Cross-platform app development with ahead-of-time compilation

David Speers

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2019

Abstract

Currently, 99.9% of mobile apps run on either Android or iOS. The problem is that Android apps are developed natively with the Android SDK and Kotlin/Java and iOS apps are developed natively with the iOS SDK and Swift/Objective-C. Therefore, if app developers want to release their apps for both platforms, they will need to develop the same app twice.

One solution to this problem is to use cross-platform SDKs which allow code to be written once and released on both platforms. Historically these SDKs have had limitations and problems which kept them from being commonly used. Flutter is a cross-compiled SDK supported by Google that claims to fix many of these problems.

In this paper, these claims were evaluated by developing the same app twice; once with Flutter written in Dart, and once with the Android SDK written in Kotlin. Ease-of-development and performance of the two apps were also compared. The results show that Flutter is a valid choice for Android app development, but that it still has some limitations.

Acknowledgements

I want to thank my supervisors; Dr Ajitha Rajan, Dr Stephen Gilmore, and Dr Hugh Leather for their guidance, help and support throughout this project.

I would also like to thank my family for continuously supporting me in the pursuit of my academic goals.

Table of Contents

Chapter 1

Introduction

1.1 Project Motivation

According to Gartner [1], in 2017 Android had 85.9% of global smartphone market share while Apple had 14.0%. This number is reflected by the Google Play stores' 75.7 Billion App Downloads vs the Apple App stores' 29.6 Billion in 2018 [2]. However, app revenue for the Google Play Store was only $24.8 Billion compared to Apple's $46.6 Billion in 2018 [2] - therefore, if mobile app developers want to reach as many users as possible and maximise revenue, they need to utilise both platforms. However, as Android apps are created natively using Java/Kotlin and the Android SDK and iOS apps are created natively using Objective-C/Swift and the iOS SDK, developers have been looking for ways to write code once and release it on both platforms (known as cross-platform app development).

Historically cross platforms SDKs have had their limitations, including requiring developers to program the UI for iOS and Android separately (only sharing backend code) [3], only supporting basic app development features [4], and having legacy problems such as bad performance or security (often due to being written in HTML, CSS, and JavaScript) [5].

Flutter [6] is a cross-compiled SDK supported by Google that claims to allow for Ahead-of-Time compilation, native performance and increased security and is written in Dart. As a result, Flutter should avoid many of the issues that have haunted other cross-platform SDKs.

The first stable version of Flutter was released on December 4th, 2018 [7] and as Flutter is relatively new, developers do not yet know if they can trust Flutter to deliver on all of the things they require, especially in light of the history of problematic cross-platform SDKs. In this project, Flutter's claims will be investigated by comparing Native Android development using the Android SDK and Kotlin with cross-compilation development using Flutter and Dart.

1.2 Project Goals

The goal of this project is to test the effectiveness of the Flutter SDK by developing an app twice; once with Flutter and once with the Android SDK. The two implementations of the same app will then be compared for ease-of-development and performance.

1.3 Contributions

- Identified key features that a cross-platform app must have for Android development. These key features are Activities and UI, Navigation and Intents, Animations and Transitions, Web-based Content, Touch and Input, Audio and Video, Background Tasks, App Data and Files, Connectivity, Google Play Instant, User Data and Identity, User Location, Camera, Sensors, and Renderscript.
- Designed, developed, and tested an app that implements most of these key features, both natively with the Android SDK and cross-compiled with Flutter.
- Evaluated how the development process between the SDKs differed.
- Measured and evaluated the performance of these two apps.
- Measured and evaluated the performance of other simple Native and Flutter apps.
- Measured and evaluated the performance of Rosetta Code algorithms when compiled Ahead-of-Time on an Android device.

1.4 Report Structure

1. **Background**
   Provides information on how the Android SDK and Flutter compile and run on Android devices. Also, analyses the core concepts of cross-platform app research and examines existing research.
2. **Methodology**
   Identifies the core features of Android apps and provides an overview of the app that was developed to test these features.
3. **SDK Comparison**
   Evaluates how the two SDKs differed with the features implemented in the app.
4. **Performance Evaluation**
   Investigates the performance of the two SDKs by examining the startup and compilation time, APK size, and CPU and memory usage of multiple apps. Also, explores how fast the two SDKs compute complex algorithms.
5. **Conclusion**
   Summarises the findings and identifies possible areas for future research.

Chapter 2

Background

In this chapter, how the Android SDK and Flutter compile apps to Android will be explained. In addition, existing research on cross-platform app development will be examined.

2.1 Terminology

In other papers about cross-platform development [8] [9] [10], developing Android apps with the Android SDK and iOS apps using the iOS SDK is referred to as the *Native* approach. Therefore, in this paper we use the same terminology, referring to the Android SDK with Kotlin as the *Native* approach, and Flutter with Dart as the *Flutter* approach.

2.2 Native Android Development

The three highest layers of the Android Platform Stack are (in descending order):

- Android Apps.
- The Java API Framework.
- Android Runtime (ART).

ART
Each Android app runs an instance of ART. ART is Ahead-of-Time (AOT) compiled, which means that when an app is installed, it translates bytecode into machine code and stores it on the device (as an APK file). This allows apps to run far faster than when Just-in-Time (JIT) compiled.

Java API Framework and Android Apps
The Java API Framework is a modular set of system components and services that Android Apps use. Two of the most important are:

- The Activities Manager, which allows developers to create a window in which an app draws its UI.
- The View System, which allows developers to draw UI components into activities.

When developers create Native Android apps, they use the Android SDK which is a comprehensive set of development tools written in Java/Kotlin that allow developers to build, run, and debug Android apps. These tools invoke the Java API Framework.

2.3 Flutter for Android Development

Flutter consists of three layers:

- The Flutter Framework, written in Dart.
- The Flutter Engine, written in C/C++.
- The Flutter Embedder, written in Java/Kotlin (Android) or Objective-C/Swift (iOS).

All of these layers occur within the *Android Apps* layer from the previous section.

Embedder
The Embedder is the bottom layer and provides a shell that hosts the Dart VM. The shell is platform specific, giving access to the native platform APIs and hosting the platform relevant UI canvas. This means that when compiling Flutter to Android, the app will run an Engine and a Framework on top of a single Activity, which can use any Android SDK API.

If the developer wants to call an API from the Android SDK that is not already defined by Flutter or one of its plugins, they need to alter the Embedder. In this paper this was required in two sections (Sections 4.4 and 5.3).

Engine
The Engine acts as a mediator between the Framework and the Embedder. The Framework written in Dart can call functions and send primitives to the Embedder using the Engine's *Platform Channels*. The Engine is also where the Dart Runtime and Skia [11] are defined. Skia is a graphics engine and allows the Framework to draw its UI.

If the developer wants to alter the Dart Runtime, how the UI components are rendered, or how calls are sent to the Embedder, they need to alter the Engine. Throughout the study, this was not required.

Framework
The Framework is the main area of focus in this study and most Flutter projects. This is where the developer writes their UI and any Dart logic in their Flutter app.

Upon installation of a Flutter app, not only is the Android shell (Embedder) compiled AOT with ART, but the Dart code also compiles AOT and this is one of the main reasons why Flutter claims they have managed to achieve Native app performance.

A summary of Flutter's stack is shown in Figure 2.1.

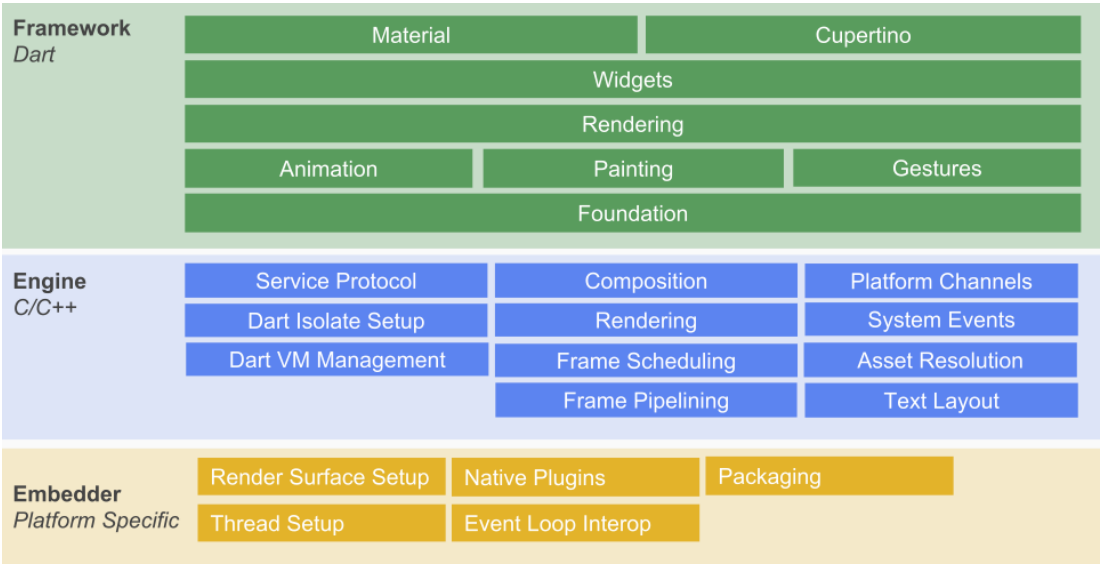


Figure 2.1: Flutter System Overview [12]

2.4 Literature Review

2.4.1 Core Concepts of Cross-Platform Research

Biørn-Hansen et al. [10] published a paper called *A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development*. In their paper they assess the current state of cross-platform mobile development research. The paper, which was published in July 2018, identifies Flutter as a possible SDK for future research. The paper identified the following as core concepts when it comes to the research of cross-platform apps:

- User Experience
- Software Platform Features
- Performance and Hardware
- Security

This paper focuses on two of these: Software Platform Features (Chapter 4), and Performance and Hardware (Chapter 5).

2.4.2 Other Comparative Mobile Studies

As Flutter is relatively new little research is available on it, especially research which compares it to other mobile SDKs. Nevertheless, three interesting mobile SDK com-

parison papers were found, one of which also compared Flutter.

Swift vs F#
Staszak [13] developed a Tinder-esque Tenant-Landlord matching app, comparing iOS development with Swift against Xamarin with F#.

This paper examined Software Platform Features and User Experience, however, it did not investigate Performance. Although this paper did not justify why the application used was a good choice for comparing the SDKs it is still an excellent example of a paper which analyses two SDKs.

React Native vs Flutter, cross-platform mobile application frameworks
Wu [14] compared Flutter and React Native. This paper compared Software Platform Features and Performance.

While the paper provided a good entry point for comparing the two platforms it only used a very simple app that displayed movie and television information. When measuring performance, Wu measured frames per second and I/O writing. While these metrics are interesting it is more common to use CPU or Memory usage as metrics when testing performance [15] [16].

Comprehensive Analysis of Innovative Cross-Platform App Development Frameworks
Majchrzak et al. [17] compared React Native, Ionic, and Fuse by using a mock app that was designed to test the SDKs' abilities to access Camera, GPS, Image Gallery, and Contacts and compare some basic UI components. The purpose of their paper was only to compare Software Platform Features.

The paper investigated whether each platform can implement a feature and how difficult it was to implement. As Flutter's *Platform Channel* can make any Native API calls (see Section 2.3) it is not relevant for us to check whether Flutter can implement a feature or not. However, this does not mean that implementing these features is easy because some *Platform Channel* calls can get very complicated; for example, when dealing with Native UI components. Majchrzak et al. [17] evaluated whether implementing each UI feature was Simple, Intermediate, or Complex - a similar evaluation criteria was used in the SDK features comparison of this study.

Chapter 3

Methodology

To compare the two SDKs, the same app was developed twice; once Natively and once with Flutter. The first step was to identify what constitutes a good app. Although the literature review did not identify any consensus on this, the Official Android Developer Documentation [18] identifies key features that the Android SDK offers. These key features are split into two sections; “core developer topics”, and “best practices”. Having looked at these features, an app was designed to test as many of these features as possible. This resulted in a quiz app where users guess words that are googled together.

3.1 Core Developer Topics

In the documentation the following were listed as core topics:

- Activities
- Intents
- UI and Navigation
- Animations and Transitions
- Images and Graphics
- Audio and Video
- Background Tasks
- App Data and Files
- Touch and Input
- Connectivity
- Web-based Content
- Instant Apps

- User Data and Identity
- User Location
- Camera
- Sensors
- Renderscript

In the app, all the features were implemented, with the exception of the last five. This is because it is difficult to design an app that will implement every core topic and due to the time constraints in a UG4 Honours Project it was not feasible to develop an additional app to test the last five features. It is also worth noting that in existing literature [13] [14] these last five features were also not tested. Nevertheless, I believe the app covers enough features to provide a reliable comparison between the two SDKs.

Some of these topics are closely related, and so were implemented together. Therefore when comparing the SDKs in Chapter 4 changes were made to the list as follows:

- Activities and UI were grouped together.
- Navigation and Intents were grouped together.
- Images and Graphics were explained as part of the Activities and UI section.

3.2 Best Practices

In the documentation, the following are listed as best practices: Testing, Performance, Accessibility, Security, Build for Billions, Build for Enterprise, and Google Play. In this study, only testing and performance were explored.

Testing consisted of:

- Unit Tests
- Integration Tests
- UI Tests

Performance explored:

- APK Size
- CPU Usage
- Memory Usage
- Compilation Time
- Startup time

3.3 Potential Biases in the Study

The following were potential biases in the study:

- The app was first developed in Flutter, followed by the Native implementation.
- Prior to the study I had a year of experience using the Android SDK and Kotlin and no experience with Flutter.

However, when evaluating and comparing the two SDKs I tried to keep this in mind and do not believe it affected the outcome of the study.

3.4 App Overview

The app that was developed is called Trends with Friends (referred to as the Trends app in the remainder of this study). The Trends app is a game where users compete with friends or the computer to guess commonly searched queries on Google. To do this, it uses Google Trends [19]. As there is no official Google Trends API, a Node.js server was implemented which uses an unofficial Google Trends API [20]. The app sends the user's answers to the server as a JSON object in an HTTP POST Request. It then waits for the server to reply with its JSON object as an HTTP POST Response.

3.5 App Structure

The app's structure can be seen in Figure 3.1.

MainActivity
Upon startup, the user is greeted by the MainActivity. From here they can go to one of the following Activities:

- AboutActivity
- AchievementsActivity
- ThemesActivity. To get to the ThemesActivity, the user can click on the play button with 'Party Mode' displayed over it to play against human opponents or the play button with 'CPU Mode' displayed over it to play against the computer. To select the number of players or CPU's difficulties the plus and minus buttons can be tapped.

