate where the picture was taken.

Figure 4.13 shows an example of communication with the Firestore document database.

4.3.3 File Structure

With Redux as the project's core, it is straightforward to separate most logic from the user interface and follow a clear file structure. For example, the logic that concerns data exists in a `data` folder, logic around location tracking is placed in the `location` folder, and logic that handles navigation resides in the `navigation` folder. These folders encapsulate only business logic and usually contain no UI components. The file structure for such folder consists of the following files:

13. `https://github.com/LuisThein/platform_maps_flutter`.
14. `https://github.com/react-native-svg/react-native-svg`.
15. `https://github.com/dnfield/flutter_svg`.

```
// previous code omitted

return defer( observableFactory: () =>
    firestore() Module
        .collection(USER_COLLECTION) CollectionReference<DocumentData>
        .doc(currentUser.uid) DocumentReference<DocumentData>
        .collection(ROUTE_COLLECTION) CollectionReference<DocumentData>
        .add({
            id: getRouteId(state$.value),
            routeCoordinates: JSON.stringify(locationData.routeCoordinates),
            timestamp: new Date().toISOString(),
            images: JSON.stringify(getPreparedImages(state$.value)),
        })
).pipe(
    switchMap( project: () => {
        return [createActionLocationRecordingSaved()]
    }),

    // following code omitted
```

```
// previous code omitted

return FirebaseFirestore.instance
    .collection(USER_COLLECTION)
    .doc(currentUser.uid)
    .collection(ROUTE_COLLECTION)
    .add({
      'id': getRouteId(store.state),
      'routeCoordinates':
          jsonEncode(action.data.map((rec) => rec.encoded).toList()),
      'timestamp': DateTime.now().toUtc().toIso8601String(),
      'images': jsonEncode(getPreparedImages(store.state)
          .map((rec) => rec.encoded)
          .toList()),
    })
    .asStream()
    .switchMap((_) {
      return Stream.value(const LocationRecordingSavedAction());
    })

// following code omitted
```

Figure 4.13: Firestore database communication example (left React Native, right Flutter).

- `actions.ts/actions.dart` – these files contain all Redux actions that flow through the system and functions that create these actions (see 4.3.1.2).
- `epics.ts/epics.dart` – are files that contain all the *epics* functions (see 4.3.1.3).
- `reducer.ts/reducer.dart` – contain the Redux state reducers (see 4.3.1.2).
- `selectors.ts/selectors.dart` – selector files contain functions called *selectors* that take application state as an input, optionally perform some transformations, and return data that can be used in UI components or epics.
- `state.dart` – file with class defining the structure of appropriate sub-state (e.g., data or location sub-state). State files are only present in Flutter implementation because it is more boilerplate to describe the state in an object-oriented language. In React Native implementation, the type for sub-state is defined in the corresponding reducer file.

The full file structure of the client application consists of previously described folders containing a logic related to the Redux library, a `components` folder which consists of reusable UI components, logger folder which encapsulates the implementation of logging, `screens` folder that contains UI components and utilities for specific screens, a `store` folder that includes files necessary to set up Redux store and middleware, and framework and platform-specific files (see Appendix D for visual representation).

4.3.4 Implementation Differences

The three implementations are as similar as possible. The user interfaces look almost identical, except for some platform-specific features (e.g., buttons at the bottom edge of the screen on Android devices). However, there is one significant difference in the behavior of the React Native and Flutter implementations. The Flutter map widget implementations did not provide any method to distinguish between map location updates triggered by user action (e.g., scrolling) and location updates triggered programmatically. The result is that the map continuously animates to the current user's location, and it virtually disallows to scroll away. This issue does not affect the iOS platform because Apple maps provide a different option for user location tracking. Additionally, there is a slight difference in the zoom of individual maps in the route gallery. The Flutter map widget uses a different method to describe the location on the map than React Native, and I was not able to make the zoom identical.

5 Technologies Comparison

Since the primary goal of this thesis was to compare the used technologies, I performed a set of experiments focused on chosen comparison metrics using the implemented applications. The evaluation is accompanied by the subjective assessment of personal experience with the used technologies.

The first section of this chapter describes the evaluation methodology. Next two sections present time behavior and resource utilization characteristics. The following section focuses on other metrics such as installed application size, code sharing, the codebase size, the compilation time, and build specifics. The last section summarizes my personal experience with the focus on the perceived ease of writing and the advantages and disadvantages of React Native and Flutter respectively.

5.1 Methodology

I compared a subset of performance efficiency characteristics described by ISO/IEC 25010 [24], namely time behavior and resource utilisation.

The evaluated time behavior characteristics include: *a*) the application startup time (page load time in the case of PWA), and *b*) the render time of each screen.

The evaluated performance and resource utilization metrics include: *a*) CPU usage, *b*) memory usage, *c*) and rendering performance in frames per second (FPS).

The measurements were performed using *release* configurations of the applications in order to mimic the production-ready optimized applications. I took ten measurements for each characteristic. Next, I computed the basic statistical attributes (mean average, median, population variance, and standard deviation) and performed the significance tests to identify the differences using Microsoft Excel. Namely:

- *unpaired two-sample T-Test* when comparing two sets of measurements (e.g., only React Native and native Flutter);
- *the analysis of variance* (*one-way ANOVA*) followed by a *post-hoc Tukey's Honest Significance Test* (*HSD*) when comparing more than two data sets.

The significance level was $\alpha = 0.05$ for all the tests. The results were rounded to one significant figure.

Tables with full measured results are available in Appendix F and all statistical analysis calculations are attached in the thesis archive in IS MU.

5.1.1 Apparatus

The measurements were performed using Xcode 13.1 and Android Studio Arctic Fox | 2020.3.1 Patch 3 against the 0.66.3 version of React Native and version 2.5.1 of Flutter. Additionally, for the React Native implementation, I used a version with the Hermes JavaScript engine enabled.

As testing devices, I used an iPhone SE (2^{nd} generation, running iOS 14.7.1), an older low-end Android device Xiaomi Redmi 7 (running Android 10 QKQ1.191008), and a 13-inch MacBook Pro 2020 (2 GHz Quad-Code Intel Core i5, 16 GB LPDDR4X memory, and Intel Iris Plus Graphics 1536 MB).

5.2 Time Behavior Characteristics

5.2.1 Application Startup Time

5.2.1.1 Procedure

To measure application startup time, I used the Firebase Performance module. Without any configuration, it automatically tracks metrics such as application startup time (first contentful paint in case of web applications) or HTTP requests duration. I measured the startup time on a fresh application build after deleting all previous application data for each of the ten samples.

For the web implementation, I compared the time of the *first contentful paint* metric that determines when the browser rendered the first bit of content from the DOM[1] and showed first visual feedback to the user [6].

I decided to track the time of startups with cleared cache on both iOS and Android devices, as well as the cached startups, which I consider a more valuable metric for PWAs.

5.2.1.2 Results

Table 5.1: Summary of startup times (FCP in case of PWA). Reported values are mean averages with standard deviation.

	React Native	Flutter	PWA Clean	PWA Cached
Android	224.4±36.2ms	233.6±51.6ms	3911±103ms	1168±130ms
iOS	192.8±39.9ms	198.3±38.5ms	2252m±55ms	572±40ms

Table 5.1 summarizes the results. The ANOVA showed that the effect of **used technology** on **application startup time** between at least two measured data sets was significant for both Android and iOS:

- Android ($F_{3,36} = 3511.11$, $p < 0.05$)
- iOS ($F_{3,36} = 4557.62$, $p < 0.05$)

A post-hoc Tukey's HSD test indicated that on both platforms, React Native and Flutter had significantly faster startup times than both clean and cached PWA. Additionally, cached PWA startup was substantially faster than clean PWA startup. There was no statistically significant difference between React Native and Flutter implementation.

From the user perspective, there was no visible delay in the startup time of native applications, but the startup time of the PWA was noticeably slower, although closer to native implementations in the case of cached PWA on the iOS testing device.

5.2.2 Screen Render Times

5.2.2.1 Procedure

In order to measure render times of each screen, I tracked the time from when the code to generate the measured component (or widget in case of Flutter) was called until the rendering ended. In the case of React Native, where only functional components were used (functions that return components in JSX form [15]), the time when the rendering starts is the beginning of the function body. The time when rendering ends is determined by the call to the first useEffect React hook[2] which corresponds to a class component's `componentDidMount` lifecycle event. In Flutter, the logic was very similar. The time tracking started in the widget's constructor, and the result was reported when the widget's first run of the *build* method ended. Once these metrics were gathered, they were sent to Logz.Io, where I could see the results. I decided to choose this method because I wanted to see the results from release builds. All the performance tracking tools available in used frameworks are only applicable to debug and profile builds.

5.2.2.2 Results

The summary of results can be seen in tables 5.2 (Android) and 5.3 (iOS). The findings are presented below.

Android Results: The ANOVA showed that the effect of **used technology** on **screen render time** between at least two measured groups in all screens was significant:

1. DOM (Document Object Model) represents web page as a tree structure where each node is an object representing part of the document.
2. See `https://reactjs.org/docs/hooks-intro.html`.

Table 5.2: Summary of screen render times on Android. Reported values are mean averages with standard deviation.

Screen	React Native	Flutter	PWA
Login	14.1±2.7ms	10.9±2.0ms	26.2±3.5ms
Map	21.4±2.3ms	28.3±2.9ms	58.9±6.0ms
Gallery	43.1±9.2ms	58.5±6.5ms	1108.3±94.0ms
Route Detail	14.5±1.9ms	20.9±1.7ms	99.3±10.6ms

Table 5.3: Summary of screen render times on iOS. Reported values are mean averages with standard deviation.

Screen	React Native	Flutter	PWA
Login	3.3±1.7ms	3.1±1.4ms	19.2±2.2ms
Map	8.9±3.0ms	10.7±2.4ms	31.2±2.3ms
Gallery	27.7±2.1ms	32.4±3.6ms	446.9±36.2ms
Route Detail	7.2±1.1ms	9.6±1.2ms	44.5±3.6ms

- Login Screen ($F_{2,27} = 74.07$, $p < 0.05$)
- Map Screen ($F_{2,27} = 212.82$, $p < 0.05$)
- Gallery Screen ($F_{2,27} = 1122.21$, $p < 0.05$)
- Route Detail Screen ($F_{2,27} = 506.98$, $p < 0.05$)

A post-hoc Tukey's HSD test showed that React Native and Flutter had both significantly faster render times than PWA for all the screens. While there were no significant differences between React Native and Flutter for *Login*, *Gallery* and *Route Detail Screens*, React Native was significantly faster in rendering the *Map Screen*.

iOS Results: Similar to Android, the ANOVA performed on the iOS measurements indicated that the effect of **used technology** on **screen render time** was significant in all screens between at least two of the measured groups:

- Login Screen ($F_{2,27} = 243.58$, $p < 0.05$)
- Map Screen ($F_{2,27} = 204.52$, $p < 0.05$)
- Gallery Screen ($F_{2,27} = 1177.43$, $p < 0.05$)
- Route Detail Screen ($F_{2,27} = 742.45$, $p < 0.05$)

The follow-up post-hoc Tukey's HSD test indicated that React Native and Flutter had both significantly faster render times than PWA for all the screens. There was no statistically significant difference between React Native and Flutter render times.

From the user perspective, there was a major lag during the Gallery screen rendering in PWA implementation on both Android and iOS. The delays in other screens' rendering were barely noticeable.

5.3 Resource Utilization Characteristics

5.3.1 Procedure

I prepared two different test scenarios to measure the rest of the resource utilization metrics. The measurements started ten seconds after navigating to the currently considered screen to give the application time to stabilize and lasted for ten seconds.

The first measurement focused on the resource utilization on each screen when the application was *idle*, and no interactions were performed during the test.

The second measurement focused on resource utilization while interacting with the application (i.e., *active* application usage). I performed the following pre-defined set of interactions on each screen:

- **Login Screen** – I focused the *email address* input and typed *crha.thesis@gmail.com*. After this step was finished, I manually deleted the text. This process was repeated until the measured time period passed.
- **Map Screen** – I kept scrolling from the top of the screen to the bottom to keep navigating the map in a single direction.
- **Gallery Screen** – I used an account that had thirty saved routes to scroll from the first route to the last one and vice versa repeatedly.
- **Route Detail Screen** – The test was performed on a route with six saved images. I performed the same steps on the map located in the top section of this screen as in the pre-defined scenario of the *Map Screen*.

The FPS measurements were performed only in the *active* usage test scenario since the *idle* FPS are not a relevant metric for general purpose applications.

Limitations: Measuring resource utilization metrics in the *release* configurations of applications is limited. The approach that was selected for CPU, memory, and FPS allowed to aggregate a mean average of the metrics over the measured time frame. As a result, the final statistical results are calculated from the mean averages of each test run.

5.3.2 CPU Usage

5.3.2.1 Tooling

There is a difference in how the results are reported on the iOS and Android platforms. The measured results for iOS show the CPU usage *across all cores*, which means the value can be greater than 100%. For Android, the results show the *total CPU usage of the device*, and the limit is 100%. This is caused by the differences in the iOS and Android tooling.

Xcode Debug Gauges was used to capture the CPU usage of the native application on iOS. *Android Profiler*, which is a part of Android Studio IDE, was used for the native Android implementation. Both tools allow attaching to a process running on a connected device and display a real-time overview of CPU and other metrics.

Measurements of the PWA implementation were done using Google Chrome *Remote Debugging* tool for the Android device and Safari *Web Inspector* for the iOS device. These tools allow a remote workstation to attach to a web page on Android and iOS respectively. In the case of the Google Chrome remote debugger, I used the *Performance* and *Performance monitor* tools, which allow us to record application performance metrics over a time period. The Web Inspector provides similar statistics for iOS devices using the *Timelines* tool.

Screenshots of the interface of used tools are available in Appendix E.

5.3.2.2 Results

Summarized results are available in tables 5.4 and 5.5.

Table 5.4: Summary of CPU usage on Android. Reported values are mean averages with standard deviation.

Screen	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
Login	1.2±0.4%	1.2±0.4%	1.4±0.5%	7.9±2.6%	9.1±2.2%	10.2±1.0%
Map	1.8±0.8%	1.5±0.8%	1.7±0.6%	15±2%	15.6±2.8%	37±1%
Gallery	3.1±1.8%	4.4±2.4%	6.5±1.1%	33.9±2.4%	25.3±2.8%	85.5±4.3%
Detail	1.6±0.8%	1.7±0.9%	5.1±1.6%	10.3±1.8%	10.3±1.1%	39.2±2.1%

Android Results: The ANOVA showed that in the idle test scenario, the effect of **used technology** on **CPU usage** was significant between at least two measured groups in the *Gallery* and *Route Detail Screens*:

Table 5.5: Summary of CPU usage on iOS. Reported values are mean averages with standard deviation.

Screen	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
Login	1.6±0.8%	1.5±0.7%	1.5±0.5%	14.9±4.4%	13.6±3.3%	28.5±1.1%
Map	1.9±0.8%	1.7±0.8%	1.4±0.5%	48.9±1.6%	46.9±3.7%	73.2±3.0%
Gallery	1.7±0.6%	2.7±1.9%	2.1±0.7%	163.3±32.4%	83.8±19.1%	109.9±7.6%
Detail	1.4±0.5%	1.7±1.0%	2.2±0.6%	51.6±1.9%	52.5±2.2%	84.8±1.7%

- Login Screen ($F_{2,27} = 0.64, p > 0.05$)
- Map Detail Screen ($F_{2,27} = 0.39, p > 0.05$)
- Gallery Screen ($F_{2,27} = 7.81, p < 0.05$)
- Route Detail Screen ($F_{2,27} = 27.21, p < 0.05$)

Tukey's HSD test showed that React Native had significantly lower idle CPU usage on the *Gallery Screen* than the PWA. Additionally, both React Native and Flutter had significantly lower CPU usage on the *Route Detail Screen*. Other than this, there were no significant differences in the idle CPU usage.

In the case of the active usage test scenario, the analysis of variance suggested that there was no significant difference between active CPU usage in the *Login Screen*. However, it suggested that there was a significant difference between at least two of the measured data sets in the *Map*, *Gallery*, and *Route Detail Screens*:

- Login Screen ($F_{2,27} = 2.85, p > 0.05$)
- Map Screen ($F_{2,27} = 302.03, p < 0.05$)
- Gallery Screen ($F_{2,27} = 879.34, p < 0.05$)
- Route Detail Screen ($F_{2,27} = 837.07, p < 0.05$)

Tukey's HSD test showed that React Native and Flutter had significantly lower CPU usage than the PWA in the *Map*, *Gallery*, and *Route Detail Screens*. Furthermore, Flutter had significantly lower CPU usage than React Native in the *Gallery Screen*.

iOS Results: The same analysis performed on iOS showed that for data measured during the idle CPU usage test, there was no significant difference in the effect of **used technology** on **CPU usage**:

- Login Screen ($F_{2,27} = 0.07, p > 0.05$)
- Map Screen ($F_{2,27} = 1.11, p > 0.05$)
- Gallery Screen ($F_{2,27} = 1.59, p > 0.05$)
- Route Detail Screen ($F_{2,27} = 2.74, p > 0.05$)

For the active usage test scenario, the ANOVA showed that the effect of **used technology** on **CPU usage** was significant between two or more measured groups in all the screens:

- Login Screen ($F_{2,27} = 52.23, p < 0.05$)
- Map Screen ($F_{2,27} = 228.41, p < 0.05$)
- Gallery Screen ($F_{2,27} = 30.07, p < 0.05$)
- Route Detail Screen ($F_{2,27} = 890.19, p < 0.05$)

Tukey's HSD test showed that React Native and Flutter used significantly fewer resources than PWA in the *Login*, *Map*, and *Route Detail Screens*. In the *Gallery Screen* Flutter used substantially fewer resources than React Native and PWA. Also, PWA used significantly fewer resources in the *Gallery Screen* than React Native.

5.3.3 Memory Usage

5.3.3.1 Tooling

To measure memory used by the example application, I took the same approach as when measuring CPU. All of the used tools – Xcode Debug Gauges, Android Profiler, Google Chrome Remote Debugging, and Safari Web Inspector report multiple metrics simultaneously. This allowed me to gather metrics for CPU and memory usage in the same test run by looking at different sections of the reports from the tools.

5.3.3.2 Results

The summarized results of application memory usage are available in tables 5.6–5.9.

Table 5.6: Summary of idle memory usage on Android. Reported values are mean averages with standard deviation.

Screen	React Native	Flutter	PWA
Login	56.05±0.64MB	68.51±0.55MB	17.38±0.44MB
Map	91.48±1.82%	143.12±3.04MB	21.55±0.44MB
Gallery	373.17±0.98%	342.54±0.71%	102.72±0.64MB
Detail	382.76±2.28MB	355.49±0.66MB	112.97±0.64MB

Table 5.7: Summary of active memory usage on Android. Reported values are mean averages with standard deviation.

Screen	React Native	Flutter	PWA
Login	63.73±1.32MB	77.98±1.03MB	25.26±0.52MB
Map	106.11±3.06%	158.39±2.69MB	26.3±0.4MB
Gallery	416.19±12.61MB	408.78±21.25MB	143.43±0.93MB
Detail	404.3±4.04%	377.99±2.36%	122.64±0.61MB

Table 5.8: Summary of idle memory usage on iOS. Reported values are mean averages with standard deviation.

Screen	React Native	Flutter	PWA
Login	29.02±0.36MB	50.21±0.48MB	22.71±0.33MB
Map	68.48±0.71%	116.6±0.88MB	24.08±0.36MB
Gallery	250.49±0.54%	240.51±0.62%	124.06±0.6MB
Detail	284.32±0.47MB	349.72±1.53MB	135.8±0.61MB

Table 5.9: Summary of active memory usage on iOS. Reported values are mean averages with standard deviation.

Screen	React Native	Flutter	PWA
Login	35.07±0.49MB	57.17±0.65MB	25.96±0.51MB
Map	80.78±0.8MB	132.6±1.65MB	30±0.67MB
Gallery	415.09.9±28.09MB	452.99±46.6MB	200.19±1.64MB
Detail	305.54±3.02MB	371.35±1.46MB	142.63±0.79MB

Android Results: The ANOVA suggested that in the idle test scenario, the effect of **used technology** on **memory usage** was significant between at least two compared groups in all screens:

- Login Screen ($F_{2,27} = 21253.64$, $p < 0.05$)
- Map Screen ($F_{2,27} = 7922.06$, $p < 0.05$)
- Gallery Screen ($F_{2,27} = 313636.8$, $p < 0.05$)

- Route Detail Screen ($F_{2,27} = 98939.78$, $p < 0.05$)

Tukey's HSD test showed that the PWA consumed significantly less memory in all screens than both native implementations. In the *Login* and *Map Screens*, React Native memory consumption was considerably lower in comparison to Flutter. However, in the *Gallery* and *Route Detail Screen*, Flutter consumer substantially less memory than React Native.

The same analysis performed on the active usage comparison showed significant differences between two or more compared groups on all screens as well:

- Login Screen ($F_{2,27} = 6523.77$, $p < 0.05$)
- Map Screen ($F_{2,27} = 7133.93$, $p < 0.05$)
- Gallery Screen ($F_{2,27} = 1066.41$, $p < 0.05$)
- Route Detail Screen ($F_{2,27} = 29380.84$, $p < 0.05$)

Tukey's HSD test results implied that PWA memory consumption was significantly lower than native applications memory consumption on all screens. In the *Login* and *Map Screens*, React Native implementation used less memory than the Flutter implementation. On the other hand, in the *Route Detail Screen*, Flutter memory consumption was significantly lower than the one of React Native. There was no significant difference in the memory consumption between React Native and Flutter in the *Gallery Screen*.

iOS Results: In the case of *idle* iOS test scenario, the analysis of variance showed a significant difference in the effect of **used technology** on **memory usage** in all screens:

- Login Screen ($F_{2,27} = 11885.6$, $p < 0.05$)
- Map Screen ($F_{2,27} = 41376.18$, $p < 0.05$)
- Gallery Screen ($F_{2,27} = 128740.94$, $p < 0.05$)
- Route Detail Screen ($F_{2,27} = 110911.06$, $p < 0.05$)

The results of follow-up post-hoc Tukey's HSD test showed that the PWA used significantly less memory in all screens than native implementations. In the *Login*, *Map*, and *Route Detail Screens*, React Native used substantially less memory than Flutter. Lastly, in the *Gallery Screen*, Flutter implementation used significantly less memory than React Native.

The one-way ANOVA analysis performed on the results of active scenario memory usage on iOS suggested that there is a statistically significant difference between at least two measured groups in all screens:

- Login Screen ($F_{2,27} = 7570.87$, $p < 0.05$)
- Map Screen ($F_{2,27} = 18574.36$, $p < 0.05$)
- Gallery Screen ($F_{2,27} = 169.34$, $p < 0.05$)
- Route Detail Screen ($F_{2,27} = 31532.78$, $p < 0.05$)

Tukey's HSD test showed that PWA memory consumption was significantly lower in all screens than the consumption of React Native and Flutter native applications. Additionally, React Native consumed significantly less memory than Flutter in all screens.

5.3.4 FPS (Frames per Second)

5.3.4.1 Tooling

To capture frame rendering times of iOS implementations, I used the *Instruments* application which is included in the Xcode IDE. This application provides a number of tools that can be used to debug specific metrics of the attached application. Specifically, I used the *Metal System Trace* tool, which allows a granular overview of rendered frames in the selected time range.

On the Android device, I used the *System Tracing* tool, which becomes available once the device is connected to a workstation and USB debugging is enabled. When toggled, it starts gathering data from the whole system, which are grouped by the application package name. This data, among else, contain frame rate information over a time period.

I did not find a reliable way to measure the release build FPS of the PWA implementation on the testing devices. There are tools that gather FPS metrics in debug builds, but results gathered that way are not comparable to optimized release builds of native applications. This is a limitation of the used evaluation methodology, and PWA was omitted from the FPS comparison.

5.3.4.2 Results

Since the display refresh rate of both Android and iOS testing devices is 60Hz, the effective cap and the best achievable result for FPS is sixty, even if the hardware manages to calculate more FPS [5]. The summarized results are available in Table 5.10.

Table 5.10: Summary of FPS comparison. Reported values are mean averages with standard deviation.

Screen	Android		iOS	
	React Native	Flutter	React Native	Flutter
Login	59.5±0.7	59.7±0.5	60±0	60±0
Map	52.4±1.4	54.1±1.3	60±0	60±0
Gallery	36.3±1.8	35.3±1.7	59.6±0.7	59.7±0.5
Route Detail	52.7±1.4	38.4±1.7	59.9±0.3	60±0

Android Results: T-Test showed significant differences in the effect of the **used framework** on the **FPS** in the *Map* and *Route Detail Screens*, where Flutter implementation had more FPS than React Native implementations in the *Map Screen* and React Native had significantly more FPS than Flutter in the *Route Detail Screen*:

- Login Screen (T_{18} = -0.74, $p > 0.05$)
- Map Screen (T_{18} = -2.64, $p < 0.05$)
- Gallery Screen (T_{18} = 1.22, $p > 0.05$)
- Route Detail Screen (T_{18} = 19.89, $p < 0.05$)

iOS Results: T-Test showed that the effect of **used framework** on the **FPS** was not significant in any screen:

- Login Screen – measured data was identical in all rounds of testing
- Map Screen – measured data was identical in all rounds of testing
- Gallery Screen (T_{18} = -0.37, $p > 0.05$)
- Route Detail Screen (T_{18} = -1, $p > 0.05$)

Note on PWA: Despite the inability to reliably measure the FPS of PWA, it is worth noting that the user-perceived frame rate felt lower than in the case of native implementations, especially on Android. In the *Gallery Screen* the scrolling was staggering, and the lagging made the application hard to use.