AboutActivity
Provides a tutorial of the app and how to play the game, both as a video and text.

AchievementsActivity
Displays the user's locked and unlocked achievements.

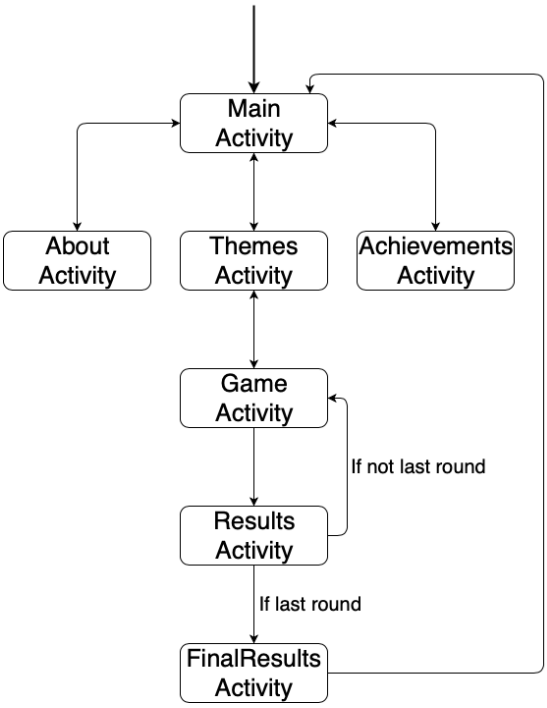


Figure 3.1: App Structure

ThemesActivity
In the ThemesActivity the user selects the theme they want to play. The theme they choose determines the terms that they will need to match in the GameActivity.

GameActivity
In the GameActivity the user is given a term. They must then match the term with a word which will combine with the term to become their answer for that round. For example, if given the word 'Super' and the user typed 'Bowl' then their answer becomes 'Super Bowl'. After typing their answer, they tap the floating action button to submit their answer. If the user is the last to give their answer, then the app continues to the ResultsActivity. Otherwise, it clears the text box and the next player types and submits their answer.

ResultsActivity
The ResultsActivity is split between two tabs. The first tab displays the results of the users' answers' and their scores. The second tab displays a graph which shows the scores over the last year. This graph is important because the user's scores will change over time - for example, if the user answers 'Super Bowl' in February they will likely have a higher score than if they answer that in July because Google's search traffic for the term 'Super Bowl' increases significantly around February.

Once the user presses either the back button in the ResultsActivity or the next button they will either:

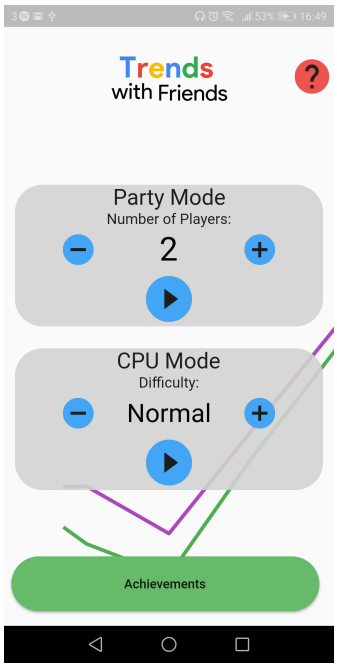
- Go to the FinalResultsActivity (if it is the last round of the game).
- Go back to the GameActivity where a new term is displayed that they must match

with (if it is not the last round of the game).

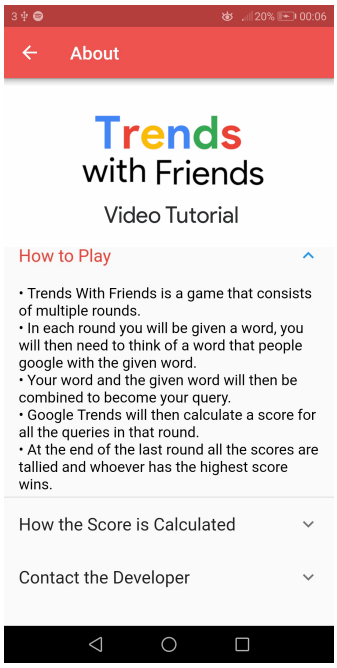
FinalResultsActivity
The FinalResultsActivity declares the winner of the game. Once the user presses the back or next button, they return to the MainActivity.

3.6 App Screenshots

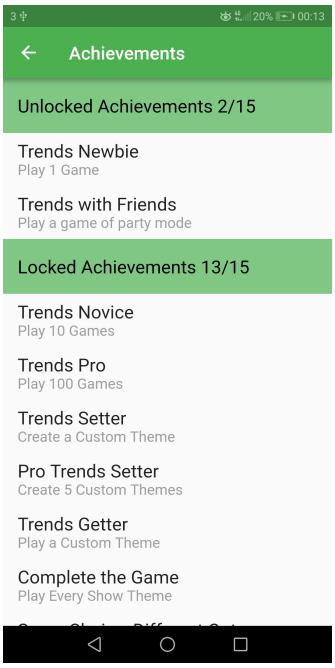
This section contains the screenshots of the Activities of the Flutter implementation. The Activities of the Native implementation look very similar and can be in found in Appendix A.



a) MainActivity

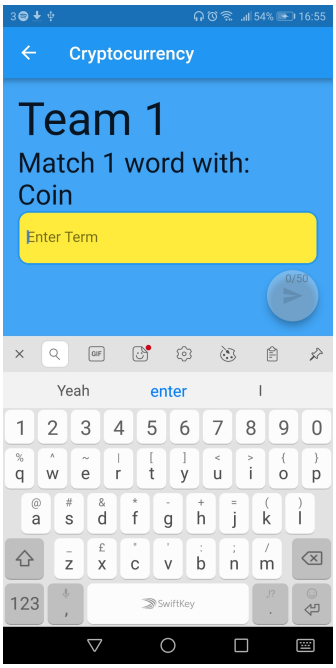


b) AboutActivity

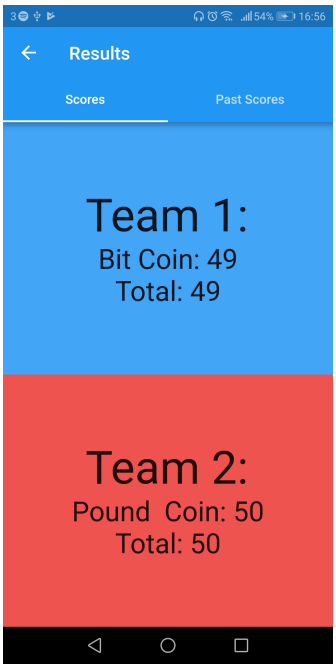


c) AchievementsActivity

d) ThemesActivity



e) GameActivity



f) ResultsActivity - Tab 1

g) ResultsActivity - Tab 2

h) FinalResultsActivity

Chapter 4

SDK Comparison

This chapter compares both SDKs by comparing the implementations of the topics described in Chapter 3 (Activities and UI, Navigation and Intents, Animations and Transitions, Web-based Content, Touch and Input, Audio and Video, Background Tasks, App Data and Files, Connectivity, Google Play Instant, and Testing). Each section begins by describing the feature and its importance within the app. It also describes how that feature was implemented using Native and Flutter, and evaluates the difficulty level of both implementations, including comments on issues encountered and how they were resolved. This chapter concludes with a brief look at the popularity of the SDKs and their languages and an overview of the strengths and weaknesses of both SDKs.

4.1 Activities and UI

According to the Android Documentation [18] *"An activity provides the window in which the app draws its UI. This window typically fills the screen, but may be smaller than the screen and float on top of other windows. Generally, one activity implements one screen in an app."* Therefore, in order to draw the app's UI we will need to use multiple Activities and navigate between them. Each Activity draws UI components such as Buttons, Images, TextViews etc.

4.1.1 Terminology

Before looking at specific examples we need to define some key differences and definitions.

In Native an Activity is typically split between two files:

- An XML file which defines the Layouts and Views. A Layout defines the structure of its children's Views and is a type of ViewGroup. Figure 4.1 illustrates a general view hierarchy.

- A Kotlin file which references the XML file and defines the logic of that Activity. It is worth noting that it is possible to create Views programmatically within the Kotlin file and, if desired, the entire activity can be defined within this file but this is typically not done unless Views need to be created dynamically.

In Flutter the UI is drawn completely with Dart. Instead of an Activity with a View Hierarchy every UI element in Flutter is a Widget (Note: it is a naming convention to call Widgets that act as Activities "Pages"). Each Widget can either have a child, which allows it to be the parent of one Widget, or multiple children which allows it to be the parent of a list of Widgets. Therefore, although in Flutter there are no Activities, Layouts, or Views there are Widget equivalents to these Native concepts and so these concepts will be referred to by their Native name when referring to the general term.

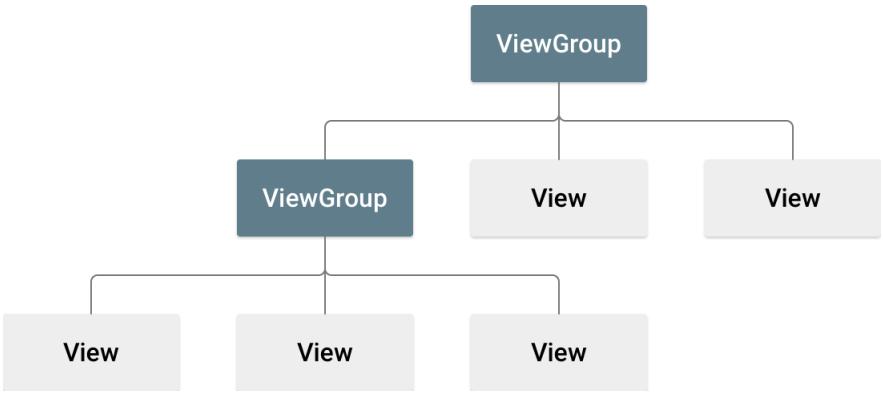


Figure 4.1: View Hierarchy [21]

4.1.2 Basic Activity

I considered a basic activity to be one that follows the typical View Hierarchy structure where the entire UI is made using XML without requiring any Kotlin code. An example of this is the app's MainActivity seen in Figure 4.2 whose Hierarchy can be seen in Figure 4.3. In this Activity we have the Trends with Friends logo as an image, several Buttons, TextViews, and ImageViews, and an animated background (more on that in Section 4.3). To reduce the amount of code and make comparing the two SDKS easier for this section I will only include the Trends logo and Achievements button whose shape is defined by a Drawable.

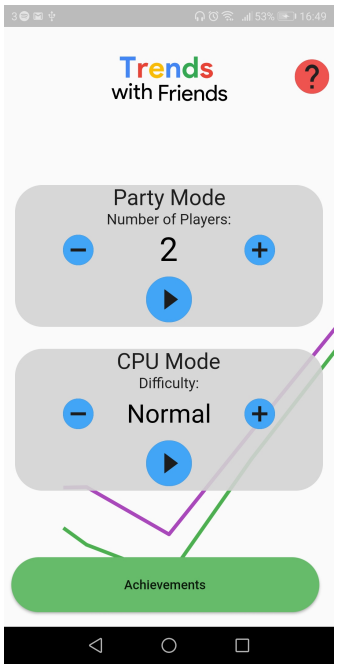


Figure 4.2: MainActivity

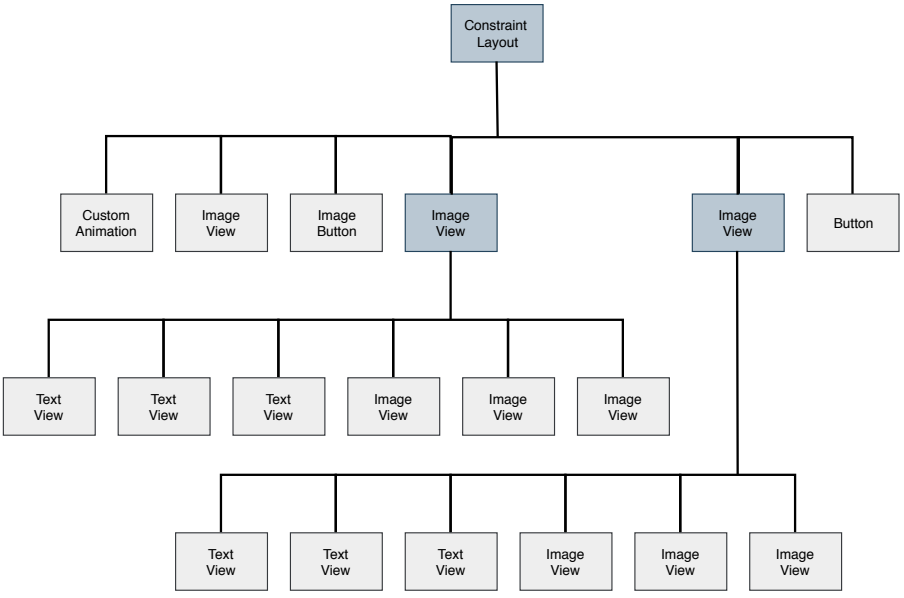


Figure 4.3: MainActivity Hierarchy

Native

XML

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">

    <ImageView
            android:layout_width="192dp"
            android:layout_height="70dp"
            app:srcCompat="@drawable/trends_logo"
            android:id="@+id/trendsLogo"/>

    <Button
        android:layout_width="match_parent"
        android:layout_height="70dp"
        android:background="@drawable/btn_green"
        android:text="Achievements"
        android:id="@+id/achieveBtn"
        android:scaleType="fitCenter"
    />
</LinearLayout>
```

XML

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android" >
    <solid
            android:color="#32CD32"/>
    <corners
            android:radius="20dp"/>
    <stroke
            android:width="1dp"
            android:color="#aeaeae"/>
</shape>
```

To reference this XML file the Kotlin file calls `setContentView(R.layout.activity_main)` and to reference the Achievements button within the activity it calls `findViewById(R.id.achieveBtn)` where the id was defined inside the XML.

Flutter

As mentioned previously, in Flutter every UI component can be defined as a Widget within a Dart file.