5.4 Other Application Metrics

5.4.1 Application Size

5.4.1.1 Procedure

The *application size* was measured immediately after a clean installation, before the application stored any data locally. On iOS, I used the system tool available in *Settings* → *General* → *iPhone Storage* to find

out the application size. On Android, the application size can be seen in *Settings* → *Apps* → *Manage Apps*. The Android system also provides information about the size of the installed PWA application. However, the iOS system does not provide this information.

5.4.1.2 Results

Table 5.11 overviews the application size for each platform and technology. Since the measured values did not differ between individual installations, it was not necessary to perform a further statistical evaluation.

PWA had by far the smallest size on the Android platform, taking only 319KB on the device's storage. Flutter implementation was behind PWA with 33.19MB, and the React Native application took the most space on Android – 50.51MB.

The information about PWA size in iOS is not available, therefore I only compared React Native and Flutter application sizes. The smaller-size application was the one implemented in React Native, taking 47.2MB of the available space in the device storage. The Flutter implementation occupied 81.3MB of the device storage.

There is a notable difference between the size of the Flutter application on Android and iOS – 48.11MB, whereas the difference for React Native was only 3.31MB. Unfortunately, I was unable to identify the cause of this difference.

Table 5.11: Comparison of application sizes.

Technology	Android	iOS
React Native	50.5MB	47.2MB
Flutter	33.2MB	81.3MB
PWA	319KB	n/a

5.4.2 Code Sharing

When it comes to code sharing, both technologies are comparable. In the example application, I was able to share close to 100% of the code between Android and iOS. That was mostly because the used libraries that have different native implementations for each platform but provide unified cross-platform API.

React Native codebase had six locations where it conditionally called different code for iOS and Android. Three of these calls were related to different styling on each platform to achieve as similar look as possible. The rest was related to a platform-specific behavior of map and location modules.

The Flutter implementation had only two places in the codebase with conditional code branches based on if the platform is iOS or Android. The first place was responsible for rendering the correct system alert when saving/discarding the currently tracked route and the second one handles a difference between Android and iOS location updating. However, when it comes to the web target, there were eleven occurrences in total where the functionality required to use a different code for the web platform. Two cases were caused by discrepancies between the usage of the camera module and uploading pictures to Firebase on the web and mobile platforms (Figure 5.3). The rest of the places where code was called differently for web target were caused by the fact that the platform_maps plugin did not support web target, and a different module had to be used.

5.4.3 Codebase Size

React Native codebase consisted of 51 files, excluding generated iOS, Android, IDE, and `node_modules` files. Out of these, 38 files contained code, and the remaining 13 files were used for configuration of the application and code quality tools such as linter[3] and prettier[4]. The React Native implementation has 2552 developer-written lines of code excluding any generated files or necessary changes made to these files.

3. Linter is a static code analysis tool that can be configured to report various programming errors or stylistic errors
4. Prettier is opinionated code formatter which can be configured to run through the codebase on save.

The Flutter codebase had 61 files, excluding the files generated for iOS, Android, and web platforms, as well as the files generated for the IDE and plugins. Forty-nine files were developer-written and included code. The remaining 12 files were configuration files required by Flutter, Firebase, and the linter. Dart code did not need to install a separate prettier code formatter because the plugins for each supported IDE included the Dartfmt tool, which is a standardized way to format Dart code. The Flutter implementation has 3691 developer-written lines of code, excluding generated files or changes made to these files. I also did not count the changes made in the forked `platform_maps` plugin.

5.4.4 Build Time

This subsection compares build times of the example application implementations. I measured *clean* and *cached* build times separately because the difference between the two is usually notable. One-way ANOVA was used to compare the Android, iOS, and PWA build times. The measured build times do not include build preparation time. The results are summarized in Table 5.12 and full data is presented in Appendix F.

Table 5.12: Summary of build times. Reported values are mean averages with standard deviation.

Platform	Clean Build			Cached Build		
	RN	Flutter	PWA	RN	Flutter	PWA
Android	57.48±5.11s	29.21±3.52s	36.54±2.27s	10.18±0.61s	3.58±0.48s	0.81±0.25s
iOS	379±12.88s	94.69±13.89s	36.54±2.27s	36.42±1.02s	5.58±0.93s	0.81±0.25s

5.4.4.1 PWA

The duration of web target builds was reported directly by the Flutter CLI when the command `$ flutter build web` finished. I decided to compare results gathered from web builds with Android and iOS build times in the following sections.

5.4.4.2 Android

I used the built-in tools in the Android Studio IDE to measure the build times of Android applications. After triggering the build of the Android Application Package (APK), developers can see information about the build by navigating to *View* → *Tool Windows* → *Build*. There are two tabs with different information. The first tab is *Build Output*, which provides a high-level overview of the total time since the build was triggered. *Build Analyzer*, on the second tab, shows a detailed time breakdown.

The ANOVA suggested that there was a significant difference in the effect of **used technology** on the **build time** between at least two measured groups in both clean and cached build test cases:

- Clean build ($F_{2,27} = 125.98$, $p < 0.05$)
- Cached build ($F_{2,27} = 947.79$, $p < 0.05$)

Tukey's HSD test showed that both native Flutter and PWA build times were significantly faster than React Native build times in both clean and cached scenarios. In the case of clean builds, native Flutter builds were faster than PWA build times. On the other hand, PWA builds were substantially faster than native Flutter builds in the cached scenario.

The difference between React Native and native Flutter clean build times seemed to have been caused by a build step that bundles JavaScript code and assets, as shown in figures 5.1 and 5.2.

5.4.4.3 iOS

To measure the build times of iOS applications, I used a tool provided by the Xcode IDE. Developers can measure build time with a time summary of different steps by navigating to the following option *Product* → *Perform Action* → *Build With Timing Summary*.

The ANOVA analysis suggested that there was a significant difference in the effect of **used technology** on the **build time** between at least two measured groups in both clean and cached build test cases:

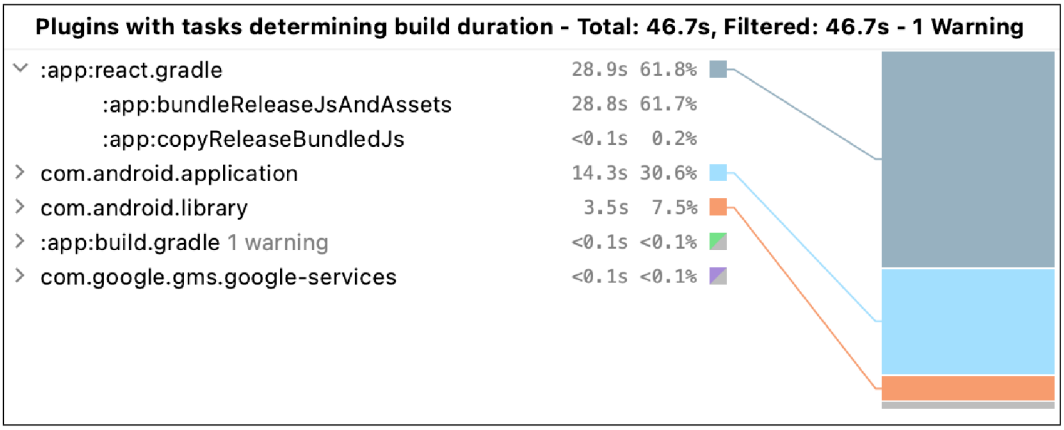

Figure 5.1: Build analyzer breakdown of React Native clean Android build.

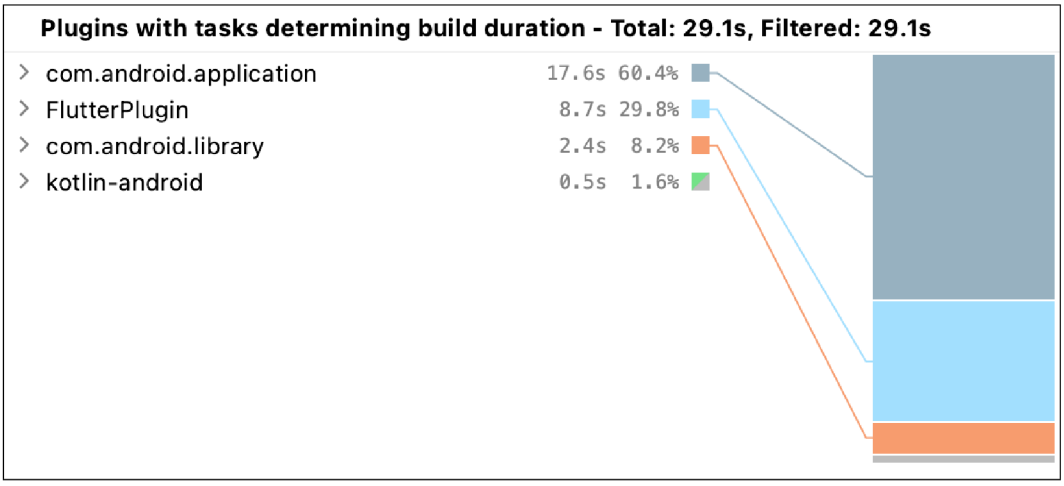

Figure 5.2: Build analyzer breakdown of Flutter clean Android build.

- Clean build ($F_{2,27} = 2490.13$, $p < 0.05$)
- Cached build ($F_{2,27} = 5104.13$, $p < 0.05$)

A post-hoc Tukey's HSD test showed that PWA and native Flutter implementations had significantly faster build times than React Native in clean and cached test scenarios. Additionally, PWA build times were substantially quicker than native Flutter builds in both cases as well.

The cause of such difference is the fact that React Native builds more components. My assumption is that since Flutter implements its widgets using the Dart language and renders them through the Skia graphical engine, it does not have to implement and build as many native modules as React Native, which has native implementation of each component it exposes. Additionally, a notable portion of React Native build time was spent bundling React Native code, on average ≈29.8s.

5.5 Summary

This section summarizes the results of the technologies comparison. There was no clear winner of the comparison as a whole, and each technology performed well in some aspects over the other. The list below serves as a high-level summary of the measured metrics:

- Application Startup Time – React Native and Flutter outperformed PWA on both platforms.

- Screen Render Times – React Native and Flutter outperformed PWA on all screens on both platforms. On Android, React Native was faster in rendering the *Map Screen*.
- CPU Usage – In the idle test scenario on the Android platform, React Native and Flutter had lower CPU usage in the *Route Detail Screen*, and React Native also used fewer resources in the *Gallery Screen*. In the active usage test scenario on Android, React Native and Flutter outperformed PWA in *Map, Gallery*, and *Route Detail Screens*. Additionally, Flutter did better than React Native in the *Gallery Screen*. In the idle test scenario on iOS, there were no significant differences between used technologies. However, in the active test scenario, React Native and Flutter used significantly fewer resources than PWA in the *Login, Map*, and *Route Detail Screens*. In the *Gallery Screen*, Flutter outperformed React Native and PWA, and PWA used fewer resources than React Native.
- Memory Usage – In the idle test case on Android, PWA used less memory in all screens than both native implementations. In the *Login* and *Map Screens*, React Native outperformed Flutter. On the other hand, Flutter was better in the *Gallery* and *Route Detail Screens*. In the active test scenario on Android, PWA outperformed native implementations on all screens. In the *Login* and *Map Screens*, React Native did better than Flutter. In the *Route Detail Screen*, Flutter used less memory than React Native. During the idle test scenario on iOS, PWA outperformed native implementations on all screens. React Native did better than Flutter in the *Login, Map*, and *Route Detail Screens*, while Flutter was better in the *Gallery Screen*. In the active test scenario on iOS, PWA used less memory than the native implementations on all screens. Additionally, React Native outperformed Flutter on all screens.
- Frames per Second – On Android, Flutter bested React Native in the *Map Screen*, while React Native showed better results in the *Route Detail Screen*. On iOS, there were no significant differences between React Native and Flutter.
- Application Size – On Android, PWA had the smallest size, followed by Flutter, while React Native application was the largest. On iOS, React Native produced a smaller-sized application than Flutter. PWA size could not be determined.
- Code Sharing – Both technologies allowed for close to 100% shared application code.
- Codebase Size – React Native's codebase size was smaller than the size of Flutter codebase.
- Build Time – React Native had the slowest build times on both platforms and in both clean and cached scenarios. Flutter was faster than PWA when it came to clean builds on the Android platform. In the rest of the cases, PWA was the fastest to build.

5.6 Personal Experience

I consider both React Native and Flutter to be great technologies. The implementation part of the thesis had its obstacles, but overall, it was not difficult to create fully cross-platform applications. In general, both used frameworks were reasonably easy to learn, and the *entry barrier* was insignificant. I think that both React Native and Flutter had good enough documentation of the core components. However, React Native has more extensive documentation of external plugins over Flutter.

I believe the React Native JSX UI composition syntax was more natural than the object-oriented way of composing UI in Flutter. Additionally, Flutter requires developers to write more boilerplate code due to the object-oriented Dart language. The especially painful experience was working with Redux. The logger middleware also required significantly more code because a JSON encoding method had to be written for each Redux action. However, I was surprised by the speed of the Flutter application builds. When it came to the iOS build system, React Native was far behind Flutter.

Without any native plugins, building a progressive web application out of a Flutter codebase can be done in a matter of seconds. However, I suppose the web target of Flutter is not yet fully mature at the moment of writing due to the issues with external libraries such as the maps and camera libraries.

5.6.1 Implementation Obstacles

This section describes some of the obstacles, inconveniences, and cumbersome behavior I encountered during implementation. Some of them were most likely caused by inexperience with the specific technology, but several issues had different reasons. The most significant issues and obstacles are presented below.

Styling in React Native: There is significant difference in the behavior of shadows in Android and iOS. React Native translates its components into the underlying operating system elements using the system's APIs. Most visual style APIs are similar between these platforms, but shadows are an exception. In iOS, the shadow is set using the `shadowColor` style property and the iOS-only `shadowOffset`, `shadowOpacity`, and `shadowRadius` style properties [19]. In Android, the only way to implement shadow for a component is through the `elevation` style property (optionally combined with the `shadowColor` property) [20]. It is non-trivial to achieve a consistent look between these platforms. Furthermore, there are some other caveats. For `elevation` to properly apply shadow on Android, the component must also have `backgroundColor` style property applied. This behavior is not documented and can be very hard to debug. Additionally, the `overflow` layout property affects the behavior of iOS shadows but not the Android elevation property, which is also not documented and quite cumbersome. To complicate things further, the `overflow` property also affects `borderRadius` property but only on the Android platform. Lastly, the `elevation` property also affects the z-index ordering of the components, which further complicates UI composition. In general, this behavior is very confusing and messy.

Missing features in the `platform_maps` plugin: As mentioned earlier, there was no single best maps library for Flutter. I used the only library that provided sufficient cross-platform functionality. However, I soon encountered issues with missing features that prevented the application from working similarly to the one implemented in React Native. After considering the options, I decided to fork the repository and implement some of the simple missing functionality myself. I created a pull request in the source repository, but unfortunately, it looks like the repository is not maintained regularly, and, at the time of writing, the pull request was still pending review[5].

Missing support for the web target in the `platform_maps` plugin There were several options for resolving this, such as creating another more complex pull request in the platform_maps plugin repository. But due to the lack of responsiveness from the repository maintainers, I decided to resolve it by installing a different package for the web target with a comparable API.

Incomplete documentation of the camera plugin: There were several undocumented differences in the usage of the Flutter camera plugin for mobile platforms and the web platform. It was a severe pain point to figure out the correct usage of the web target and how to upload the results to Firebase Cloud Storage even though the solution was relatively simple (Figure 5.3).

5.6.2 Advantages/Disadvantages of React Native and Flutter

The main perceived advantages of React Native are:

- the JSX UI composition syntax, which should feel natural to anyone coming from web development,
- automated JSON encoding and decoding out of the box,
- less boilerplate code.

The main disadvantages are:

- The need for serialized JSON communication between JavaScript part of the framework and native part of the framework,
- the styling differences between Android and iOS for shadows and several other properties.

Similarly, the main advantages of Flutter include:

- The fact that the Skia graphical engine handles the rendering part of the framework,

5. `https://github.com/LuisThein/platform_maps_flutter/pull/44`.

```
// previous code omitted

String refName = '/${currentUser.uid}/$routeId/$imgId.jpg';
Reference ref = storage.ref(refName);

final fn = kIsWeb
    ? () => ref.putData(
        action.data!, SettableMetadata(contentType: 'image/jpeg'))
    : () => ref.putFile(File(action.path));

return fn()
    .asStream()
    .switchMap((_) => ref.getDownloadURL().asStream().switchMap((url) {
          RouteImage img = RouteImage(
            id: imgId,
            url: url,
            coordinates: coords,
          );

          return Stream.value(OnPictureUploadedAction(image: img));
        }))
    .handleError((err) {
  logger.error('_pictureTakenEpic - err uploading picture', {
    'err': err,
  });
  return Stream.value(
      OnPictureUploadFailedAction(path: refName, err: err)
  );
```

Figure 5.3: The relevant part of the code responsible for uploading images to Firebase Cloud Storage.

- the fast build times,
- the aim of the Flutter project to go beyond mobile and support a variety of platforms.

The main disadvantages are:

- The amount of boilerplate code,
- more complicated JSON serialization and deserialization,
- the confusing fact that some style properties were separate widgets (e.g., the Padding widget) but other similar style properties were attributes of widgets instead (e.g., margin).

6 Conclusion

The goal of this thesis was to compare several cross-platform technologies (React Native and Flutter) and philosophies (compile-to-native and PWA), and design and implement a non-trivial example application in each of them. I designed and implemented a location tracking application inspired by Strava and Calimoto applications. Next, I conducted an empirical comparison of the implementations and described my personal experience with the used technologies.

The empirical comparison consisted of statistical evaluation of differences between several time behavior characteristics (e.g., application startup time, render times of screens, or application build times) and resource utilisation metrics, such as CPU usage, memory consumption, or frames per second.

My personal experience included different aspects of used technologies, their learning curves, and the obstacles that I encountered during the implementation of the example applications.

To conclude, there was no clear winner between React Native and Flutter. Both frameworks performed similarly in most of the measured metrics, and the end-user experience was very good in both cases. React Native has a lower entry barrier as it uses JavaScript/TypeScript language to develop applications. The Dart language, used by Flutter, is not as well-known, and fewer developers are proficient with it. This difference is even more significant if the developer has previous experience with web development. React Native also had better documentation of external libraries used in the example application than Flutter. On the other hand, Flutter outperformed React Native in the build times, making the development experience more pleasant.

When it comes to the PWA implementation, the performance difference was substantial, and the user experience was much worse than in the case of compile-to-native applications. However, it is unclear if the nature of the example application or the fact that Flutter was used for the implementation were the reasons.

For general purpose applications, cross-platform technologies are a viable alternative to native applications and can be used to save development and time resources. In addition, they do not require new developers to understand and know the specifics of the underlying platforms, especially when developing less complex applications. However, when it comes to performance-intensive applications and games, I would still recommend considering a native implementation.

Bibliography

[1] Android. *What is Android*. URL: https://www.android.com/what-is-android/ (visited on Nov. 11, 2021).

[2] Esteban Angulo and Xavier Ferre. "A Case Study on Cross-Platform Development Frameworks for Mobile Applications and UX". In: (2014). DOI: 10.1145/2662253.2662280. URL: https://dl.acm.org/doi/10.1145/2662253.2662280 (visited on Nov. 12, 2021).

[3] Elasticsearch B.V. *What is the ELK Stack?* URL: https://www.elastic.co/what-is/elk-stack (visited on Nov. 18, 2021).

[4] Designtechnica Company. *What is Android fragmentation, and can Google ever fix it?* 2018. URL: https://www.digitaltrends.com/mobile/what-is-android-fragmentation-and-can-google-ever-fix-it/ (visited on Nov. 12, 2021).

[5] AVADirect Custom Computers. *Frame Rate (FPS) vs Refresh Rate (Hz)*. URL: https://www.avadirect.com/blog/frame-rate-fps-vs-hz-refresh-rate/.

[6] Mozilla Corporation. *First contentful paint*. URL: https://developer.mozilla.org/en-US/docs/Glossary/First_contentful_paint (visited on Nov. 23, 2021).

[7] Mozilla Corporation. *Introduction to progressive web apps*. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction (visited on Nov. 13, 2021).

[8] Jonathan Hayward. *Reactive Programming with JavaScript*. Birmingham: Packt Publishing, 2015. ISBN: 1783558555.

[9] Adobe I/O. *Update for Customers Using PhoneGap and PhoneGap Build*. 2020. URL: https://blog.phonegap.com/update-for-customers-using-phonegap-and-phonegap-build-cc701c77502c (visited on Nov. 12, 2021).

[10] LogRocket Inc. *Why use Redux? A tutorial with examples*. URL: https://blog.logrocket.com/why-use-redux-reasons-with-clear-examples-d21bffd5835/ (visited on Nov. 17, 2021).

[11] Apple Inc. *iOS 14 Network privacy permission check is not shown after reinstall, nor is the toggle shown in Settings -> Privacy -> Local Network*. URL: https://developer.apple.com/forums/thread/660485 (visited on Nov. 11, 2021).

[12] Apple Inc. *Swift – The powerful programming language that is also easy to learn.* URL: https://developer.apple.com/swift/ (visited on Nov. 11, 2021).

[13] Apple Inc. *SwiftUI*. URL: https://developer.apple.com/xcode/swiftui/ (visited on Nov. 11, 2021).

[14] CollectiveMind Inc. *React Native's upcoming re-architecture*. URL: https://collectivemind.dev/blog/react-native-re-architecture (visited on Nov. 13, 2021).

[15] Facebook Inc. *Components and Props*. URL: https://reactjs.org/docs/components-and-props.html (visited on Nov. 24, 2021).

[16] Facebook Inc. *Hermes, JavaScript engine optimized for React Native*. URL: https://hermesengine.dev/ (visited on Nov. 23, 2021).

[17] Facebook Inc. *Native Modules Intro*. URL: https://reactnative.dev/docs/native-modules-intro (visited on Nov. 13, 2021).

[18] Facebook Inc. *React Native in H2 2021*. URL: https://reactnative.dev/blog/2021/08/19/h2-2021o (visited on Nov. 13, 2021).

[19] Facebook Inc. *Shadow Props*. URL: https://reactnative.dev/docs/shadow-props#props (visited on Nov. 18, 2021).

[20] Facebook Inc. *View Style Props*. URL: https://reactnative.dev/docs/view-style-props#elevation-android (visited on Nov. 18, 2021).

[21] Statista Inc. *Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2021*. 2021. URL: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/ (visited on Nov. 11, 2021).

[22] Statista Inc. *Number of smartphone users from 2016 to 2021*. 2021. URL: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (visited on Nov. 10, 2021).

[23] Wikimedia Foundation Inc. *Android, The plaftorm pushing what's possible*. URL: https://www.android.com/ (visited on Nov. 11, 2021).

[24] ISO/IEC. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en (visited on Nov. 20, 2021).

[25] Google LLC. *About Skia*. URL: https://skia.org/about/ (visited on Nov. 14, 2021).

[26] Google LLC. *Flutter architectural overview*. URL: https://docs.flutter.dev/resources/architectural-overview (visited on Nov. 14, 2021).

[27] A. Margara and T. Rabl. "Definition of Data Streams." In: *Encyclopedia of Big Data Technologies*. Ed. by S. Sakr and A.Y. Zomaya. Springer, Cham, 2019. URL: https://doi.org/10.1007/978-3-319-77525-8_188.

[28] Rob Napier and Kumar Mugunth. *iOS 7 Programming Pushing the Limits. Develop Advance Applications for Apple iPhone, iPad, and iPod Touch*. John Wiley & Sons, Ltd, 2014. ISBN: 1118818342.

[29] Ian Sommerville. *Software Engineering (9th Edition)*. London: Pearson, 2011. ISBN: 0137035152.

[30] StatCounter. *Mobile Operating System Market Share Worldwide*. 2021. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide (visited on Nov. 10, 2021).

[31] Eirch Gamma; Richard Helm; Ralph Johnson; John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN: 0201633612.

A List of Electronic Attachments

- FavoriteRoutes-RN.zip – Source code of the React Native implementation.
- FavoriteRoutes-Flutter.zip – Source code of the Flutter implementation.
- statistical_analysis.xlsx – ANOVA and T-Test computations on all the statistically compared data.

B Installation Manual for Developed Applications

In order to build and run the example application build in React Native, we need to do the following:

- Follow the `React Native CLI Quickstart` installation guide at `https://reactnative.dev/docs/environment-setup` (we need a mac with Xcode IDE installed in order to run iOS version of the application).
- Install the `yarn` package manager (install guide available at `https://yarnpkg.com/`).
- Navigate to the project root in a terminal and use command `yarn install-native` in case that we are using mac, if we are planning to use only the Android implementation `yarn install` is sufficient.
- Start the application by calling `yarn android` or `yarn ios`.

In order to build and run the Flutter version of the application, we need to perform following steps:

- Follow the installation guide at `https://docs.flutter.dev/get-started/install`
- Navigate to the project root in terminal and run command `flutter pub get`.
- Start the application by calling `flutter run`. By default flutter will start the we version of the application. If we start device emulator or connect physical device, this command will build the application to the running target (or let the user select a destination if there are multiple running targets).

C User Interface

This appendix shows screenshots of each screen of the example application in all of its possible states. The screenshots were taken using the React Native implementation, but visual differences are between implementations are negligible.