Dart

```
@override
Widget build(BuildContext context) {
return new Column(
    mainAxisAlignment: MainAxisAlignment.start,
    children: [
        Image.asset("assets/images/TrendsWithFriendsLogo.png",
           height: 50.0),
        ButtonTheme(
          height: 60.0,
          child: RaisedButton(
            onPressed: () => _pushSelected(text),
            color: bottomButtonsColor,
            textColor: secondaryButtonColor,
            child: new Text("Achievements", style: customFont(color:
                Colors.black87, fontsize: 7.0)),
            shape: RoundedRectangleBorder(borderRadius: BorderRadius
                .all(Radius.circular(30.0)))
        )
      )
    ]
);
```

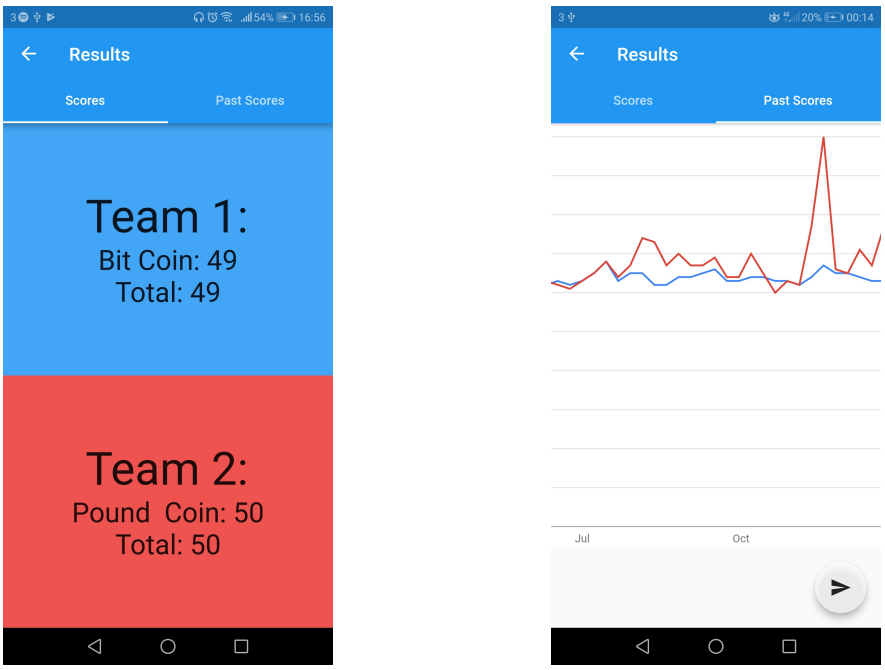
Evaluation
Both SDKs have their merits here. The biggest benefit of XML is that the graphical interface can do a lot of the work for the developer and allows them to visualise how the UI will look like without having to compile to an Android device. However, I found that Flutter is more readable because it's not a markup language and it is slightly more concise. It is worth noting that some developers, particularly those with a background in web development, appreciate the separation of writing the layout with a markup language and writing the background logic with a programming language. Nevertheless, both approaches are intuitive and readable.

	Difficulty Level
Native	Easy
Flutter	Easy

Table 4.1: Basic Activity Comparison

4.1.3 Tabbed Activity

Tabbed activities are commonly used to split between multiple fragments, allowing users to swipe between them, while remaining on the activity. An example is the app's ResultsActivity seen in Figure 4.4. In the code snippets below, the fragments have been abstracted out to only show the activity code.



a) Tab 1 b) Tab 2

Figure 4.4: ResultsActivity

Native

Kotlin

```
class ResultsActivity : AppCompatActivity() {

    private var mySectionsPagerAdapter: SectionsPagerAdapter? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_results)

        mySectionsPagerAdapter = SectionsPagerAdapter(
            supportFragmentManager)

        container.adapter = mySectionsPagerAdapter

        container.addOnPageChangeListener(TabLayout.
            TabLayoutOnPageChangeListener(tabs))
        tabs.addOnTabSelectedListener(TabLayout.
            ViewPagerOnTabSelectedListener(container))
    }

    inner class SectionsPagerAdapter(fm: FragmentManager) :
        FragmentPagerAdapter(fm) {

        override fun getItem(position: Int): Fragment {
            return when (position) {
                0 -> ScoresFragment.newInstance(intent.
                    getStringExtra("query"), intent.
                    getStringArrayExtra("terms"))
```

```
                1 -> TrendsGraphFragment.newInstance(intent.
                    getStringExtra("query"), intent.
                    getStringArrayExtra("terms"))
                else -> throw Exception("Number of Pages out of
                    range")
            }
        }

        override fun getCount(): Int {
            return 2
        }
    }
}
```

Flutter

Dart

```
Scaffold(
    appBar: AppBar(
        bottom: TabBar(tabs: myTabs),
    ),
    body: new Builder(builder: (BuildContext context) {
        _scaffoldContext = context;
        return new TabBarView(children: myTabContents);
    }),
)
```

Evaluation
The Native implementation required a lot of boilerplate code and was more difficult to implement than Flutter.

	Difficulty Level
Native	Intermediate
Flutter	Easy

Table 4.2: Tabbed Activity Comparison

4.1.4 Updating the UI

In Native if a UI component changes it will automatically update the Android UI. For example, when the user presses the + button on the MainActivity the TextView's text is updated and the Android UI text will also change. However, this is not the case when it comes to Flutter. In Flutter, the `setState` method is used, once this method is invoked all Stateful Widgets will be redrawn and any UI changes will appear on the Android UI. (In Flutter all Widgets can either be Stateful or Stateless - Stateless Widgets won't change state and so can't change their UI. This is done so that when `setState` is called not all Widgets are redrawn as this affects the app performance).

4.1.5 App Lifecycle

When creating an Android app it is important to understand the lifecycle of Android Activities. When an Activity is created it calls the `onCreate` method, followed by `onStart` and `onResume`. `onPause` is called when the Activity loses focus (i.e. it is partially visible) and `onStop` is called when the Activity is hidden, either because another Activity has been called over it or when the Activity being terminated. This is summarised in Figure 4.5.

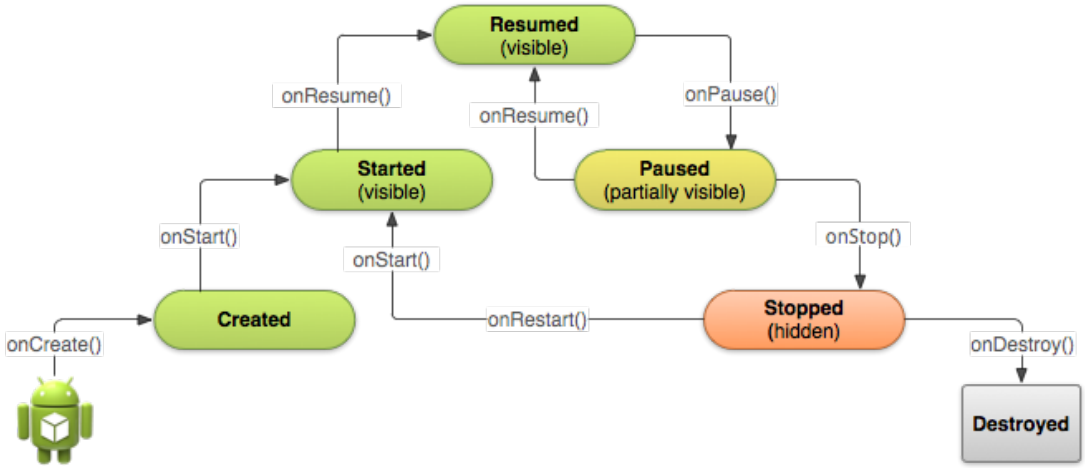


Figure 4.5: Android Activity Lifecycle [22]

In the Trends app this was required to stop the animation in the MainActivity from running in the background when it is hidden - this was needed to reduce memory usage and so the animation is stopped inside the `onPause` method and then resumed in the `onResume` method.

Native

In Native to call this method simply override the `onPause` and `onResume` methods inside the Activity.

Kotlin

```
class MainActivity : AppCompatActivity() {
    override fun onPause() {
        super.onPause()
        animationView.stop()
    }

    override fun onResume() {
        super.onResume()
        animationView.start()
    }
}
```

Flutter

With Flutter this process is more involved. The equivalent of `onCreate` is `initState`, `onStart` is `build`, and `onDestroy` is `dispose`. However, if you want to override `onResume`, `onPause`, or `onStop` you need extend your class with `WidgetsBindingObserver`, add the observer in your `initState` method, remove the observer in your `dispose` method, and override the `didChangeAppLifecycleState` method.

Dart

```
class AnimatedHomeState extends State<AnimatedHome> with
    WidgetsBindingObserver {
    @override
    void initState() {
        super.initState();
        WidgetsBinding.instance.addObserver(this);
    }

    @override
    void dispose() {
        WidgetsBinding.instance.removeObserver(this);
        super.dispose();
    }

    @override
    void didChangeAppLifecycleState(AppLifecycleState state) {
        if (state == AppLifecycleState.inactive) {
            _stopAnimation();
        } else {
            _startAnimation();
        }
    }
}
```

Evaluation

While the process is slightly more involved in Flutter and requires more code it is still easy to implement for both SDKs.

	Difficulty Level
Native	Easy
Flutter	Easy

Table 4.3: App Lifecycle Comparison

4.2 Navigation and Intents

Navigation is the process of changing to a different Activity. However, data often needs to be transferred between activities, for example, the selected theme needs to be transferred from the ThemesActivity to the GameActivity.

Native

With Native this is done using intents.

Kotlin

```
val intent = Intent(this, GameActivity::class.java)
intent.putExtra("theme", theme)
this.startActivity(intent)
```

To call the intents from the GameActivity use `intent.getStringExtra("theme")`.

Flutter

In Flutter `Navigator.of(context).push(<Page>)` is used, where the Page Widget usually takes some parameters as arguments. In this example, GamePage is a custom Widget that takes theme as optional arguments.

Dart

```
Navigator.of(context).push(new GamePage(theme: theme));
```

We can then call the arguments within our GamePage with `widget.theme`.

Evaluation
With both implementations navigating to a new activity is easy.

	Difficulty Level
Native	Easy
Flutter	Easy

Table 4.4: Navigation and Intents Comparison

4.3 Animations and Transitions

In the Trends app a custom animation was used. The MainActivity has a custom background animation where 2 lines of random colour move from the left of the screen to the right, changing direction twice. The lines then erase from left to right. There is also a custom transition. Instead of using the default transition of loading a new activity the app loads the new activities by sliding them in from right to left, recreating the default transition of iOS apps.

4.3.1 Custom Animation

In Native the custom animation was implemented by creating a custom View that overrides the onDraw method. The `path` variable is a randomised Path object, and `paint` is a random Paint object.

Native

Kotlin

```
public override fun onDraw(c: Canvas) {
    super.onDraw(c)
    c.drawPath(path, paint)
    c.drawPath(path, paintEraser)
}
```

This draws the path the lines will take. However, to animate the lines an AnimatorSet was used, with a seperate ObjectAnimator per line.

Kotlin

```
val animations = AnimatorSet()

val animatorDraw = ObjectAnimator.ofFloat(this@PathView, "draw", 1f,
    0f) //"draw" calls the setDraw function
animatorDraw.duration = 4000

val animatorErase = ObjectAnimator.ofFloat(this@PathView, "erase", 1
   f, 0f)
animatorErase.duration = 4000

animations.play(animatorErase).after(animatorDraw)
animations.start()

fun setDraw(phase: Float) {
    paints[0].pathEffect = createPathEffect(lengths[0], phase, 0.0f)
    invalidate()//will call onDraw
}

fun setErase(phase: Float) {
    paintErase.alpha = 255
    paintErase.pathEffect = createPathEffect(lengths.max()!!, phase,
        0.0f)
    invalidate()//will call onDraw
}

fun createPathEffect(pathLength: Float, phase: Float, offset: Float)
   : PathEffect {
    return DashPathEffect(
        floatArrayOf(pathLength, pathLength),
        Math.max(phase * pathLength, offset)
    )
}
```

To then loop the animation once it finishes a listener was added to the `animations` object.

Kotlin

```
animations.addListener(object : Animator.AnimatorListener {
override fun onAnimationEnd(animation: Animator) {
    //*Rerandomise path and paints object
    animations.start()
}
//Following Overrides Required Even When unchanged
override fun onAnimationStart(animation: Animator) {}
override fun onAnimationCancel(animation: Animator) {}
override fun onAnimationRepeat(animation: Animator) {}
})
```

Flutter

In Flutter the custom animation was implemented using a custom CustomPainter that overrides the paint method and animates using an AnimationController. In the AnimationController the duration of the animation is defined and the controller counts from 0 to 1 over that duration. For example, when animating a line you define the length of it to be *lengthOfLine*animationController.value*. While Flutter does have a drawPath method no way of animating it was found and so I wrote my own animatePath method that can be found in Appendix B. It animates the Path by using the drawLine method and the animation controller.

Dart

```
AnimationController _controller = new AnimationController(
  duration: Duration(seconds: 8),
  vsync: this,
);
```

Dart

```
new CustomPaint(
  painter: new LinesPainter(_controller),
  child: _homePageUI(),
);
```

Dart

```
class LinesPainter extends CustomPainter {
  final Animation<double> _animation;

  LinesPainter(this._animation) : super(repaint: _animation);

  void animateXLines(Canvas canvas, Color lineColor, List<Offset>
      points) {
    //found in Appendix B
  }

  @override
  void paint(Canvas canvas, Size size) {
```

```
    animatePath(canvas, points, paint));
  }

  @override
  bool shouldRepaint(LinesPainter oldDelegate) {
    return true;
  }
```

Evaluation
To animate lines in both SDKs is quite difficult, because it requires learning and using a lot of technical UI components and methods. I found that the Native implementation was easier in this case because I could animate the `drawPath` method.

	Difficulty Level
Native	Intermediate
Flutter	Hard

Table 4.5: Custom Animations Comparison

4.3.2 Transitions

Native

In Kotlin to implement the sliding transition the transitions first need to be defined in XML. There are four in total: two when starting a new Activity and two when finishing that Activity. The XML files can be seen in Appendix C. Once defined, when starting the new Activity simply call `overridePendingTransition(R.anim.slide_in_left, R.anim.slide_out_left)` and when finishing that Activity call `overridePendingTransition(R.anim.slide_out_right, R.anim.slide_in_right)`.

Flutter

In Flutter this animation is predefined in the Flutter library, as are a lot of other iOS UI components which exist to easily allow Flutter apps to have the same style as Native iOS apps. To create an iOS-styled Activity transition from the MainActivity to the ThemesActivity first define the CupertinoRoute:

Dart

```
class ThemesPageRoute extends CupertinoRoute {
    ThemePage() : super(builder: (BuildContext context) => new
        ThemePage());
}
```

Then call `Navigator.of(context).push(new ThemePageRoute());`.

Evaluation
Both implementations are easy to implement but I found the Flutter implementation to be less tedious.

	Difficulty Level
Native	Easy
Flutter	Easy

Table 4.6: Transitions Comparison

4.4 Web-based Content

Often apps need to access web-based content such as a website or an HTML file. In the Trends app to display trends over time a graph was used. To do this the Google Charts API [23] was used. However, as the API only works with an HTML file a WebView was needed to display the content. Because the chart displays data that is obtained in either Kotlin or Dart code a channel was needed to communicate between Kotlin or Dart and JavaScript so that the JavaScript in the HTML file could be used. The HTML file can be found in Appendix D. In the file there are methods which call `Android` such as `Android.getQuery()` - as a WebAppInterface class was used with a `@JavascriptInterface` these commands could interact with Kotlin when added to the WebView with `trendsWebView.addJavascriptInterface(WebAppInterface(), "Android")`.

Native

Kotlin

```
class TrendsGraphFragment : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState

        val trendsWebView = rootView.findViewById<WebView>(R.id.
            TrendsWebView)

        trendsWebView.addJavascriptInterface(WebAppInterface(), "
            Android")
        trendsWebView.settings.javaScriptEnabled = true
        trendsWebView.loadUrl("file:///android_asset/graph.html")
    }

    inner class WebAppInterface {
        @JavascriptInterface
        fun getQuery() : String {
            Log.d("Query", query)
            return query.toString()
        }

        @JavascriptInterface
        fun getTerms() : String {
```

```
            Log.d("Terms", terms)
            return terms.toString()
        }

        @JavascriptInterface
        fun getVals() : String {
            return allValues
        }
    }
}
```

Flutter

In Flutter, the official WebView plugin was used [24]. Originally to display the data I tried using its `onWebViewCreated` argument and the `evaluateJavascript` method to call JavaScript after the Webview was created.

Dart

```
WebView(
    initialUrl: "file:///android_asset/flutter_assets/" + "assets/
        webviews/graph.html",
    javascriptMode: JavascriptMode.unrestricted,
    onWebViewCreated: (WebViewController webViewController) {
    //sleep(Duration(milliseconds: 600));
    webViewController.evaluateJavascript(
        "google.charts.load("current", {"packages":["line"]});"
        "var terms = $terms;"
        "var rows = $data;"
        "google.charts.setOnLoadCallback(drawChart);"
    )
);
```

However, the problem is that onWebViewCreated is run when the WebView is created and not once the JavaScript in the webview has finished executing and so the `<script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script>` script that loads the line charts hadn't finished executing and so calling the Google method results in an error. One solution would be to pause the thread for about 500ms to wait for the script to run, but this is bad in Flutter as it causes the UI to pause and there is no way to be sure that the JavaScript has finished executing by then. My final solution involved editing the flutter_webview plugin, which is quite complicated as it involves understanding how Flutter interacts with Java (which was briefly described in Chapter 2). Appendix E shows the three files that I had to edit, with my code highlighted. I was then able to pass the data into the graph using the same `@JavascriptInterface` so now the WebView could be used as follows, passing our data in with a JavascriptChannel.

Dart

```
WebView(
    initialUrl: "file:///android_asset/flutter_assets/" + "assets/
        webviews/graph.html",
    javascriptMode: JavascriptMode.unrestricted,
```

```
    javascriptChannels: <JavascriptChannel>[
    _androidJavascriptChannel(context),
    ].toSet()
);

JavascriptChannel _androidJavascriptChannel(BuildContext context) {
    return JavascriptChannel(
        name: "Android",
        data: data,
    )
}
```

Evaluation
Having this WebView problem and fixing it shows one of the main issues when using Flutter - if you have an issue there is a lack of support and fixing the issue may require a deeper understanding of Android development and knowledge of how the Flutter-Android communication channel works. This is why many Flutter jobs advertise that the developer should also know either Objective-C/Swift or Java/Kotlin so that if the developer faces a Flutter problem they can fix it with the Native implementation.

	Difficulty Level
Native	Intermediate
Flutter	Hard

Table 4.7: Web-based Content Comparison

4.5 Touch and Input

In the app there are three different types of input interactions - buttons which the users can click on, lists that the user can scroll through, and EditText Views where the user can type their query in.

4.5.1 Interactive Buttons

In Native any View that is a type of Button has a `setOnClickListener` method. In Flutter all Button Widgets have an `onPressed` method. What follows is how the Achievements button is implemented in the MainActivity.

Native

Kotlin

```
achieveBtn = findViewById(R.id.achieve_btn)

achieveBtn?.setOnClickListener {
    //start AchievementsActivity
}
```

Flutter

Dart

```
RaisedButton(
    onPressed: () {
        //start AchievementsActivity
    }
)
```

Evaluation
With both implementations handling buttons and their interactions is easy.

	Difficulty Level
Native	Easy
Flutter	Easy

Table 4.8: Interactive Buttons Comparison

4.5.2 List Views

In both Native and Flutter certain Views/Widgets automatically scroll if the list exceeds the size of the screen. In Native, ListView and GridView are examples of these that were used in the Trends app. What follows is the implementation of the ListView from AchievementsActivity.

Native

XML - ListView

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".AchievementsActivity">

    <ListView
        android:id="@+id/listview"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
    </ListView>
</LinearLayout>
```

XML - ListView Item

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
    android:orientation="vertical"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/achieveText"/>
</LinearLayout>
```

Kotlin

```
class AchievementsActivity : AppCompatActivity() {

    val achievements = ArrayList<String>()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val listView = findViewById<ListView>(R.id.listview)
        listView.adapter = AchieveAdapter(this, achievements)
    }

    class AchieveAdapter(context: Context, private var values: List<
       String>) :
            ArrayAdapter<String>(context, R.layout.
               list_item_achievements, values) {

        override fun getView(position: Int, convertView: View?,
           parent: ViewGroup): View {
            var item = convertView
            if (item == null) {
                item = LayoutInflater.from(context).inflate(R.layout
                   .list_item_achievements, parent, false)
            }
            val textView = item?.findViewById<TextView>(R.id.
               achieveText)
            textView?.text = values[position]
            return item!!
        }
    }
}
```

Flutter

Dart

```
class AchievementsPage extends StatelessWidget {
    AchievementsPage({Key key}) : super(key: key);

    List<String> achievements;

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: Text(title),
            ),
```

```
            body: ListView.builder(
                itemCount: achievements.length,
                itemBuilder: (context, index) {
                    return ListTile(
                        title: Text("${achievements[index]}"),
                    );
                },
            ),
        );
    }
}
```

Evaluation
Flutter's approach to ListViews is far more readable and easier to understand as it doesn't require very technical concepts such as ArrayAdapters and LayoutInflaters.

	Difficulty Level
Native	Intermediate
Flutter	Easy

Table 4.9: ListView Comparison

4.5.3 EditText

In the Trends app when the user starts a game the Android keyboard should appear so they can enter their answer for that round in the text box. In Native this is done with an EditText View.

Native

XML

```
<EditText
    android:layout_width="wrap_content"
    android:layout_height="41dp"
    android:inputType="textPersonName"
    android:ems="10"
    android:id="@+id/editText"/>
```

Then to make the keyboard popup call the `showKeyboard` method, and to get the user input call `text`.

Kotlin

```
editText = findViewById(R.id.editText)
editText?.showKeyboard()
editText!!.text.toString()
```

In Flutter we use the TextField Widget, setting `autofocus` to true. To get the user input a TextEditingController can be used.

Flutter

Dart

```
final myController = new TextEditingController();
TextField(
    controller: myController,
    autofocus: true,
)
myController.text;
```

Evaluation
With both implementations using EditText Views is easy.

	Difficulty Level
Native	Easy
Flutter	Easy

Table 4.10: EditText View Comparison

4.6 Audio and Video

For the Trends app I wanted to add a short video tutorial to explain the game to new users. I chose to do the video as a YouTube video because it keeps the app size small and as playing the game requires an internet connection it will not affect the user's experience.

Native

Kotlin (android-youtube-player API [25])

```
<com.pierfrancescosoffritti.androidyoutubeplayer.core.player.views.
    YouTubePlayerView
    android:id="@+id/youtube_player_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"

    app:videoId=video_id
    app:autoPlay="true"
    app:showFullScreenButton="true"/>
```

Flutter

Dart (youtube_player plugin [26]).

```
YoutubePlayer(
    context: context,
    source: video_id,
    quality: YoutubeQuality.HD,
)
```

Evaluation
Both implementations worked well and were relatively easy to implement, but it is worth noting that the Kotlin API only works if the project uses AndroidX [27], otherwise it needs to be migrated [28]. I found (as have others [29]) that upgrading the app to AndroidX required a few hours of manual changes and debugging. Nevertheless, both libraries make implementing videos easy.

	Difficulty Level
Native	Easy
Flutter	Easy

Table 4.11: Audio and Video Comparison

4.7 Background Tasks

For Android apps everything that affects the UI happens in the main thread (also known as the UI thread). All other processes should be run on another thread as a background task - particularly those that can take a long time to execute. In this app the longest running background task is getting the results from our server. However, for the Native implementation the Volley library [30] was used and this already runs network tasks on a separate thread. Another example where the app uses asynchronous programming is in the MainActivity - when users press the play button the app makes sure they have an internet connection. If they do they go to the ThemesActivity, otherwise a toast appears informing them that an internet connection is required.

Native

Kotlin - (Anko library [31])

```
doAsync {
    val isOnline = checkInternetConnection()
    uiThread {
        if (isOnline) {
            //start ThemesActivity
        } else {
            toast(''Internet Connection Required'')
        }
    }
}
```

Flutter

Dart

```
() async {
    final isOnline = await checkInternetConnection()
    if (isOnline) {
        //start ThemesPage
    } else {
```

```
        //Uses the Toast Plugin - https://pub.dartlang.org/packages/
            fluttertoast
        Fluttertoast.showToast(msg: "Internet Connection Required");
    }
}
```

Evaluation
As can be seen, when using the Anko plugin, asynchronous tasks look very similar in Native and Flutter and are easy to implement.

	Difficulty Level
Native	Easy
Flutter	Easy

Table 4.12: Background Tasks Comparison

4.8 App Data and Files

Android apps do not automatically save state or data within the app. For the Trends app I needed to keep track of achievements and the user's app preferences and so the app needed to make use of Android's file storage. This was implemented using SharedPreferences [32] which allow the app to read and write persistent key-value pairs of primitive data types.

Native

Kotlin - Set Shared Preference

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?:
    return
with (sharedPref.edit()) {
    putBoolean("Beat the Computer", true)
    commit()
}
```

Kotlin - Get Shared Preference

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?:
    return
val isUnlocked = sharedPref.getBoolean("Beat the Computer", false)
    //false is the default value if "Beat the Computer" isn't set yet
```

Flutter

Dart - Set Shared Preference (shared_preferences plugin [33])

```
SharedPreferences prefs = await SharedPreferences.getInstance();
prefs.setBool("Beat the Computer", true);
```

Dart - Get Shared Preference (shared_preferences plugin [33])

```
SharedPreferences prefs = await SharedPreferences.getInstance();
isUnlocked = prefs.getBool("Beat the Computer") ?? false; //returns
    null if not set, in which case set to false
```

Evaluation
Both implementations are easy, with Flutter being more concise and readable but Native doesn't require importing an external plugin.

	Difficulty Level
Native	Easy
Flutter	Easy

Table 4.13: App Data and Files Comparison

4.9 Connectivity

In order for the Trends app to get the popularity of the search query combinations it needs to send the queries (and other information such as whether or not the user is playing CPU mode, and if so, what difficulty the CPU is) to the Trends server. This is done by sending the data as a JSON POST request and then waiting for the server to send back the results as a JSON POST response.

Native

Kotlin (Volley HTTP library [30])

```
var results: MutableList<JSONArray> = mutableListOf()

val queue = Volley.newRequestQueue(activity)
val url = "https://trends-app-server.herokuapp.com/"
val jsonObj = JSONObject()
jsonObj.put("mode", mode)
jsonObj.put("query", query)
jsonObj.put("values", JSONArray(terms))

val jsonObjReq = JsonObjectRequest(
    Request.Method.POST, url,
    jsonObj, Response.Listener<JSONObject> { response ->
        val avgs = response.getJSONArray("averages")
        response.getJSONArray("values").iterator().forEach {
            results.add(JSONObject(it.toString()).getJSONArray("
                value"))
        }
    }
)

queue.add(jsonObjReq)
```

Flutter

Dart (Official http plugin [34])

```
final url = "https://trends-app-server.herokuapp.com/";
final response = await http.post(url, body: json.encode({"mode":
    mode, "query": query, "values": terms}));
final responseJson = json.decode(response.body);
List<String> avgs = json["averages"];
List<List<String>> results = [];
responseJson["values"].forEach((val)=>results.add(val["value"]));
```

Evaluation
I found both approaches to work well and easy to implement, but as seen above the Flutter JSON library is more concise.

	Difficulty Level
Native	Easy
Flutter	Easy

Table 4.14: Connectivity Comparison

4.10 Google Play Instant

In March 2018 Google introduced Google Play Instant [35] which allows users to try apps without downloading them. This only works if the app APK is less than 10MB. In Section 5.4 the Native version of the Trends app has a size of 2.33MB while the Flutter version is 6.07MB. Therefore, both are Google Play Instant compatible, although due to the extra size that Flutter SDKs have it is possible that, for some apps, a Native implementation would be compatible while the Flutter implementation wouldn't.

4.11 Testing

Testing is an important part of app development. Native Android testing defines three different types of tests:

- Unit Tests - run in isolation from production systems, testing a single function, method, or class. They typically mock external components of the app and so run fast as they don't require the use of an Android device.
- Integration tests - use several components to test how they interact with each other. These tests run on an Android device.
- UI tests - test that the UI has been drawn correctly and that all components display as expected. These run on an Android device.

In Flutter these tests can often be achieved with Widget tests, which can test Unit, Integration, and UI tests test without running on an Android device. In this section a Unit, Integration, and UI test for the Native implementation and a Widget test for the Flutter equivalent were run.

4.11.1 Unit Tests

I tested the GameActivity to make sure that if a user's term is an empty string that the `showTermRequiredToast()` method is called, otherwise the `pushTerm(term)` method is called.