C.1 Sign In

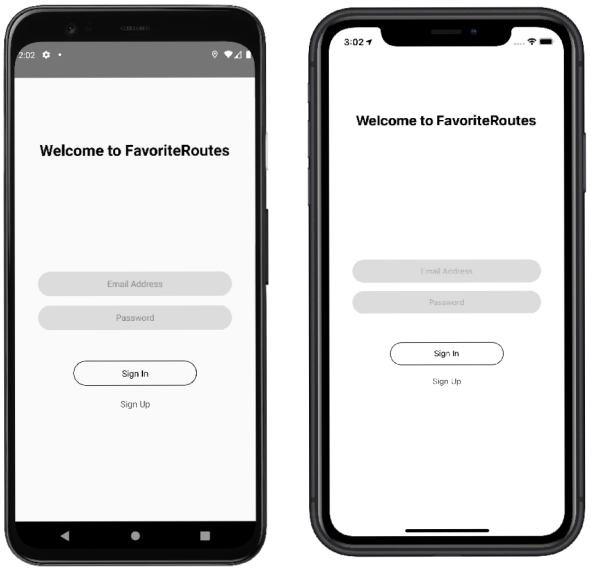

Figure C.1: Sign in page.

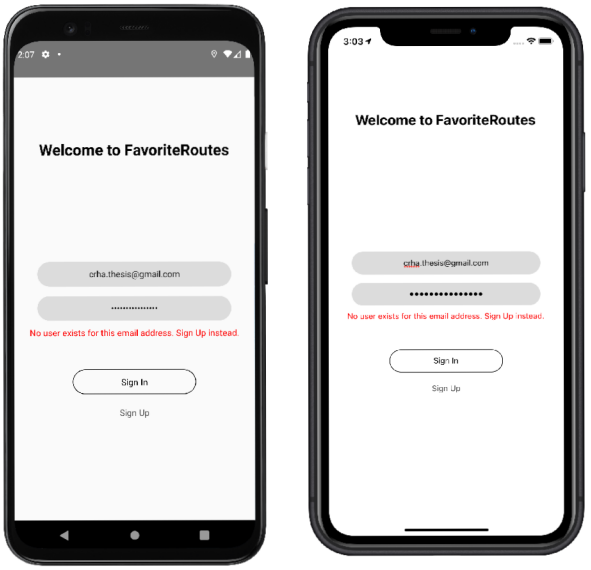

Figure C.2: Sign in page error.

C.2 Map Screen

Figure C.3: Main map page.

Figure C.4: Main map page – tracking a route[1].

1. Android simulator does not have a ability to simulate location tracking, hence, the line representing the route is only shown in iOS screenshot.
2. The screenshots are taken from Android and iOS simulators that do not show actual camera screens but only a mocked one for android and black screen for iOS.

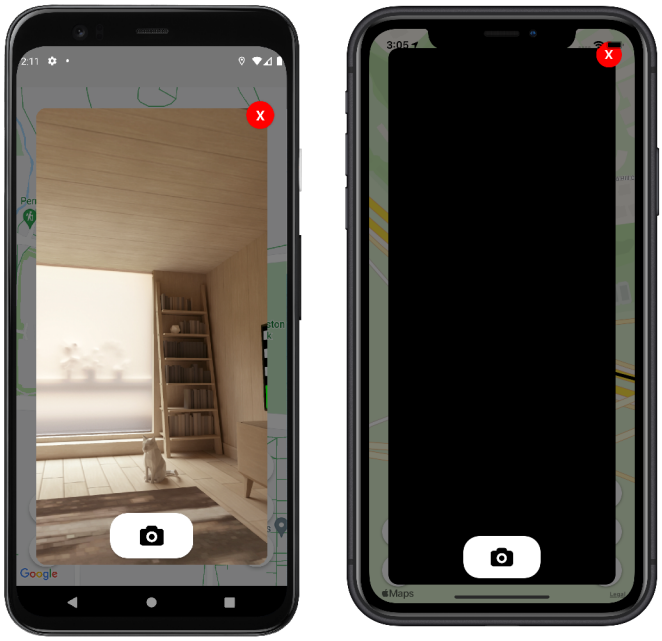

Figure C.5: Camera modal on top of the main map page[2].

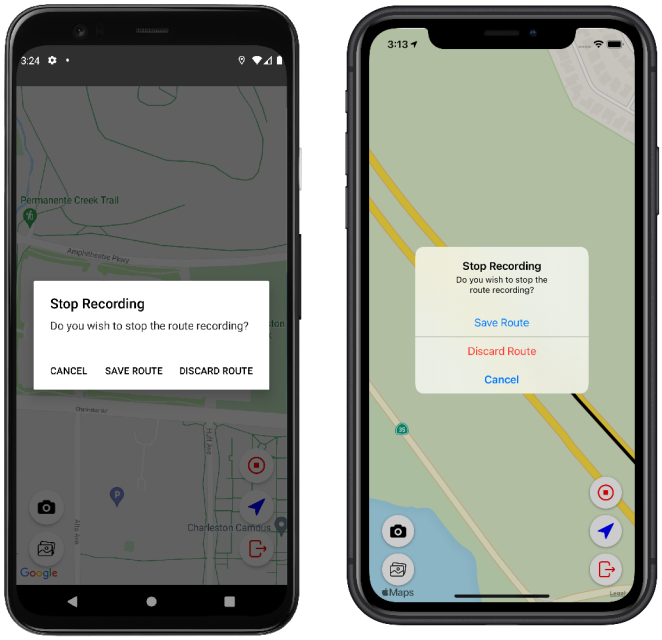

Figure C.6: System alert, which allows the user to save or discard the tracked route.

C.3 Gallery Screen

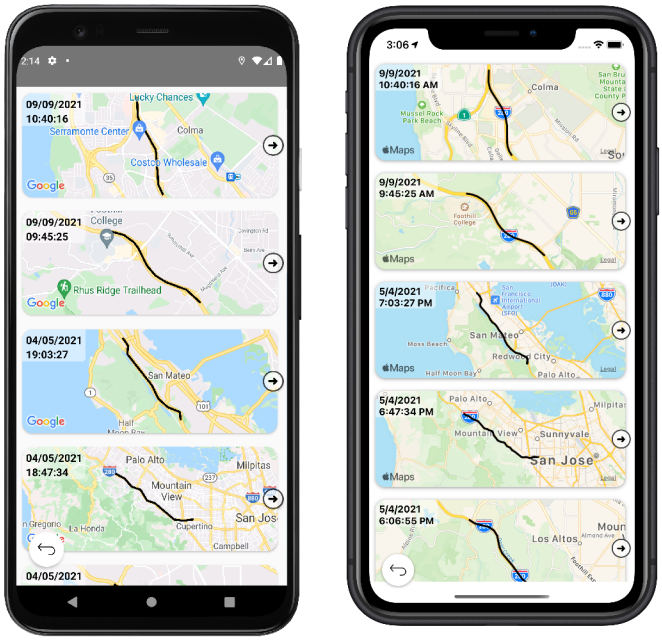

Figure C.7: Routes gallery page.

C.4 Route Detail Screen

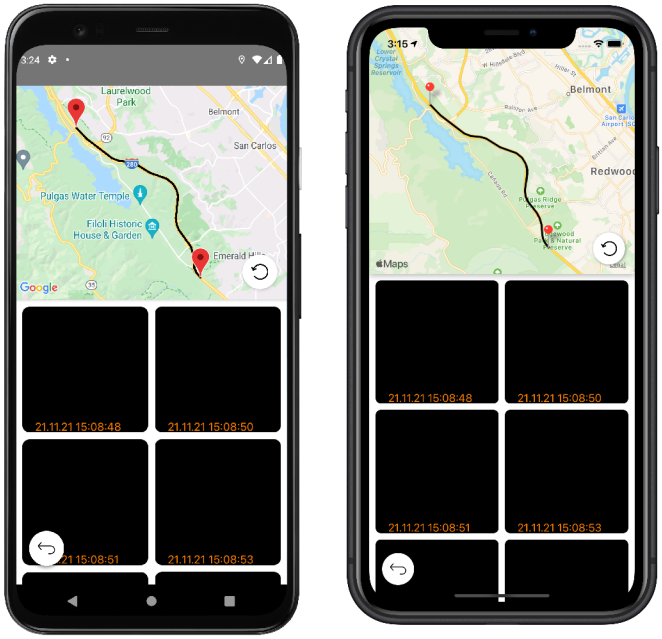

Figure C.8: Route detail with images[3].

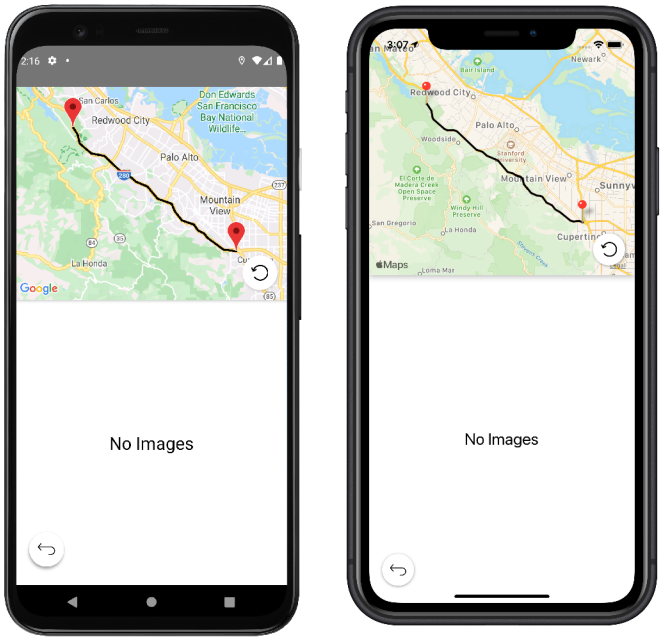

Figure C.9: Route detail with no images.

3. The images were taken in an iOS simulator, which shows just a black background and the time of the photo.

D File Structure

This appendix shows screenshots of the full file structures of React Native and Flutter implementations. It included in order to provide visual representation to purely textual description available in subsection 4.3.3.

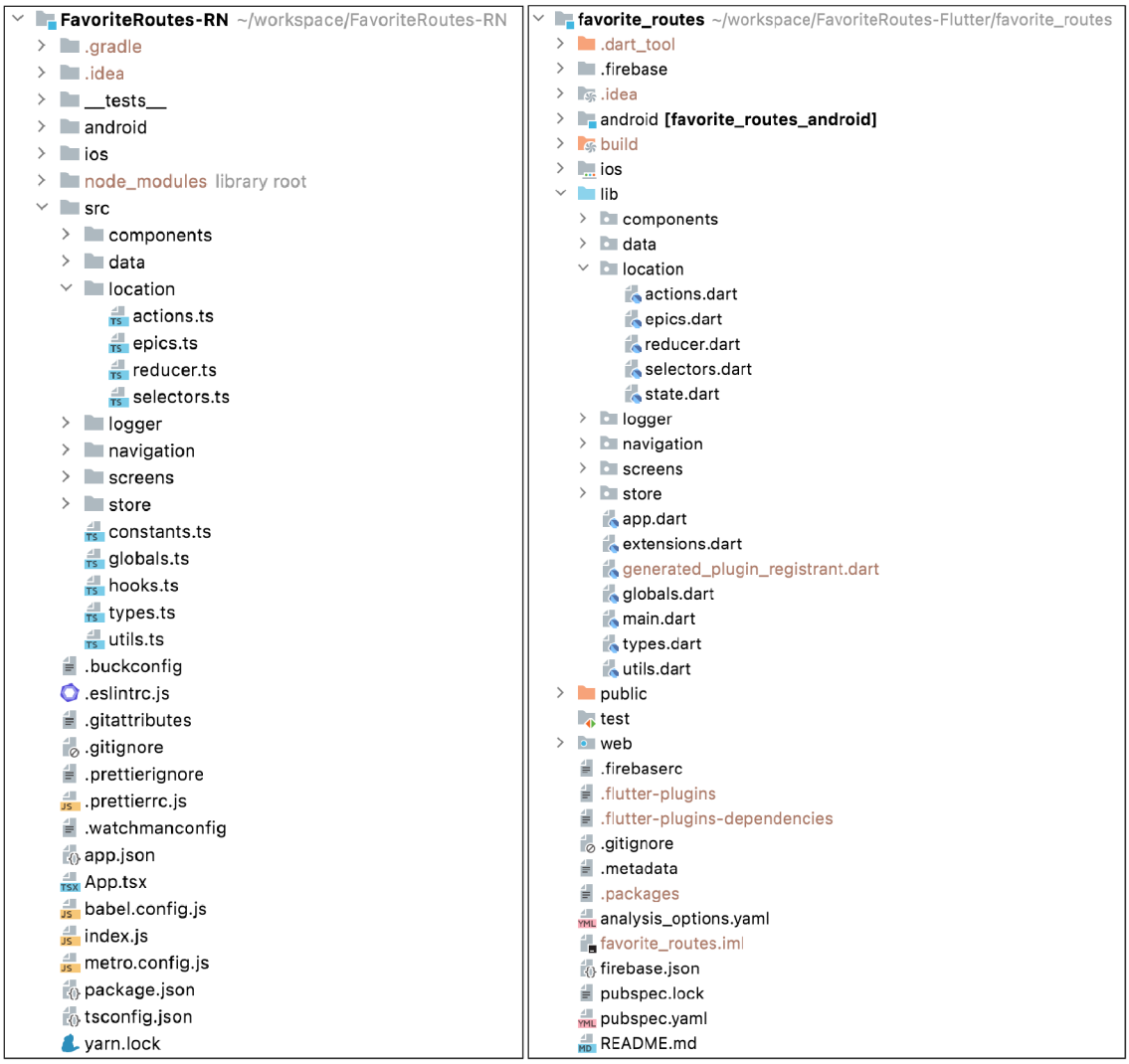

Figure D.1: React Native (left) and Flutter (right) file structures.