Native

Kotlin (Mockito library [36])

```
class GameUnitTests {
    @Test
    fun search_withEmptyTerm_callsShowTermRequiredToast() {
        val presenter = GamePresenter()
        val view = mock(GamePresenter.View::class.java)
        presenter.attachView(view)
        presenter.enterTerm("")
        verify(view).showTermRequiredToast()
    }

    @Test
    fun search_withTerm_callsPushTerm() {
        val presenter = GamePresenter()
        val view = mock(GamePresenter.View::class.java)
        presenter.attachView(view)
        presenter.enterTerm("A Term")
        verify(view).pushTerm("A Term")
    }
}
```

Flutter

For Flutter a Widget test was required because the UI can't be mocked in the same way it was with Native. To do this a mocked GamePresenter was created and injected into the GamePage Widget.

Dart (Unofficial Mockito Plugin [37])

```
class MockGamePresenter extends Mock implements GamePresenter {}

Widget buildTestableWidget(Widget widget) {
  //testable widgets need to be inside a MaterialApp widget to be
     testable
  return new MaterialApp(home: widget);
}

void main() {
    final presenter = MockGamePresenter();
```

```
    QueryPage queryPage = new QueryPage(presenter: presenter);

    testWidgets("Search with Empty Term calls ShowTermRequiredToast"
        , (WidgetTester tester) async {
        //create our view
        await tester.pumpWidget(buildTestableWidget(queryPage));

        Finder queryField = find.byKey(Key("TextField"));
        await tester.enterText(queryField, "");
        Finder fab = find.byKey(new Key("FAB"));
        await tester.tap(fab);
        await tester.pump();

        verify(presenter.ShowTermRequiredToast());
    });

    testWidgets("Search with Term calls PushTerm", (WidgetTester
        tester) async {
        await tester.pumpWidget(buildTestableWidget(queryPage));

        Finder queryField = find.byKey(Key("TextField"));
        await tester.enterText(queryField, "hello world");
        Finder fab = find.byKey(new Key("FAB"));
        await tester.tap(fab);
        await tester.pump();

        verify(presenter.pushTerm("hello world"));
    });
}
```

Evaluation
Both libraries are well documented and make running Unit tests easily manageable.

	Difficulty Level
Native	Easy
Flutter	Easy

Table 4.15: Unit Tests Comparison

4.11.2 Integration and UI tests

For an integration and UI test the following tests were run:

- All Buttons are displayed
- That pressing the + button for Party Mode increases the number of players to 5 and no higher.

Native

Kotlin (Espresso [38])

```
class MainActivityEspressoTest {
    @get:Rule
    val mActivityRule: ActivityTestRule<MainActivity> =
        ActivityTestRule(MainActivity::class.java)

    @Test
    fun checkAllButtonsDisplayed() {
        listOf(R.id.helpBtn, R.id.partyAddBtn, R.id.partyRemBtn, R.
            id.partyPlayBtn, R.id.cpuAddBtn, R.id.cpuRemBtn, R.id.
            cpuPlayBtn, R.id.achieveBtn
        ).forEach {
            viewDisplayed(it)
        }
    }

    @Test
    fun testAddButton() {
        onView(withId(R.id.partyTextView))
            .check(matches(withText("2")))
        clickButton(R.id.partyAddBtn)
        onView(withId(R.id.partyTextView))
            .check(matches(withText("3")))
        clickButton(R.id.partyAddBtn, 3)
        onView(withId(R.id.partyTextView))
            .check(matches(withText("5")))
    }

    fun viewDisplayed(rid: Int) {
        onView(withId(rid))
            .check(matches(isDisplayed()))
    }

    fun clickButton(rid: Int, xTimes: Int = 1) {
        when(xTimes) {
            1 -> onView(withId(rid)).perform(click())
            2 -> onView(withId(rid)).perform(click()).perform(click
                ())
            3 -> onView(withId(rid)).perform(click()).perform(click
                ()).perform(click())
            else -> throw Exception("Clicking button an unexpected
                number of times")
        }
    }
}
```

Flutter

Dart

```dart
void main() {

  final partyPlusBtnFinder = find.byKey(Key("Party Mode plusBtn"));

  testWidgets("Test All Buttons Loaded", (WidgetTester tester) async
      {
    // Build our app and trigger a frame.
    await tester.pumpWidget(new MyApp());

    //Verify all buttons loaded
    expect(find.byIcon(Icons.add_circle), findsNWidgets(2)); //2
        plus buttons
    expect(find.byIcon(Icons.remove_circle), findsNWidgets(2)); //2
        minus buttons
    expect(find.byIcon(Icons.play_circle_filled), findsNWidgets(2));
         //2 play buttons
    expect(find.byIcon(Icons.help), findsOneWidget); //1 help button
    expect(find.text("Achievements"), findsOneWidget); //1
        achievements button
  });

  testWidgets("Test Add Button", (WidgetTester tester) async {
    // Build our app and trigger a frame.
    await tester.pumpWidget(new MyApp());

    //Verify that our counter starts at 2.
    expect(find.text("2"), findsOneWidget);

    // Tap the "+" icon and trigger a frame.
    await tester.tap(partyPlusBtnFinder);
    await tester.pump();

    // Verify that our counter has incremented.
    expect(find.text("3"), findsOneWidget);

    // Tap the "+" icon 3 times and trigger a frame.
    await tester.tap(partyPlusBtnFinder);
    await tester.tap(partyPlusBtnFinder);
    await tester.tap(partyPlusBtnFinder);
    await tester.pump();

    // Verify that our counter has incremented.
    expect(find.text("5"), findsOneWidget);
  });
}
```

Evaluation
Testing in Native and Flutter had pros and cons. Flutter has the benefit that more complex tests that require an Android device for Native can be performed much faster in Flutter as they don't require an Android device. Also, in Flutter Widget searching seems powerful and efficient as searching for a widget can be done using a key `find.byKey(Key("Party Mode plusBtn"))` (similar to how it's done Natively) or by describing the widget `find.byIcon(Icons.add_circle)`. However, I also found that Flutter documentation on testing is scarce, as are the tools which seem more limited and buggy than the Kotlin libraries. For example, I couldn't find any Widgets when the animation was on the MainPage. To fix it I had to remove the animated Widget (similar issues were faced by this discussion [39]). Native's biggest pro is that the Espresso library it uses has a very powerful toolset. For example, developers can run tests by going through them on their Android device and Espresso will write the test for them, after which they only need to enter the expected values. Overall, I believe that Native testing is more stable and easier to learn as it has more stable and powerful libraries and more online resources to learn it. However, it is worth noting that the Flutter testing library has shown some very promising results and has the potential to be as good as, if not better than, Native in the future. Flutter's stability issues caused implementing these tests to be Intermediate but Native's Espresso library makes these tests easy.

	Difficulty Level
Native	Easy
Flutter	Intermediate

Table 4.16: Integration and UI Tests Comparison

4.12 Stack Overflow

StackOverflow Trends [40] can be used to see StackOverflow question trends over time. This can provide useful insight into the popularity and strength of the community supporting a framework or programming language, although it is obviously not a perfect indicator. It is worth noting that because Flutter is open source and on Github many of the problems and questions users have are asked on Github instead of StackOverflow. As of February 2019, 11541 StackOverflow questions had the Flutter tag associated with them, however there have been 17,337 issues raised on the Flutter Github. Nevertheless, figure 4.6 shows that Flutter increased from almost 0% to 0.5% of all StackOverflow questions in 2018 alone while Android has been steadily declining since mid-2016. This could be an indication that the Android platform is becoming stable and that less new questions need to be asked but it could be also be an indication of a decline in it's popularity.

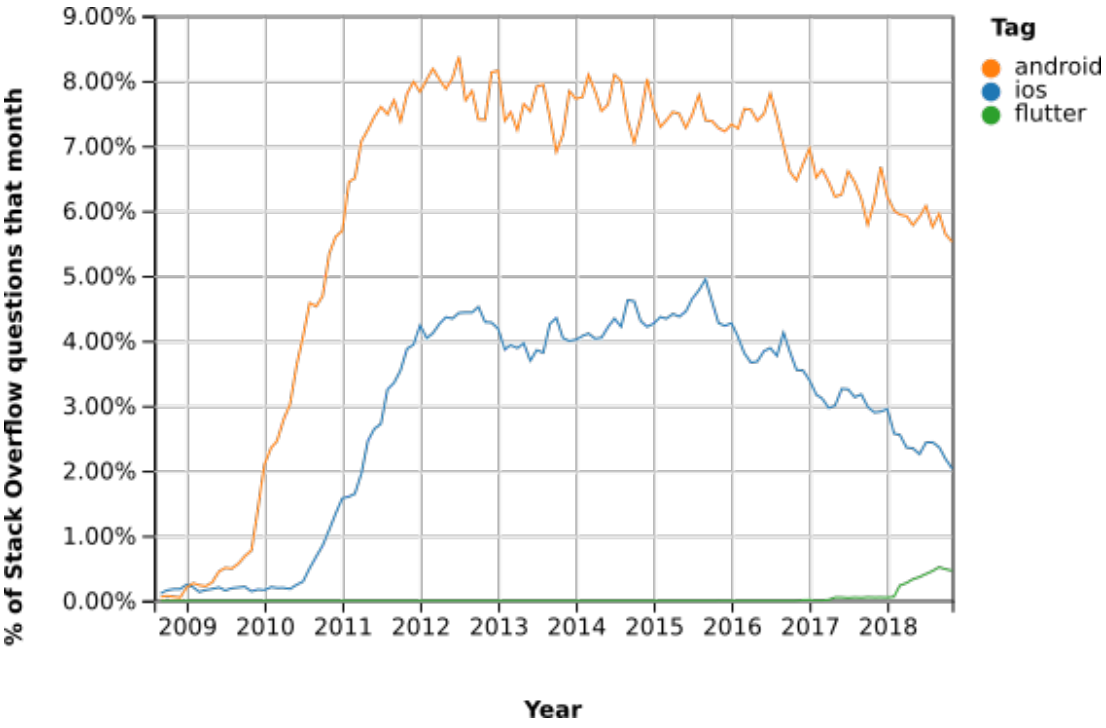


Figure 4.6: Comparing Android, iOS, and Flutter

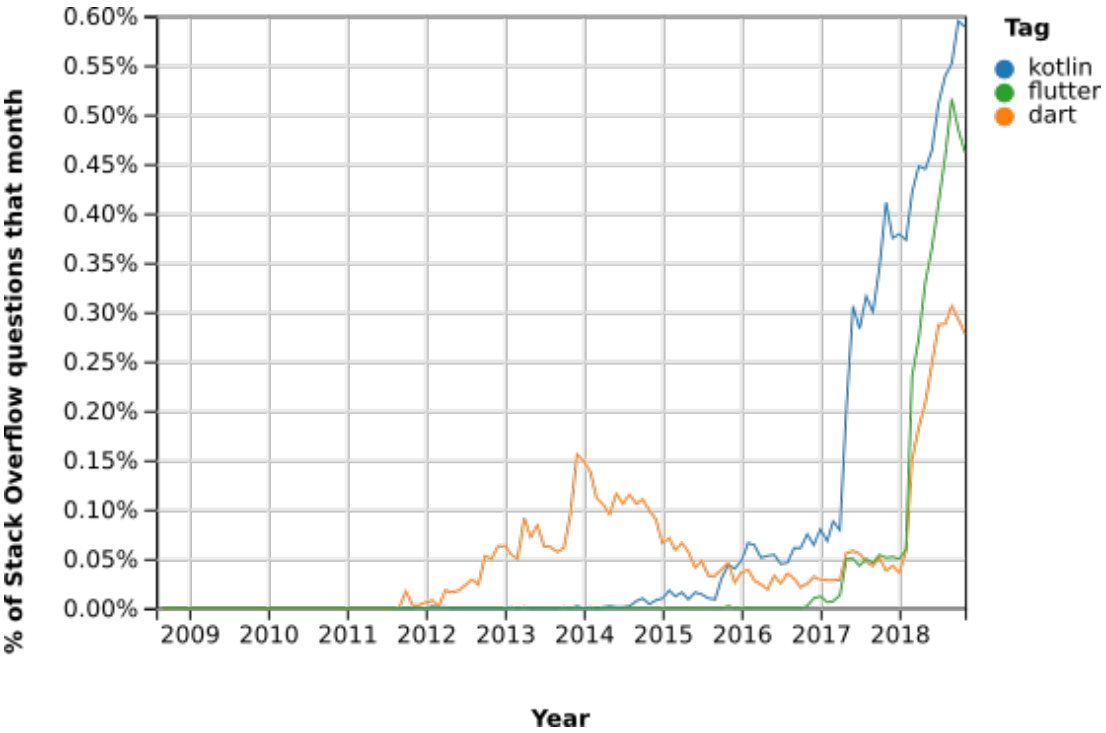


Figure 4.7: Comparing Kotlin, Flutter, and Dart

4.13 Conclusion

In this chapter we examined the eleven topics we identified in Chapter 3 and the SDKs' popularity. Overall, I found both SDKs to have their pros and cons. Flutter's biggest weakness compared to Kotlin is that as native Android development is over a decade old it is quite mature, with lots of online resources for fixing bugs. While, at the time of writing, Flutter has been out of beta for only 4 months [7]. However, the engineers at Flutter have done a far better job in writing a very developer friendly SDK - I found using Widgets for all UI components instead of having to learn about XML, Views, Layouts, Activities etc. to be significantly better and typically the Widget solution was more readable with less boilerplate. As mentioned in section 4.4 it is evident that with the current state of Flutter, if a developer has an issue with the SDK or one of its plugins they will need to either rely on the Flutter community or have some knowledge of Java and how Flutter communicates with it.

Feature	Difficulty Level	
	Native	Flutter
Basic Activity	Easy	Easy
Tabbed Activity	Intermediate	Easy
App Lifecycle	Easy	Easy
Navigation and Intents	Easy	Easy
Custom Animations	Intermediate	Hard
Transitions	Easy	Easy
Web-based Content	Intermediate	Hard
Interactive Buttons	Easy	Easy
List Views	Intermediate	Easy
EditText	Easy	Easy
Audio and Video	Easy	Easy
Background Tasks	Easy	Easy
App Data and Files	Easy	Easy
Connectivity	Easy	Easy
Unit Tests	Easy	Easy
Integration and UI Tests	Easy	Easy

Table 4.17: Difficulty Comparison of Features for the Trends app.

Table 4.17 shows that Flutter features are typically fairly simple and intuitive to implement, with the three exceptions being when there was a problem implementing a feature in the app. In contrast, with Native there no features that were hard to implement because Native features have been around for a much longer time and are so more stable. But being older some Native features felt more complicated and dated. This is particularly obvious when looking at how simple tabbed activities and list views were in Flutter compared to Native.

Chapter 5

Performance

In this Chapter, to compare the performance of the Android SDK (Native development) with Flutter both SDKs were examined and the following aspects compared:

- CPU Usage
- Memory Usage
- App Startup Time
- App Compilation Time
- APK Size

To establish another point of reference besides the Trends app, a simple Timer app by Sullivan [16] was used to compare CPU and memory usage of Native and Flutter apps. The Timer app is simply an Activity with a TextView which counts up every ten milliseconds. Because the findings were inconclusive I wrote a simple app that only animates a single straight line from the left of the screen to the right to better pinpoint the reason for this.

Finally, I used Rosetta Code [41] in order to compare the performance of complex algorithms that could potentially run on an Android app.

5.1 Different Build Modes

Before comparing Native and Flutter app performance the different build modes that a Native and Flutter app will go through during development need to be understood.

Native apps have two different build modes: Debug, and Release. Although they are similar the main differences are:

- Debug mode has the debug flag enabled which allows for debugging the app.
- Release mode has Proguard enabled which decreases the APK size by removing unused code and resources, but increases compilation time.

Flutter has three different build modes; Debug, Release, and Profile. Unlike Native development, Flutter's Debug mode is very different from the other modes:

- In Debug mode the app is compiled with Just-in-Time (JIT) compilation by creating a Script Snapshot which tokenises the code. This allows developers to get debugging information, and optimises the app for fast development by allowing for faster recompilation of the app. However, the code is not optimised for execution speed, binary size, or deployment.
- In Release mode the app is compiled with Ahead-of-Time (AOT) compilation by translating the Dart code into assembly files and compiling these files into binary code. This means that the opposite of Debug mode is true - debugging, and fast recompilation are unavailable, but the APK is optimised for fast startup, fast execution, small package sizes.
- Profile mode is the same as Release mode, except that profile-mode service extensions are enabled which allows us to use the Android Profiler.

What this means is that typically Debug mode is used while the developer is developing and profiling the app (because debugging messages are enabled and app compilation is faster) and Release mode is used for creating the app APK that end users will download onto their phones (because app size and startup time are optimised).

Therefore, when examining performance metrics that apply to the developer Debug mode will be used, when examining performance metrics that affect the end user Release mode will be used, and when profiling the app we will use Debug mode for Native and Profile mode for Flutter.

5.2 Comparing CPU and Memory Usage

To compare the apps' CPU and memory usage the Android Profiler [42] was used. The Android Profiler is a tool provided by Android Studio that provides real-time data to help developers understand how their app uses CPU and memory. To activate it press the `Profile` button in the Android Studio toolbar, and the profiler will automatically run when you run an app. To use the profiler with a Flutter project however, you need to open the app in the `project-root/android` directory.

One problem with using the Android Profiler is that there is no way to export the data and so the data can only be analysed within the profiler - this meant I had to use workarounds to capture the results within this report.

5.2.1 Trends app

CPU

Figures 5.1 and 5.2 show the results of the profiler. As stated in Section 5.2 the data from the profiler can not be exported, however, while in the profiler we can hover over the graph and the data is displayed making it easier to analyse. Therefore, I decided to

take the points of interest and draw a graph using these points (seen in Figure 5.3) to show the results clearly.

The results show that for both versions of the app it takes about 1.9 seconds for the profiler to startup and the app to install. It then takes about another 3 seconds for the Native version to start and for the MainActivity to load, while the Flutter version takes slightly longer, at about 3.4 seconds. Then for both versions the CPU load increases significantly while the MainActivity performs various startup calculations. The Flutter version uses slightly less CPU at this stage. After these startup calculations take place the app is doing nothing except for displaying the animated MainActivity. At this stage the Flutter version uses less CPU, typically around 7% of the total CPU while the Native version uses around 12%. From these results it is clear that the Flutter version uses less CPU than the Native version for the Trends app.

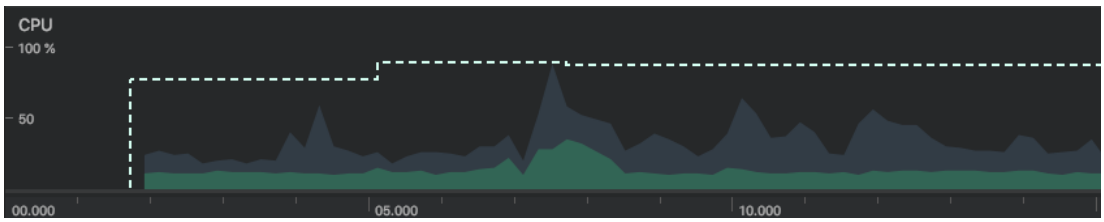


Figure 5.1: Native - Trends CPU Usage

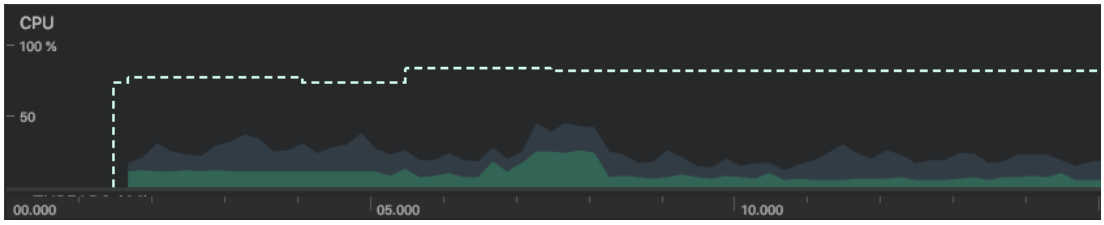


Figure 5.2: Flutter - Trends CPU Usage

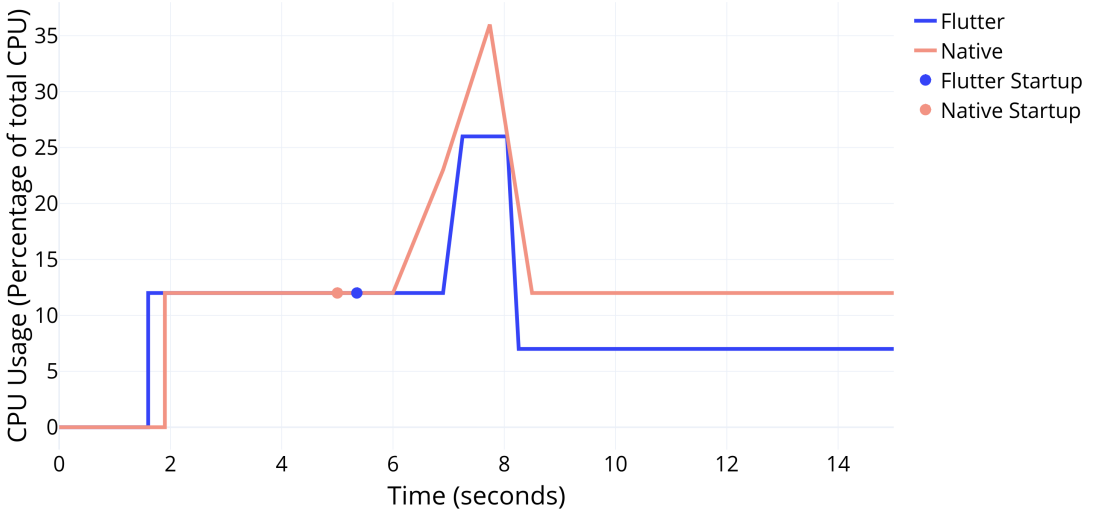


Figure 5.3: Trends CPU Usage

Memory

The Android Profiler allows us to obtain clearer screenshots of memory usage as can be seen in Figures 5.4 and 5.5. The results follow a similar trend as CPU - after startup, Flutter's memory usage is less than Native's. Flutter drops to around 80MB after startup while Native only drops to around 160MB.

The memory profiler is useful to developers as it shows which resource is using up how much memory:

- Light blue is memory allocated from Java or Kotlin code.
- Dark blue is memory allocated from C and C++ code.
- Yellow is memory used for graphics.
- Green is memory the app uses for code and resources.
- Grey is memory used that the profiler is not sure how to categorise.

The colours have been annotated and the Axes' font size increased to make them easier to read.

By analysing the figures we can see that the biggest factors that affected memory are memory allocated from C and C++, and graphics - the Native implementation uses about four times the graphics and about twice the C and C++ memory of Flutter. It is also notable that Flutter has a lot more uncategorised memory usage which probably comes from the Dart binary code.

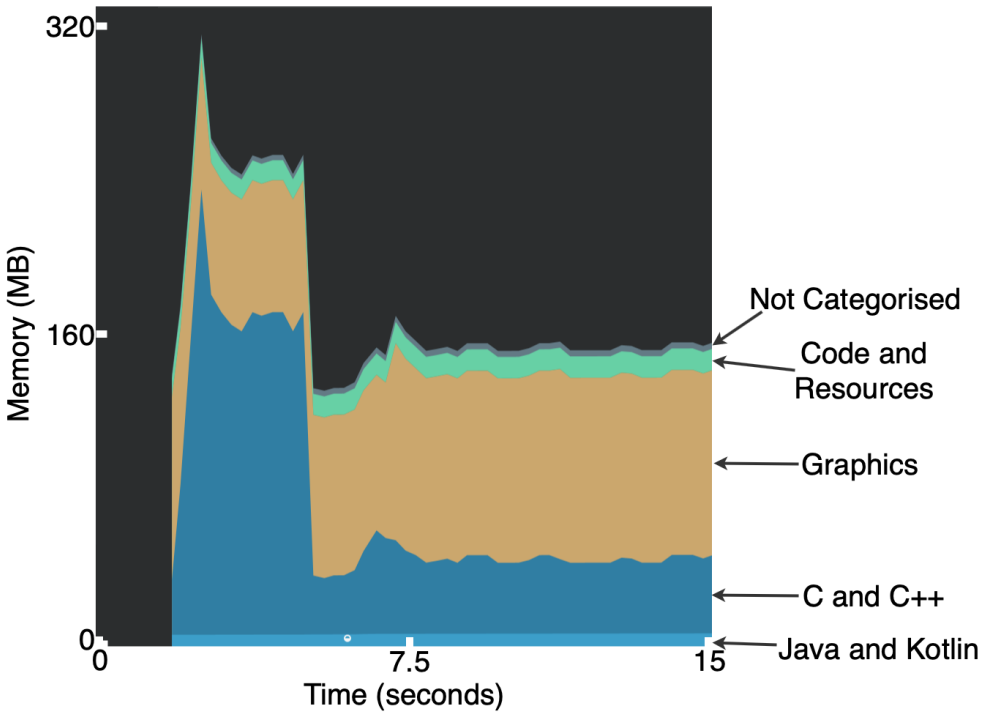


Figure 5.4: Native - Trends Memory Usage

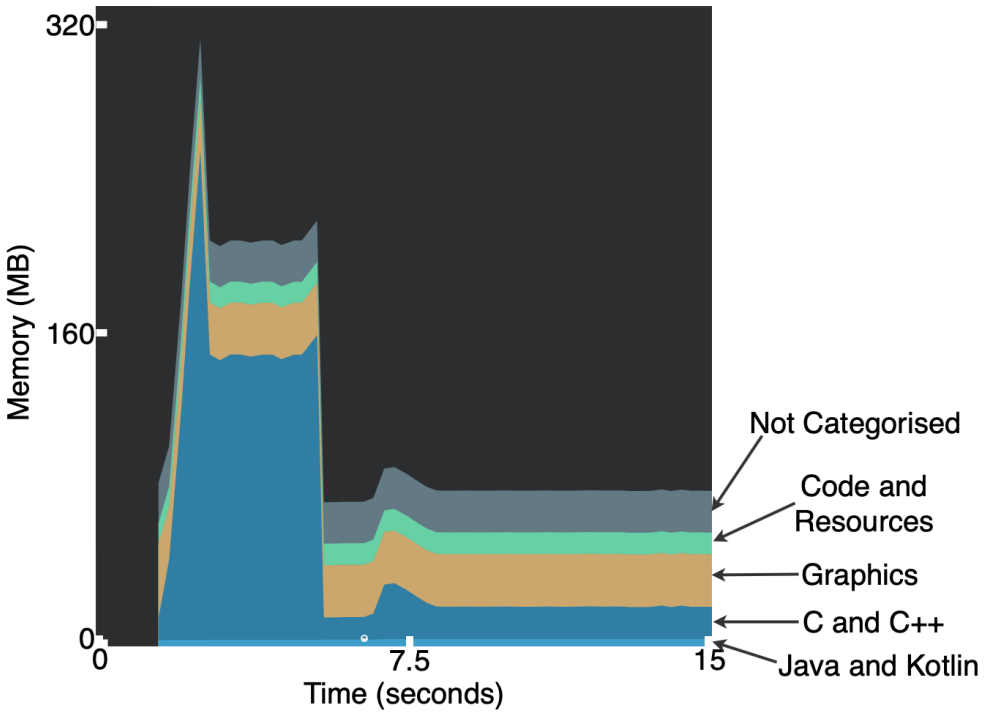
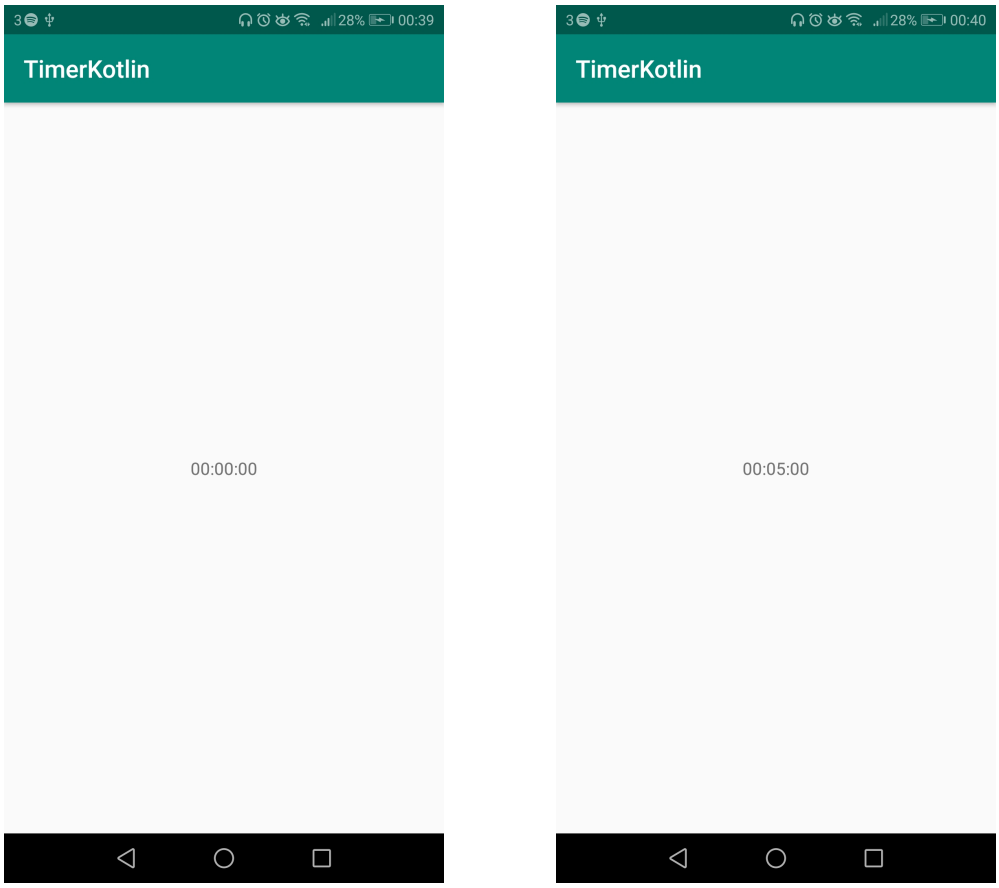