E Tooling Screenshots

This appendix provides screenshot from tools used to measure the application CPU and memory consumption. It servers as a visual addition to tools described in subsection 5.3.2.1.

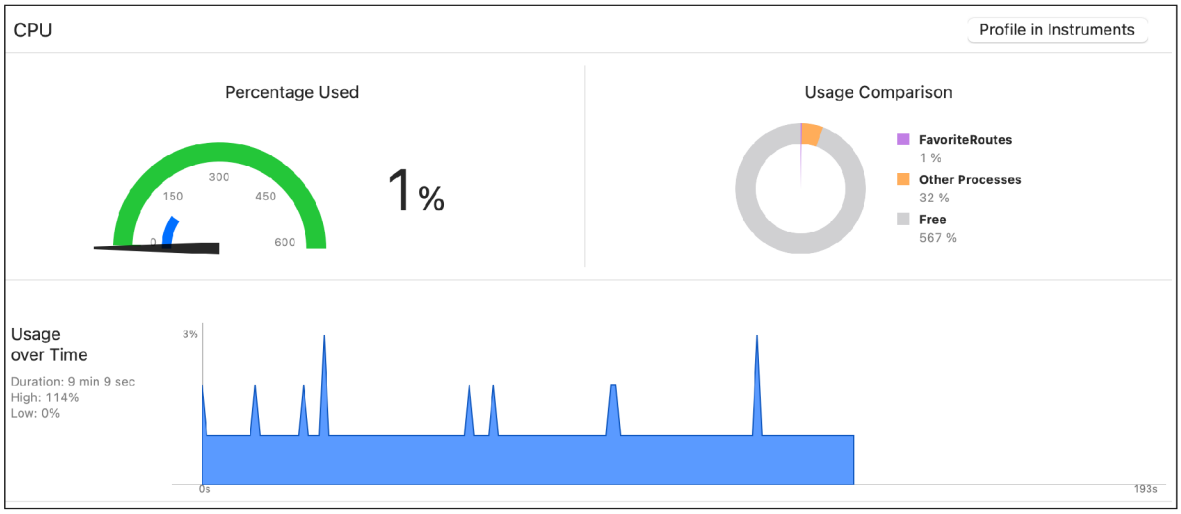

Figure E.1: Xcode Debug Gauges – one of the available views for CPU monitoring.

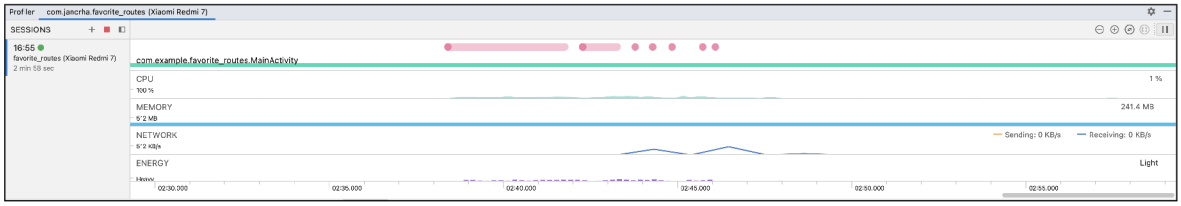

Figure E.2: Android Profiler example.

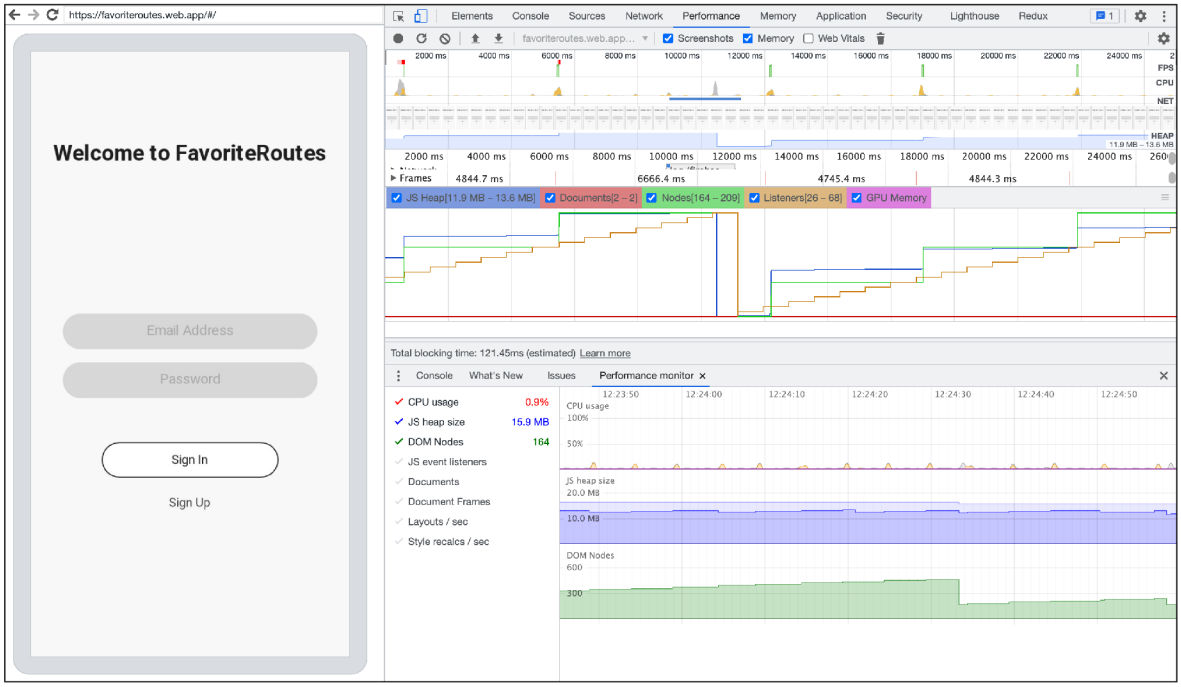

Figure E.3: Google Chrome Remote Debugging.

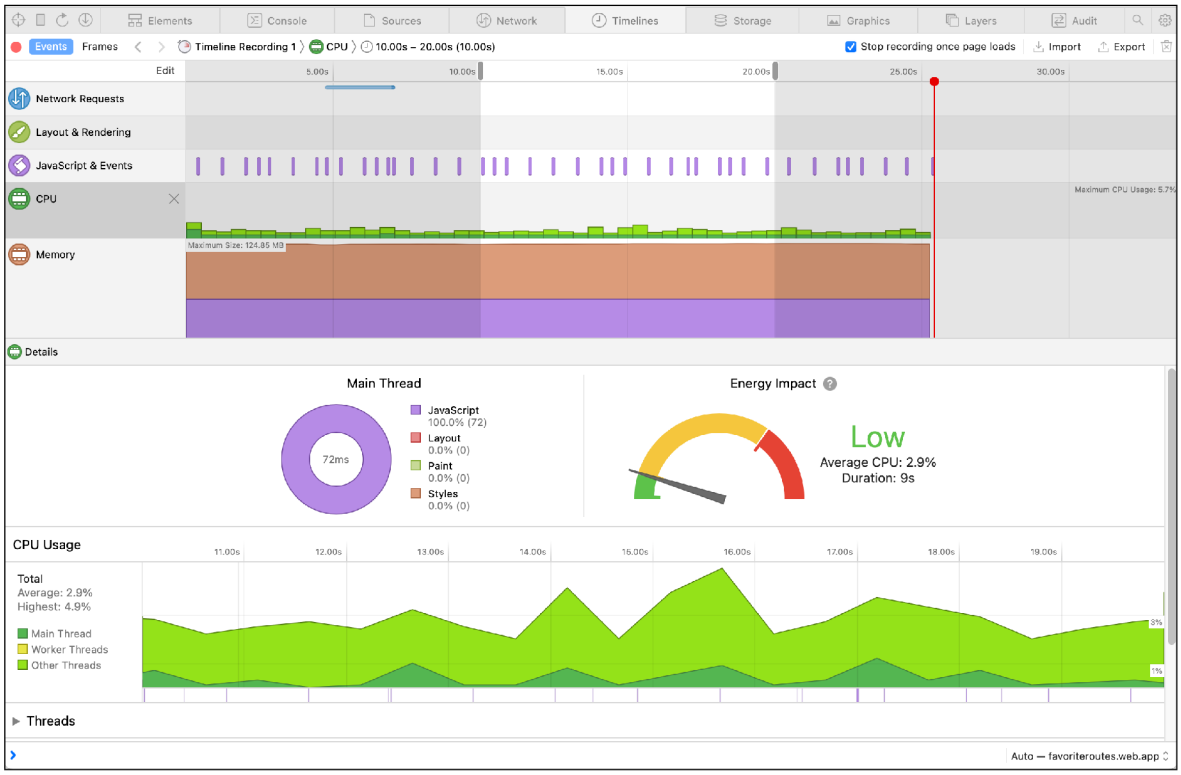

Figure E.4: Safari Web Inspector.

F Measured Data

This appendix contains individual measurements of all metrics. The data are grouped into tables according to relevance. Each table has a caption that describes the data.

Startup Times The following tables contain individual measured values and calculated statistical results of application startup times. *First Contentful Paint* was considered as web alternative to native application startup time.

Table F.1: Native applications startup times.

Test No.	Android		iOS	
	React Native	Flutter	React Native	Flutter
1	291ms	225ms	208ms	193ms
2	282ms	365ms	151ms	220ms
3	226ms	281ms	237ms	185ms
4	215ms	216ms	215ms	136ms
5	197ms	231ms	145ms	206ms
6	208ms	189ms	224ms	154ms
7	214ms	239ms	217ms	283ms
8	198ms	208ms	211ms	227ms
9	169ms	206ms	109ms	182ms
10	244ms	176ms	211ms	197ms
Mean Avg.	224.4ms	233.6ms	192.8ms	198.3ms
Median	214.5ms	220.5ms	211ms	195ms
Pop. Variance	1312.24	2665.64	1595.36	1484.41
Std. Deviation	36.23ms	51.63ms	39.94ms	38.53ms

Table F.2: First contentful paint times of PWA.

Test No.	Clean FCP		Cached FCP	
	Android	iOS	Android	iOS
1	3.83s	2.22s	1.02s	0.56s
2	3.79s	2.26s	1.35s	0.61s
3	4.04s	2.28s	1.14s	0.52s
4	4.12s	2.32s	0.98s	0.53s
5	3.97s	2.23s	1.09s	0.59s
6	3.93s	2.34s	1.19s	0.51s
7	3.85s	2.19s	1.26s	0.59s
8	3.89s	2.24s	1.38s	0.58s
9	3.78s	2.15s	1.21s	0.6s
10	3.91s	2.29s	1.06s	0.63s
Mean Avg.	3.911s	2.252s	1.168s	0.572s
Median	3.9s	2.25s	1.165s	0.585s
Pop. Variance	0.011	0.003	0.017	0.002
Std. Deviation	0.103s	0.055s	0.13s	0.04s

Screen Rendering Times Tables presented in this paragraph contain measured values and calculated statistical results of screen rendering times.

Table F.3: Login screen render times.

Test No.	Android			iOS		
	RN	Flutter	PWA	RN	Flutter	PWA
1	16ms	12ms	26ms	7ms	2ms	15ms
2	9ms	10ms	29ms	2ms	5ms	19ms
3	14ms	13ms	20ms	3ms	2ms	20ms
4	16ms	8ms	23ms	3ms	2ms	17ms
5	16ms	9ms	31ms	3ms	3ms	22ms
6	18ms	11ms	25ms	2ms	4ms	23ms
7	15ms	15ms	27ms	3ms	3ms	20ms
8	10ms	10ms	29ms	2ms	2ms	19ms
9	15ms	9ms	22ms	2ms	6ms	19ms
10	12ms	12ms	30ms	6ms	2ms	18ms
Mean Avg.	14.1ms	10.9ms	26.2ms	3.3ms	3.1ms	19.2ms
Median	15ms	10.5ms	26.5ms	3ms	2.5ms	19ms
Pop. Variance	7.49	4.09	12.16	2.81	1.89	4.76
Std. Deviation	2.74ms	2.02ms	3.49ms	1.68ms	1.37ms	2.18ms

Table F.4: Map screen render times.

Test No.	Android			iOS		
	RN	Flutter	PWA	RN	Flutter	PWA
1	27ms	30ms	61ms	5ms	7ms	33ms
2	22ms	31ms	58ms	12ms	10ms	30ms
3	23ms	28ms	50ms	12ms	15ms	29ms
4	21ms	23ms	59ms	12ms	8ms	35ms
5	19ms	33ms	67ms	12ms	8ms	30ms
6	21ms	31ms	53ms	4ms	10ms	31ms
7	22ms	27ms	63ms	8ms	14ms	31ms
8	21ms	25ms	68ms	8ms	12ms	35ms
9	20ms	29ms	60ms	6ms	12ms	28ms
10	18ms	26ms	50ms	10ms	11ms	30ms
Mean Avg.	21.4ms	28.3ms	58.9ms	8.9ms	10.7ms	31.2ms
Median	21ms	28.5ms	59.5ms	9ms	10.5ms	30.5ms
Pop. Variance	5.44	8.61	36.49	8.89	6.21	5.16
Std. Deviation	2.33ms	2.93ms	6.04ms	2.98ms	2.49ms	2.27ms

Table F.5: Gallery screen render times.

Test No.	Android			iOS		
	RN	Flutter	PWA	RN	Flutter	PWA
1	42ms	58ms	1126ms	28ms	32ms	432ms
2	43ms	67ms	1194ms	24ms	30ms	401ms
3	31ms	60ms	1025ms	29ms	27ms	459ms
4	41ms	59ms	989ms	26ms	29ms	398ms
5	46ms	51ms	1215ms	28ms	33ms	482ms
6	42ms	66ms	1189ms	32ms	39ms	454ms
7	41ms	49ms	1014ms	26ms	35ms	502ms
8	45ms	52ms	964ms	28ms	33ms	441ms
9	67ms	55ms	1202ms	29ms	37ms	494ms
10	33ms	68ms	1165ms	27ms	29ms	406ms
Mean Avg.	43.1ms	58.5ms	1108.3ms	27.7ms	32.4ms	446.9ms
Median	42ms	58.5ms	1145.5ms	28ms	32.5ms	447.5ms
Pop. Variance	84.29	42.25	8843.61	4.21	13.04	1311.09
Std. Deviation	9.18ms	6.5ms	94.04ms	2.05ms	3.61ms	36.21ms

Table F.6: Detail screen render times.

Test No.	Android			iOS		
	RN	Flutter	PWA	RN	Flutter	PWA
1	16ms	19ms	99ms	10ms	9ms	45ms
2	15ms	21ms	91ms	6ms	11ms	45ms
3	14ms	20ms	101ms	7ms	10ms	39ms
4	12ms	18s	87ms	7ms	9ms	50ms
5	13ms	24ms	85ms	8ms	10ms	49ms
6	19ms	22ms	121ms	7ms	12ms	48ms
7	15ms	21ms	105ms	6ms	8ms	39ms
8	13ms	20ms	98ms	7ms	9ms	42ms
9	13ms	21ms	112ms	7ms	8ms	45ms
10	15ms	23ms	94ms	7ms	10ms	43ms
Mean Avg.	14.5ms	20.9ms	99.3ms	7.2ms	9.6ms	44.5ms
Median	14.5ms	21ms	98.5ms	7ms	9.5ms	45ms
Pop. Variance	3.65	2.89	112.21	1.16	1.44	13.25
Std. Deviation	1.91ms	1.7ms	10.59ms	1.08ms	1.2ms	3.64ms

CPU Usage The following tables were constructed from the results of CPU usage measurements. Each table contains test results of one screen on one specific platform. Framework names are abbreviated to RN in the case of React Native and F. in the case of Flutter.

Table F.7: Idle/Active CPU usage on the Login screen on iOS.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	1%	1%	1%	11%	13%	28%
2	2%	1%	1%	9%	11%	29%
3	1%	2%	2%	15%	9%	28%
4	1%	1%	1%	18%	17%	26%
5	2%	1%	2%	22%	18%	29%
6	3%	3%	2%	20%	18%	30%
7	1%	1%	2%	19%	16%	30%
8	1%	2%	1%	13%	12%	28%
9	3%	2%	1%	12%	13%	28%
10	1%	1%	2%	10%	9%	29%
Mean Avg.	1.6%	1.5%	1.5%	14.9%	13.6%	28.5%
Median	1%	1%	1.5%	14%	13%	28.5%
Pop. Variance	0.64	0.45	0.25	18.89	10.84	1.25
Std. Deviation	0.8%	0.67%	0.5%	4.35%	3.29%	1.12%

Table F.8: Idle/Active CPU usage on the Map screen on iOS.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	1%	2%	2%	47%	49%	73%
2	1%	1%	1%	49%	41%	76%
3	3%	1%	1%	52%	45%	69%
4	3%	1%	1%	48%	53%	74%
5	2%	3%	1%	48%	50%	74%
6	3%	3%	1%	49%	49%	70%
7	1%	2%	2%	51%	48%	78%
8	2%	2%	2%	47%	42%	71%
9	1%	1%	1%	48%	43%	77%
10	2%	1%	2%	50%	49%	70%
Mean Avg.	1.9%	1.7%	1.4%	48.9%	46.9%	73.2%
Median	2%	1.5%	1%	48.5%	48.5%	73.5%
Pop. Variance	0.69	0.61	0.24	2.49	13.89	8.96
Std. Deviation	0.83%	0.78%	0.49%	1.58%	3.73%	2.99%

Table F.9: Idle/Active CPU usage on the Gallery screen on iOS.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	2%	1%	3%	110%	49%	101%
2	2%	3%	3%	148%	65%	97%
3	2%	5%	2%	178%	87%	105%
4	1%	2%	1%	194%	101%	119%
5	3%	7%	2%	215%	112%	107%
6	1%	2%	1%	203%	107%	112%
7	2%	1%	2%	167%	94%	114%
8	1%	1%	3%	136%	73%	117%
9	1%	2%	2%	153%	69%	106%
10	2%	3%	2%	129%	81%	121%
Mean Avg.	1.7%	2.7%	2.1%	163.3%	83.8%	109.9%
Median	2%	2%	2%	160%	84%	109.5%
Pop. Variance	0.41	3.41	0.49	1052.41	365.16	57.09
Std. Deviation	0.64%	1.85%	0.7%	32.44%	19.11%	7.56%

Table F.10: Idle/Active CPU usage on the Route Detail screen on iOS.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	2%	1%	2%	52%	51%	84%
2	2%	1%	2%	55%	51%	85%
3	1%	3%	2%	55%	49%	83%
4	1%	1%	1%	51%	54%	87%
5	2%	4%	2%	50%	56%	87%
6	1%	2%	3%	50%	53%	82%
7	2%	1%	3%	50%	56%	84%
8	1%	1%	2%	52%	52%	84%
9	1%	1%	3%	51%	52%	87%
10	1%	2%	2%	50%	51%	85%
Mean Avg.	1.4%	1.7%	2.2%	51.6%	52.5%	84.8%
Median	1%	1%	2%	51%	52%	84.5%
Pop. Variance	0.24	1.01	0.36	3.44	4.65	2.76
Std. Deviation	0.49%	1.01%	0.6%	1.85%	2.16%	1.66%

Table F.11: Idle/Active CPU usage on the Login screen on Android.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	1%	1%	1%	5%	7%	10%
2	1%	1%	2%	7%	6%	11%
3	1%	1%	1%	5%	11%	10%
4	1%	1%	2%	10%	9%	9%
5	1%	1%	1%	14%	8%	11%
6	2%	1%	1%	9%	12%	11%
7	1%	1%	2%	8%	13%	10%
8	1%	2%	1%	6%	9%	9%
9	1%	2%	2%	9%	7%	12%
10	2%	1%	1%	6%	9%	9%
Mean Avg.	1.2%	1.2%	1.4%	7.9%	9.1%	10.2%
Median	1%	1%	1%	7.5%	9%	10%
Pop. Variance	0.16	0.16	0.24	6.89	4.69	0.96
Std. Deviation	0.4%	0.4%	0.49%	2.63%	2.17%	0.98%

Table F.12: Idle/Active CPU usage on the Map screen on Android.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	1%	1%	2%	13%	15%	38%
2	3%	1%	1%	13%	12%	36%
3	2%	1%	1%	16%	17%	39%
4	2%	3%	2%	19%	21%	35%
5	1%	1%	2%	15%	18%	36%
6	2%	1%	2%	17%	18%	39%
7	2%	3%	1%	13%	15%	38%
8	1%	1%	3%	14%	13%	38%
9	3%	1%	2%	13%	12%	36%
10	1%	2%	1%	17%	15%	35%
Mean Avg.	1.8%	1.5%	1.7%	15%	15.6%	37%
Median	2%	1%	2%	14.5%	15%	37%
Pop. Variance	0.56	0.65	0.41	4.2	7.64	2.2
Std. Deviation	0.75%	0.81%	0.64%	2.05%	2.76%	1.48%

Table F.13: Idle/Active CPU usage on the Gallery screen on Android.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	3%	6%	5%	33%	21%	82%
2	4%	5%	7%	35%	23%	79%
3	1%	9%	6%	30%	22%	85%
4	5%	7%	7%	35%	27%	89%
5	7%	4%	7%	38%	29%	87%
6	3%	1%	9%	37%	29%	91%
7	2%	3%	7%	33%	26%	93%
8	2%	5%	5%	31%	28%	80%
9	3%	1%	6%	35%	25%	85%
10	1%	3%	6%	32%	23%	84%
Mean Avg.	3.1%	4.4%	6.5%	33.9%	25.3%	85.5%
Median	3%	4.5%	6.5%	34%	25.5%	85%
Pop. Variance	3.09	5.84	1.25	5.89	7.81	18.85
Std. Deviation	1.76%	2.42%	1.12%	2.43%	2.8%	4.34%

Table F.14: Idle/Active CPU usage on the Route Detail screen on Android.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	1%	2%	4%	8%	9%	39%
2	1%	2%	4%	8%	10%	37%
3	1%	4%	3%	10%	12%	38%
4	1%	1%	6%	14%	11%	42%
5	3%	1%	3%	11%	12%	39%
6	3%	2%	8%	9%	11%	44%
7	1%	2%	6%	12%	10%	37%
8	2%	1%	5%	11%	9%	40%
9	2%	1%	7%	11%	10%	38%
10	1%	1%	5%	9%	9%	38%
Mean Avg.	1.6%	1.7%	5.1%	10.3%	10.3%	39.2%
Median	1%	1.5%	5%	10.5%	10%	38.5%
Pop. Variance	0.64	0.81	2.49	3.21	1.21	4.56%
Std. Deviation	0.8%	0.9%	1.58%	1.79%	1.1%	2.14%

Memory Usage The following tables were constructed from the results of memory usage measurements. Each table contains test results of one screen on one specific platform. Framework names are abbreviated to RN in the case of React Native and F. in the case of Flutter.

Table F.15: Idle/Active Memory usage on the Login screen on iOS.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	29.1MB	49.9MB	23.1MB	34.3MB	56.2MB	25.3MB
2	28.3MB	49.7MB	23MB	35.1MB	58.1MB	26.1MB
3	29.2MB	50.5MB	22.4MB	35.4MB	57.5MB	26.7MB
4	29.2MB	49.8MB	22.1MB	35.2MB	57.4MB	25.9MB
5	29MB	50.1MB	22.3MB	34.9MB	57.9MB	26.3MB
6	28.4MB	50.3MB	22.8MB	34.9MB	56.7MB	25.5MB
7	29.3MB	51.2MB	22.8MB	34.7MB	56.1MB	25.8MB
8	29.1MB	50.8MB	23.1MB	36.1MB	57.2MB	26.4MB
9	29.1MB	49.6MB	22.6MB	34.6MB	57.7MB	25.1MB
10	29.5MB	50.2MB	22.9MB	35.5MB	56.9MB	26.5MB
Mean Avg.	29.02MB	50.21MB	22.71MB	35.07MB	57.17MB	25.96MB
Median	29.1MB	50.15MB	22.8MB	35MB	57.3MB	26MB
Pop. Variance	0.13	0.23	0.11	0.24	0.42	0.26
Std. Deviation	0.36MB	0.48MB	0.33MB	0.49MB	0.65MB	0.51MB

Table F.16: Idle/Active Memory usage on the Map screen on iOS.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	67.9MB	116.4MB	24.2MB	80.1MB	134.2MB	29.1MB
2	67.5MB	116.1MB	23.9MB	81.3MB	129.4MB	30MB
3	68.1MB	118.8MB	24.5MB	79.7MB	131.9MB	30.3MB
4	68.4MB	115.7MB	23.9MB	80.5MB	132.7MB	29.8MB
5	67.8MB	116.9MB	23.8MB	82MB	132.9MB	29.1MB
6	69.4MB	117MB	24.4MB	81.6MB	133.1MB	29.3MB
7	68.6MB	115.8MB	24.7MB	81.1MB	134.6MB	30.8MB
8	68.9MB	115.9MB	24.1MB	80.8MB	134MB	31.1MB
9	69.9MB	116.3MB	23.9MB	79.4MB	133.3MB	29.9MB
10	68.3MB	117.1MB	23.4MB	81.3MB	129.9MB	30.6MB
Mean Avg.	68.48MB	116.6MB	24.08MB	80.78MB	132.6MB	30MB
Median	68.35MB	116.35MB	24MB	80.95MB	133MB	29.95MB
Pop. Variance	0.5	0.77	0.13	0.64	2.74	0.45
Std. Deviation	0.71MB	0.88MB	0.36MB	0.8MB	1.65MB	0.67MB