Figure 5.5: Flutter - Trends Memory Usage

5.2.2 Timer App

The Timer app is simply an Activity with a TextView which counts up every 10 milliseconds. Figure 5.6 shows how the screen looks at the start and how it looks after five seconds.



a) After Zero Seconds b) After Five Seconds

Figure 5.6: Timer App Screenshots

CPU

Figures 5.7 and 5.8 show the results of the profiler. As in Section 5.2.1, I decided to take the points of interest and draw a graph using these points (seen in Figure 5.3) to better show the results.

As in Section 5.2.1, it takes about 1.9 seconds for the profiler to startup and the app to install and about another 3 and 3.4 seconds for the Native and Flutter versions to start and the MainActivity to load. Again, for both versions the CPU load increases quite drastically while the MainActivity performs various startup calculations. However, unlike with the Trends app, with the Timer app the Native version performs better, only using 6% of the CPU after startup compared to Flutter's 8%.

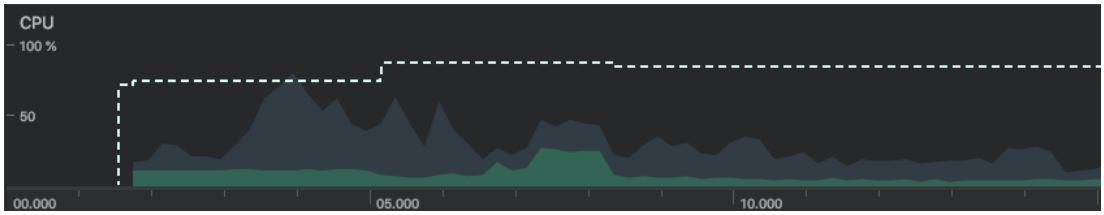


Figure 5.7: Native - Timer CPU Usage

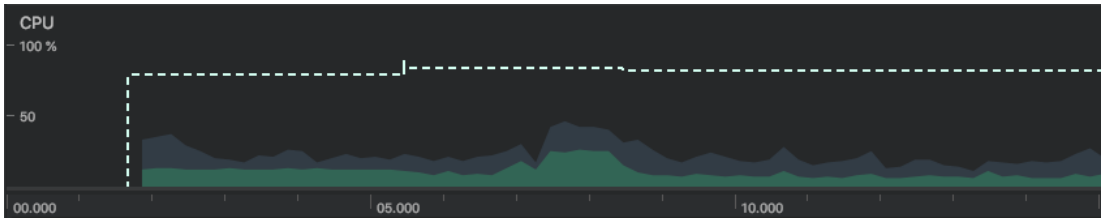


Figure 5.8: Flutter - Timer CPU Usage

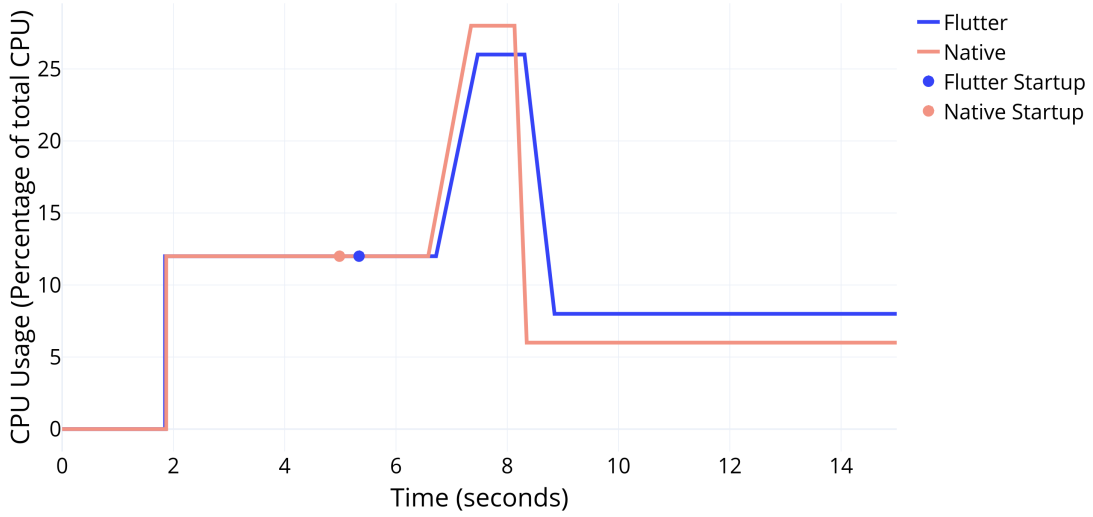


Figure 5.9: Timer CPU Usage

Memory

As with the CPU, for the Timer app, the Native version's memory usage is better than Flutter's. Native drops to around 55MB after startup while Flutter only drops to around 75MB (as seen in Figures 5.10 and 5.11).

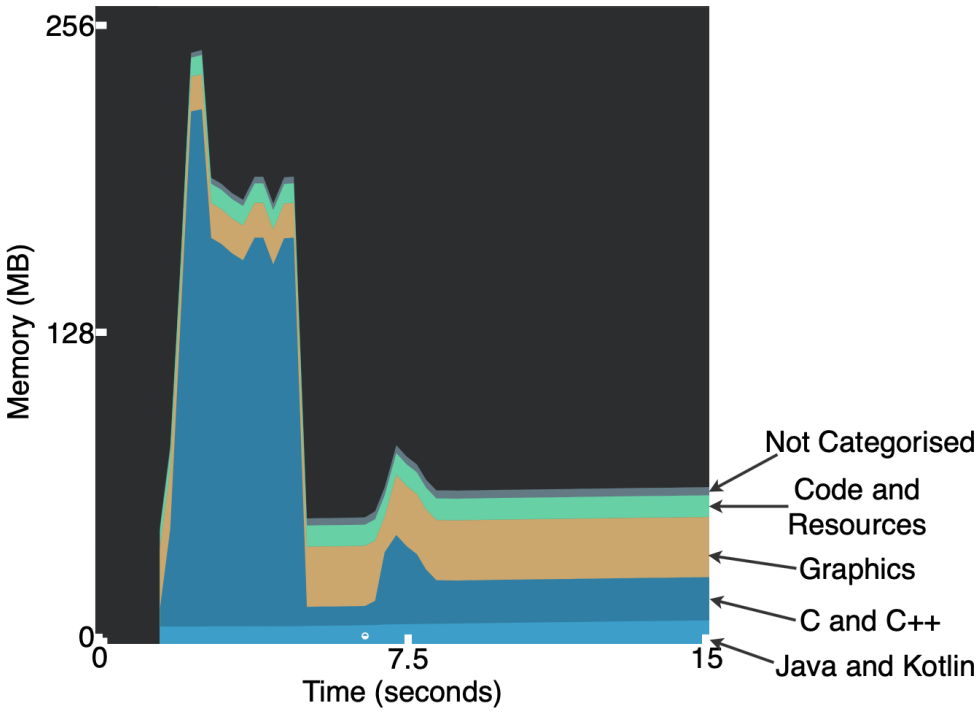


Figure 5.10: Native - Timer Memory Usage

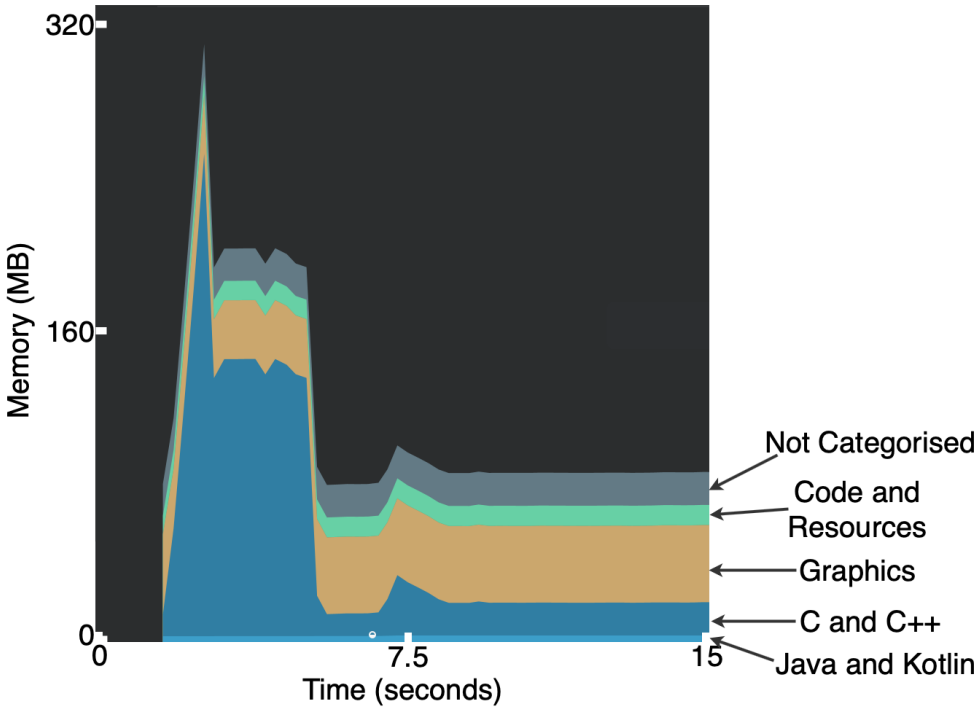
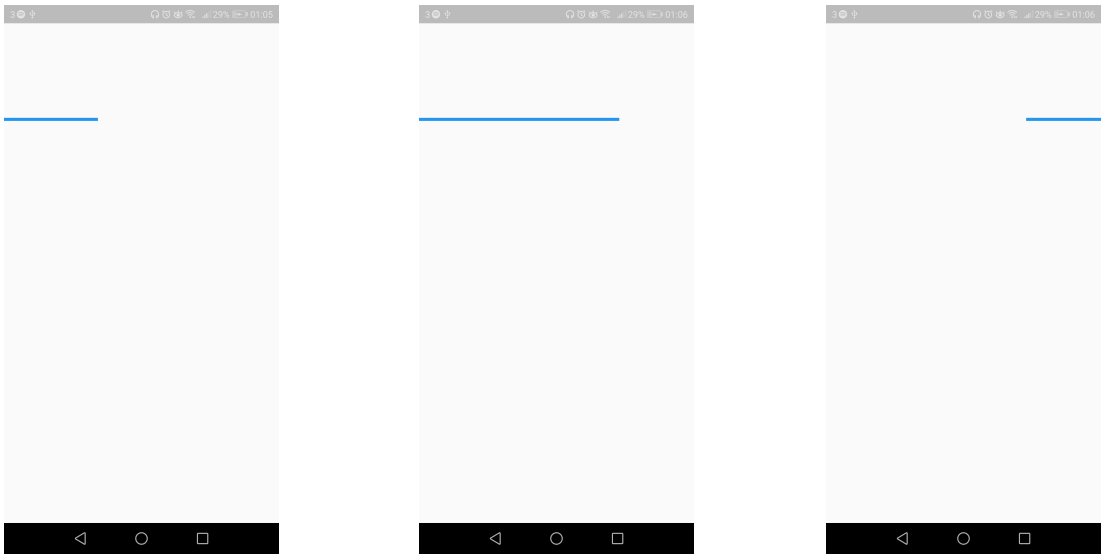


Figure 5.11: Flutter - Timer Memory Usage

5.2.3 Simple Animation App

As the results from the two apps were contradictory a simple animation app was created to animate a line from the left of the screen to the right (as seen in Figure 5.12). This was to check if the difference in performance stemmed from a possible mistake within the Trends app because it is quite complex with lots of methods and Views, compared to the one method and View within the Timer app.



a) Start b) Middle c) End

Figure 5.12: Simple Animation App Screenshots

Memory
The results (Figures 5.13 and 5.14) show that after startup the Native version uses up about 100MB of memory, while the Flutter versions only uses about 60MB. Because, this is about the same as the Trends App this shows that the cause of Flutter's better performance does in fact stem from the animation.

5.2.4 Removing the Animations

For all of the versions of all of the apps if the animation is removed, then CPU usage drops to zero, indicating that neither platform has unnecessary computations when the app is idle.