Table F.17: Idle/Active Memory usage on the Gallery screen on iOS.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	250.2MB	239.6MB	124.3MB	432.1MB	462.2MB	198.7MB
2	249.9MB	241.1MB	123.7MB	452.7MB	476.1MB	201.2MB
3	250.5MB	240.9MB	123.1MB	369.9MB	501.7MB	197.9MB
4	251.1MB	240.3MB	124.8MB	377.6MB	384MB	198.3MB
5	250.9MB	240.1MB	124.4MB	401.5MB	399.8MB	200.2MB
6	249.7MB	241.5MB	124.9MB	409.9MB	425.6MB	203.5MB
7	250.6MB	239.8MB	123.2MB	441.2MB	498.3MB	201.7MB
8	250.9MB	240.1MB	123.6MB	454.8MB	387.1MB	201.1MB
9	251.3MB	241.3MB	124.1MB	412.3MB	500.9MB	199.9MB
10	249.8MB	240.4MB	124.5MB	398.9MB	494.2MB	199.4MB
Mean Avg.	250.49MB	240.51MB	124.06MB	415.09MB	452.99MB	200.19MB
Median	250.55MB	240.35MB	124.2MB	411.1MB	469.15MB	200.05MB
Pop. Variance	0.29	0.38	0.36	789.16	2171.87	2.68
Std. Deviation	0.54MB	0.62MB	0.6MB	28.09MB	46.6MB	1.64MB

Table F.18: Idle/Active Memory usage on the Route Detail screen on iOS.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	284.1MB	349.8MB	135.7MB	301.4MB	370.2MB	141.6MB
2	284.5MB	351.7MB	135.5MB	303.5MB	372.6MB	143MB
3	283.4MB	347.3MB	135.8MB	309.1MB	369.9MB	142.1MB
4	284.9MB	347.9MB	135.1MB	305.6MB	372.8MB	142.9MB
5	283.8MB	349.1MB	136MB	305.5MB	373.1MB	143.5MB
6	284.3MB	350.9MB	135.1MB	309.9MB	370.9MB	141.9MB
7	285.1MB	352.1MB	135.9MB	302.3MB	368.6MB	141.7MB
8	284.3MB	350.4MB	136.8MB	304.6MB	370.5MB	142.3MB
9	284.6MB	349.8MB	135.2MB	310.2MB	372.8MB	144.1MB
10	284.2MB	348.2MB	136.9MB	303.3MB	372.1MB	143.2MB
Mean Avg.	284.32MB	349.72MB	135.8MB	305.54MB	371.35MB	142.63MB
Median	284.3MB	349.8MB	135.75MB	305.05MB	371.5MB	142.6MB
Pop. Variance	0.22	2.33	0.37	9.11	2.13	0.63
Std. Deviation	0.47MB	1.53MB	0.61MB	3.02MB	1.46MB	0.79MB

Table F.19: Idle/Active Memory usage on the Login screen on Android.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	55.3MB	67.8MB	17.7MB	62.2MB	78.1MB	24.5MB
2	55.9MB	69.3MB	17.1MB	66.3MB	76.2MB	25.3MB
3	56.1MB	69MB	18.1MB	64.1MB	79.3MB	25.9MB
4	55.4MB	68.7MB	17.3MB	62.9MB	77.7MB	24.8MB
5	55.1MB	68.1MB	17.8MB	63.2MB	78MB	26MB
6	56.9MB	67.9MB	16.9MB	64.4MB	79.1MB	25.8MB
7	56.3MB	68.3MB	17.2MB	62.6MB	77.2MB	24.8MB
8	57MB	68.6MB	17MB	62.7MB	76.8MB	24.7MB
9	56.7MB	69.4MB	17.9MB	63.1MB	77.9MB	25.2MB
10	55.8MB	68MB	16.8MB	65.8MB	79.5MB	25.6MB
Mean Avg.	56.05MB	68.51MB	17.38MB	63.73MB	77.98MB	25.26MB
Median	56MB	68.45MB	17.25MB	63.15MB	77.95MB	25.25MB
Pop. Variance	0.41	0.3	0.19	1.75	1.06	0.27
Std. Deviation	0.64MB	0.55MB	0.44MB	1.32MB	1.03MB	0.52MB

Table F.20: Idle/Active Memory usage on the Map screen on Android.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	89.7MB	140.2MB	21.2MB	101.6MB	162.9MB	26.2MB
2	89.6MB	139.8MB	21.5MB	109.9MB	155.2MB	25.9MB
3	90.1MB	143.6MB	21.9MB	110.3MB	159.4MB	25.9MB
4	93.5MB	147.8MB	20.9MB	104.8MB	154.6MB	26.6MB
5	90.9MB	148.1MB	21.8MB	105.5MB	160.1MB	26.3MB
6	94.1MB	139.9MB	22.3MB	103.4MB	155.9MB	26.8MB
7	89.6MB	140.6MB	21.5MB	108.7MB	157.8MB	26MB
8	93.3MB	141.5MB	21.3MB	109.1MB	161.3MB	25.7MB
9	93.7MB	144.4MB	21.2MB	102.1MB	160.4MB	26.9MB
10	90.3MB	145.3MB	21.9MB	105.7MB	156.3MB	26.7MB
Mean Avg.	91.48MB	143.12MB	21.55MB	106.11MB	158.39MB	26.3MB
Median	90.6MB	142.55MB	21.5MB	105.6MB	158.6MB	26.25MB
Pop. Variance	3.31	9.22	0.19	9.36	7.22	0.16
Std. Deviation	1.82MB	3.04MB	0.44MB	3.06MB	2.69MB	0.4MB

Table F.21: Idle/Active Memory usage on the Gallery screen on Android.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	371.5MB	341.3MB	102.6MB	432.1MB	395.2MB	142.9MB
2	374.3MB	343.6MB	103.1MB	424MB	377.2MB	143.5MB
3	374MB	342.2MB	103.9MB	430.9MB	403.9MB	142.6MB
4	372.1MB	343.1MB	102.1MB	398.3MB	423.3MB	143.8MB
5	372.9MB	341.9MB	101.9MB	403.8MB	414.8MB	144.9MB
6	372.2MB	343MB	102.4MB	407.2MB	453.1MB	141.6MB
7	374.1MB	342.5MB	103.2MB	400.1MB	431.7MB	143.1MB
8	373.7MB	342.8MB	101.8MB	431.5MB	399.4MB	143.2MB
9	374.3MB	343.3MB	103.3MB	414.7MB	401MB	144.6MB
10	372.6MB	341.7MB	102.9MB	419.3MB	388.2MB	144.1MB
Mean Avg.	373.17MB	342.54MB	102.72MB	416.19MB	408.78MB	143.43MB
Median	373.3MB	342.65MB	102.75MB	417MB	402.45MB	143.35MB
Pop. Variance	0.97	0.51	0.42	159.01	451.42	0.86
Std. Deviation	0.98MB	0.71MB	0.64MB	12.61MB	21.25MB	0.93MB

Table F.22: Idle/Active Memory usage on the Route Detail screen on Android.

Test No.	RN Idle	F. Idle	PWA Idle	RN Active	F. Active	PWA Active
1	383MB	354.9MB	113.8MB	403.1MB	380.4MB	122.4MB
2	385.1MB	356.2MB	112.9MB	409.5MB	379.1MB	121.7MB
3	379.1MB	355.1MB	112.2MB	407.5MB	377.5MB	123.9MB
4	381.2MB	354.9MB	113.1MB	401.3MB	373.2MB	123MB
5	383.9MB	355.4MB	113.5MB	399.7MB	380.2MB	122.9MB
6	379.9MB	356.1MB	113.9MB	407.9MB	379.7MB	122.5MB
7	381.5MB	355.7MB	112.4MB	406.1MB	374.9MB	123.1MB
8	386.7MB	356.5MB	111.9MB	397.8MB	377.2MB	122.2MB
9	382.5MB	354.3MB	112.7MB	409.2MB	380.6MB	122.8MB
10	384.7MB	355.8MB	113.3MB	400.9MB	377.1MB	121.9MB
Mean Avg.	382.76MB	355.49MB	112.97MB	404.3MB	377.99MB	122.64MB
Median	382.75MB	355.55MB	113MB	404.6MB	378.3MB	122.65MB
Pop. Variance	5.18	0.43	0.41	16.29	5.58	0.37
Std. Deviation	2.28MB	0.66MB	0.64MB	4.04MB	2.36MB	0.61MB

Frames per Second Tables presented in this paragraph were constructed from measurements of Frames per Second. Each table contains test results of one screen on both platforms.

Table F.23: Frames per Second on the Login screen.

Test No.	Android		iOS	
	React Native	Flutter	React Native	Flutter
1	59	60	60	60
2	60	60	60	60
3	60	60	60	60
4	60	59	60	60
5	59	59	60	60
6	60	60	60	60
7	59	60	60	60
8	60	60	60	60
9	58	60	60	60
10	60	59	60	60
Mean Avg.	59.5	59.7	60	60
Median	60	60	60	60
Pop. Variance	0.45	0.21	0	0
Std. Deviation	0.67	0.46	0	0

Table F.24: Frames per Second on the Map screen.

Test No.	Android		iOS	
	React Native	Flutter	React Native	Flutter
1	51	55	60	60
2	53	54	60	60
3	53	55	60	60
4	54	52	60	60
5	50	55	60	60
6	51	56	60	60
7	55	55	60	60
8	52	52	60	60
9	52	53	60	60
10	53	54	60	60
Mean Avg.	52.4	54.1	60	60
Median	52.5	54.5	60	60
Pop. Variance	2.04	1.69	0	0
Std. Deviation	1.43	1.3	0	0

Table F.25: Frames per Second on the Gallery screen.

Test No.	Android		iOS	
	React Native	Flutter	React Native	Flutter
1	37	36	60	60
2	38	37	60	59
3	36	34	60	60
4	33	34	59	60
5	38	38	60	60
6	33	37	60	59
7	37	33	58	59
8	36	33	60	60
9	37	35	60	60
10	38	36	59	60
Mean Avg.	36.3	35.3	59.6	59.7
Median	37	35.5	60	60
Pop. Variance	3.21	2.81	0.44	0.21
Std. Deviation	1.79	1.68	0.66	0.46

Table F.26: Frames per Second on the Route Detail screen.

Test No.	Android		iOS	
	React Native	Flutter	React Native	Flutter
1	53	39	60	60
2	55	39	60	60
3	51	40	60	60
4	52	37	60	60
5	53	41	60	60
6	55	39	60	60
7	52	40	60	60
8	52	36	60	60
9	53	37	59	60
10	51	36	60	60
Mean Avg.	52.7	38.4	59.9	60
Median	52.5	39	60	60
Pop. Variance	1.81	2.84	0.08	0
Std. Deviation	1.35	1.69	0.29	0

Build Times Following tables contain results of build times measurements.

Table F.27: Comparison of iOS build times.

Test No.	Clean Build		Cached Build	
	React Native	Flutter	React Native	Flutter
1	390.3s	91.3s	37.9s	5.6s
2	404s	86.8s	35.6s	4.8s
3	372.2s	130.4s	36.2s	5s
4	359.4s	101.1s	38s	5.3s
5	377.4s	84s	35.5s	4.8s
6	388s	95.7s	37.9s	5.5s
7	369.2s	79.8s	36s	5.2s
8	371.3s	92.7s	35.7s	5.6s
9	390s	82.8s	35.4s	5.8s
10	368.2s	102.3s	35.9s	8.2s
Mean Avg.	379s	94.69s	36.42s	5.58s
Median	374.8s	92s	35.95s	5.4s
Pop. Variance	165.96	193.03	1.05	0.87
Std. Deviation	12.88s	13.89s	1.02s	0.93s

Table F.28: Comparison of Android build times.

Test No.	Clean Build		Cached Build	
	React Native	Flutter	React Native	Flutter
1	56.1s	33s	11.1s	3.8s
2	52s	30.4s	11.3s	3.6s
3	53.3s	27.7s	10.4s	4s
4	52.9s	31.1s	10.5s	3s
5	52.8s	25.5s	9.8s	4.6s
6	65.6s	36.4s	9.8s	3.9s
7	65.2s	24.4s	9.9s	3.3s
8	64.4s	29s	9.4s	3.2s
9	56.6s	25.5s	9.5s	3s
10	53.9s	29.1s	10.1s	3.4s
Mean Avg.	57.48s	29.21s	10.18s	3.58s
Median	55.05s	29.05s	10s	3.5s
Pop. Variance	26.14	12.41	0.37	0.23
Std. Deviation	5.11s	3.52s	0.61s	0.48s

Table F.29: Comparison of clean and cached web build times.

Build No.	Clean Web Build	Cached Web Build
1	35.1s	1.4s
2	36s	0.7s
3	40.4s	0.6s
4	37s	0.9s
5	40.9s	0.6s
6	37.1s	1.1s
7	35.7s	0.7s
8	34.2s	0.7s
9	34.4s	0.6s
10	34.6s	0.8s
Mean Avg.	36.54s	0.81s
Median	35.85s	0.7s
Pop. Variance	5.13	0.06
Std. Deviation	2.27s	0.25s