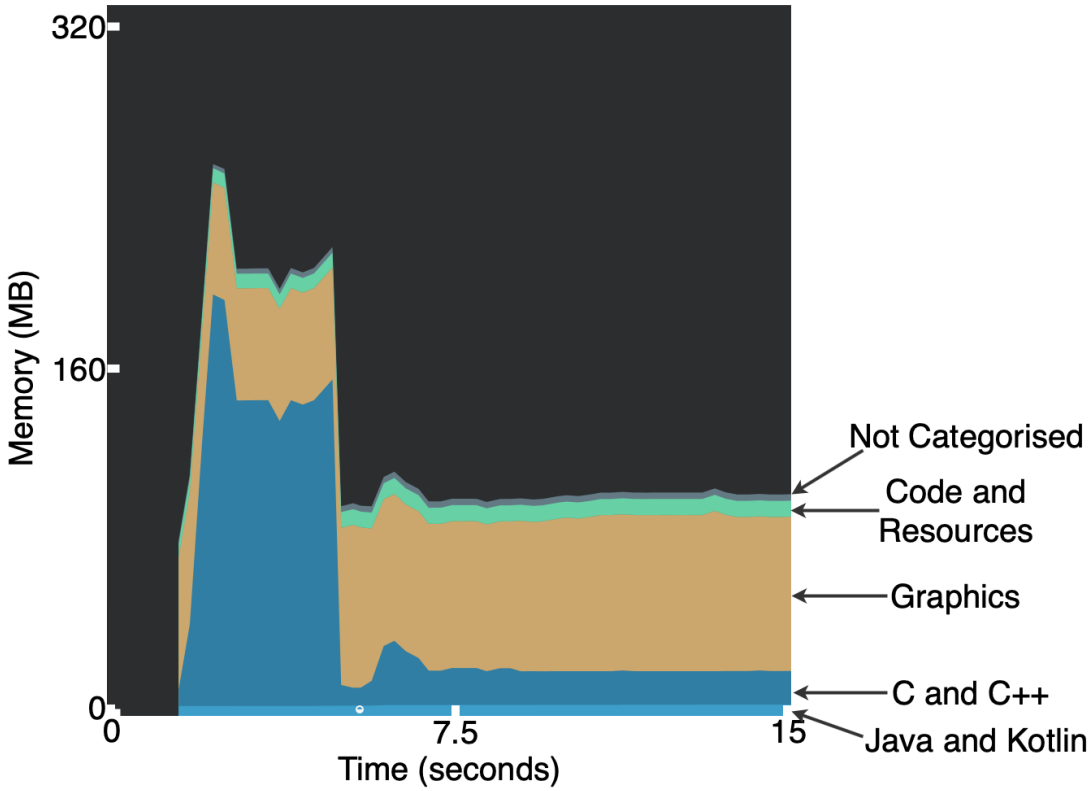


Figure 5.13: Native - Simple Animation Memory Usage

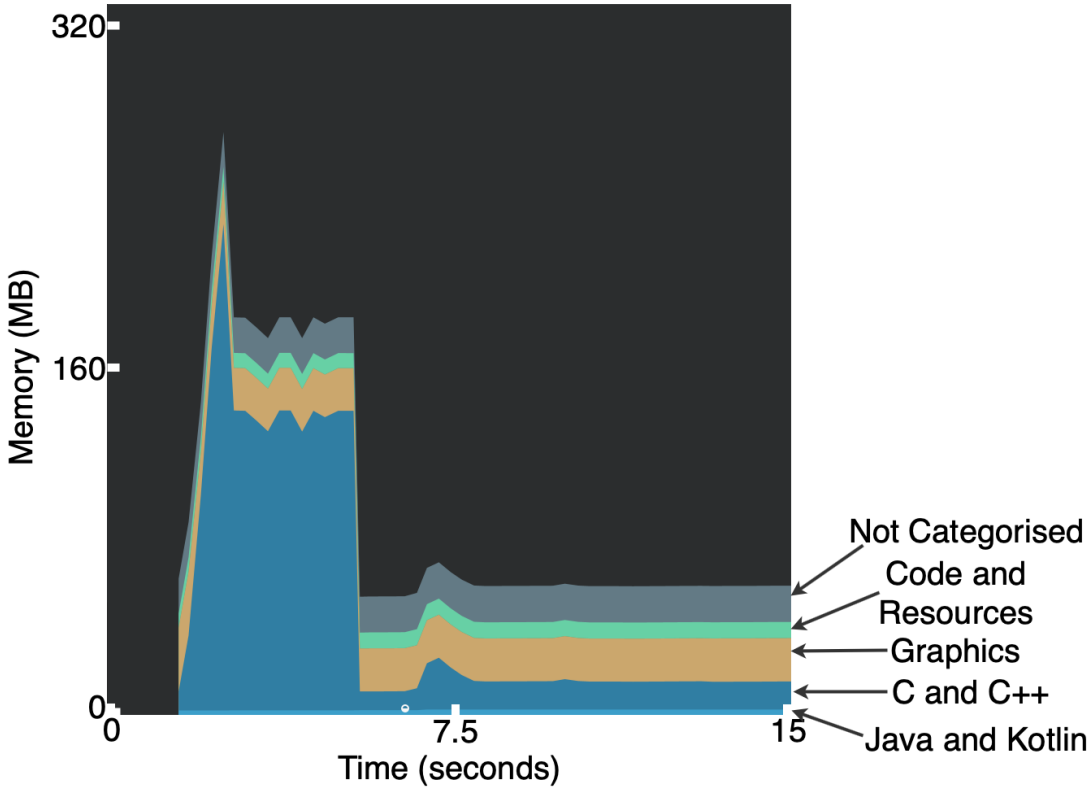


Figure 5.14: Flutter - Simple Animation Memory Usage

5.2.5 Conclusion

From the results the following can be concluded:

- When animating lines, using Flutter and a CustomPainter Widget performs better than Native and a custom animated View.
- When updating a value within the UI, using Native and having a value automatically change in the XML layout once it is changed performs better than changing the value of a Widget and calling `setState` which redraws the entire Widget.
- Neither platform has unnecessary computations when the app is idle.

5.3 Comparing App Startup Time

The startup time of an app was defined as the time it takes for the first frame of the MainActivity to appear after the app is opened. To calculate this in Kotlin the `reportFullyDrawn()` method was added to the `onCreate` method in the MainActivity. However, for Flutter additional code was needed in the MainActivity.java file found in the `<project-root>/android` directory.

Native

Kotlin

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    reportFullyDrawn()
}
```

Flutter

Java

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    GeneratedPluginRegistrant.registerWith(this);
    getFlutterView().addFirstFrameListener(new FlutterView.
        FirstFrameListener() {
         @Override
         public void onFirstFrame() {
              MainActivity.this.reportFullyDrawn();
              MainActivity.this.getFlutterView().
                  removeFirstFrameListener(this);
         }
    });
}
```

After adding and compiling the code, logcat [43] was opened (logcat is the built-in system message logging and debugging tool for Android Studio), filters disabled and

'Fully drawn' was searched for. To calculate the average startup time I started each app ten times and averaged the results of the Native and Flutter versions. Figure 5.15 shows that Flutter's startup time was around 0.15-0.2 seconds slower than Native's.

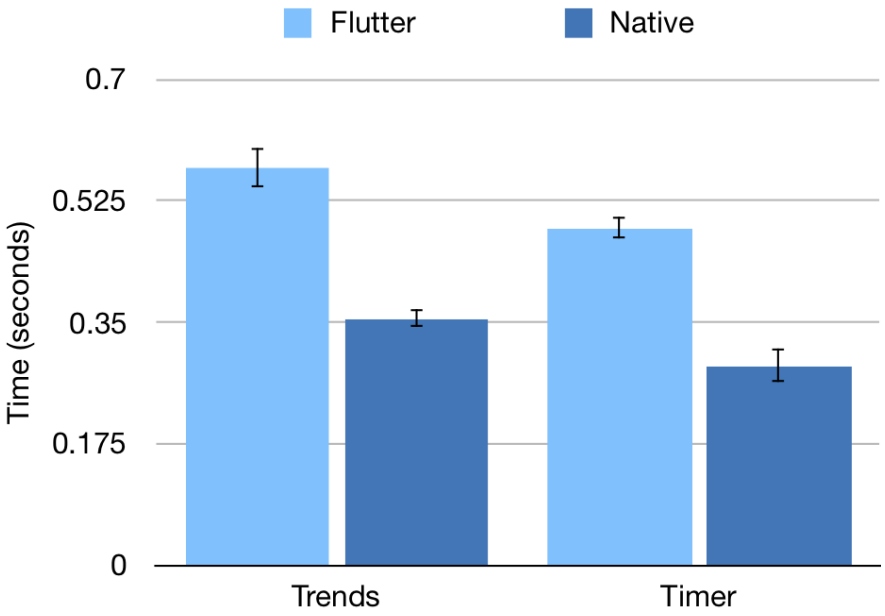


Figure 5.15: Startup Time Comparison

5.4 Comparing App Compilation

To compile an app and run it on Android the app needs to be built as an APK file. In Flutter this is done with the `flutter build apk` command, and in Native this is done with the `./gradlew assembleDebug` command. The time is calculated by prepending the `time` command to the command we run in the terminal. To compare app compilation of the apps the compilation time of the Debug versions and the APK size of Release versions were examined. The Debug mode compilation time was used as this impacts developers who will build debug versions of apps multiple times before creating a final release version. The size of APK for the Release mode was used as this impacts users who will install the APK on their Android device. As with section 5.3 to calculate the average time, each app was compiled ten times and the results averaged.

Figure 5.16 shows that Flutter's compilation time was around 11-12 seconds faster than Native's, which is likely because Flutter apps are Just-in-Time compiled when in Debug mode. Figure 5.17 shows that the Flutter APKs are at least 3MB more than their Native counterpart. This is not likely to be significant as users will typically have a 32 or 64GB smartphone.

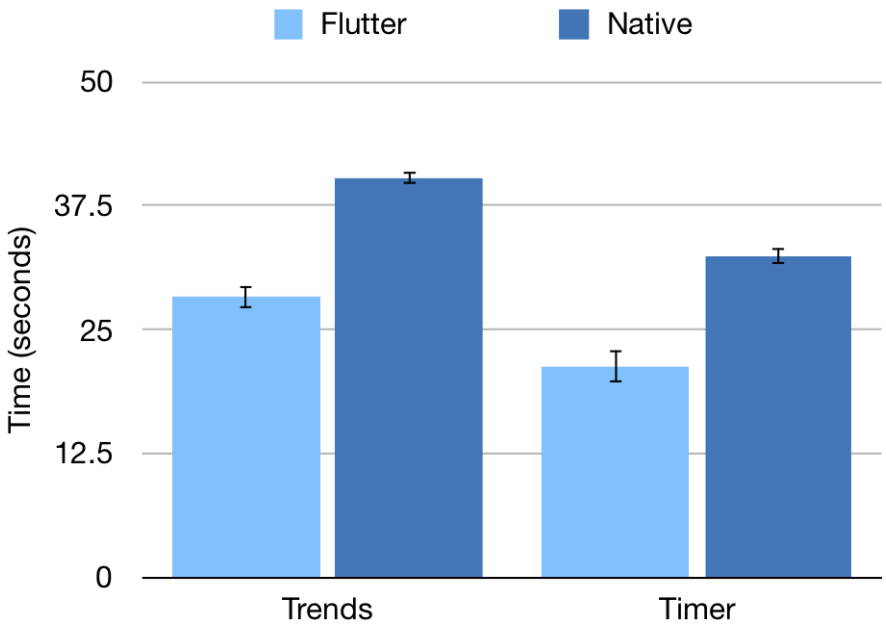


Figure 5.16: Compilation Time Comparison

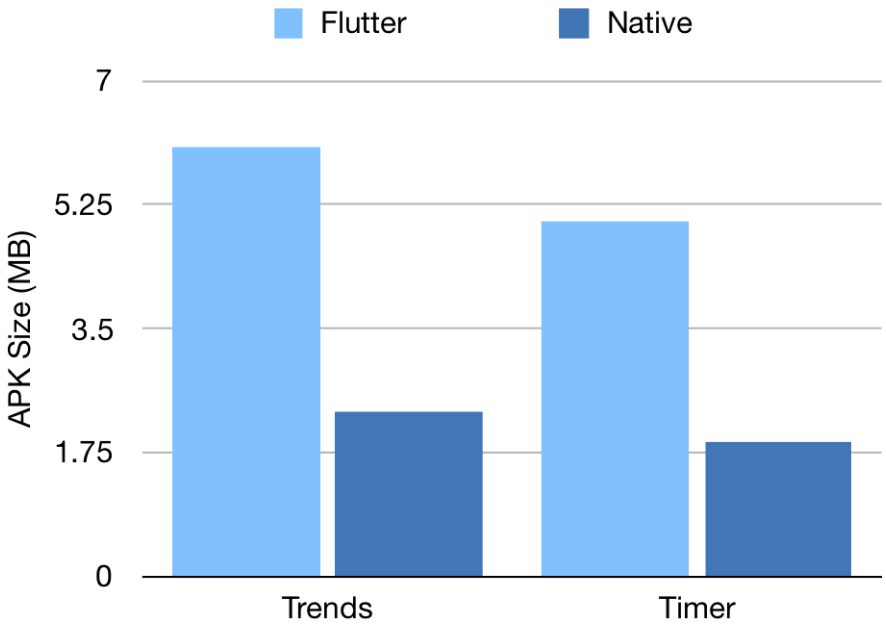


Figure 5.17: APK Size Comparison

5.5 Dart vs Kotlin - a Rosetta Code Case Study

Rosetta Code [41] is a website with a wide array of algorithms and programming problems. Users then solve these problems using their desired programming language. As none of the apps developed for this paper had very complex algorithms, code from Rosetta Code was used to test the two SDKs with more complex algorithms. The time taken to run the algorithms on an Android device using either Native and Kotlin code or Flutter and Dart code was measured.

Kotlin is one of the officially supported languages on Rosetta Code and so has over 1000 solutions to different problems. However, Dart is not supported and, as of February 2019, only has 73 solutions. As many of these problems are very simple, such as printing a single string or making a single assertion, I decided to take a sampling of 22 Dart and Kotlin solutions that seemed somewhat complex, and using an empty class as a baseline, compare their performances. Table 5.1 shows all of the algorithms and their corresponding letter in the graphs.

Letter	Name
A	Arbitrary Precision Ints
B	Array Length
C	Average Number
D	Babbage Problem
E	Binary
F	Bubble Sort
G	Caesar Cipher
H	Coin Counter
I	Dot Product
J	Empty Class
K	Factorial
L	Fibonacci Series
M	Greatest Number
N	Hamming Numbers
O	Heap Sort
P	http
Q	Kaprekar Numbers
R	Knapsack Problem
S	Monte Carlo
T	N Queens
U	Prime Generator
V	Quick Sort
W	RegEx

Table 5.1: List of Algorithms From Rosetta Code

Figure 5.18 shows the runtime of all of the algorithms. Typically, the algorithms ran very fast, taking between 0.001 and 0.3 seconds. However, for the following five algorithms; A, F, H, N, Q the two languages had a run time difference of more than a second.

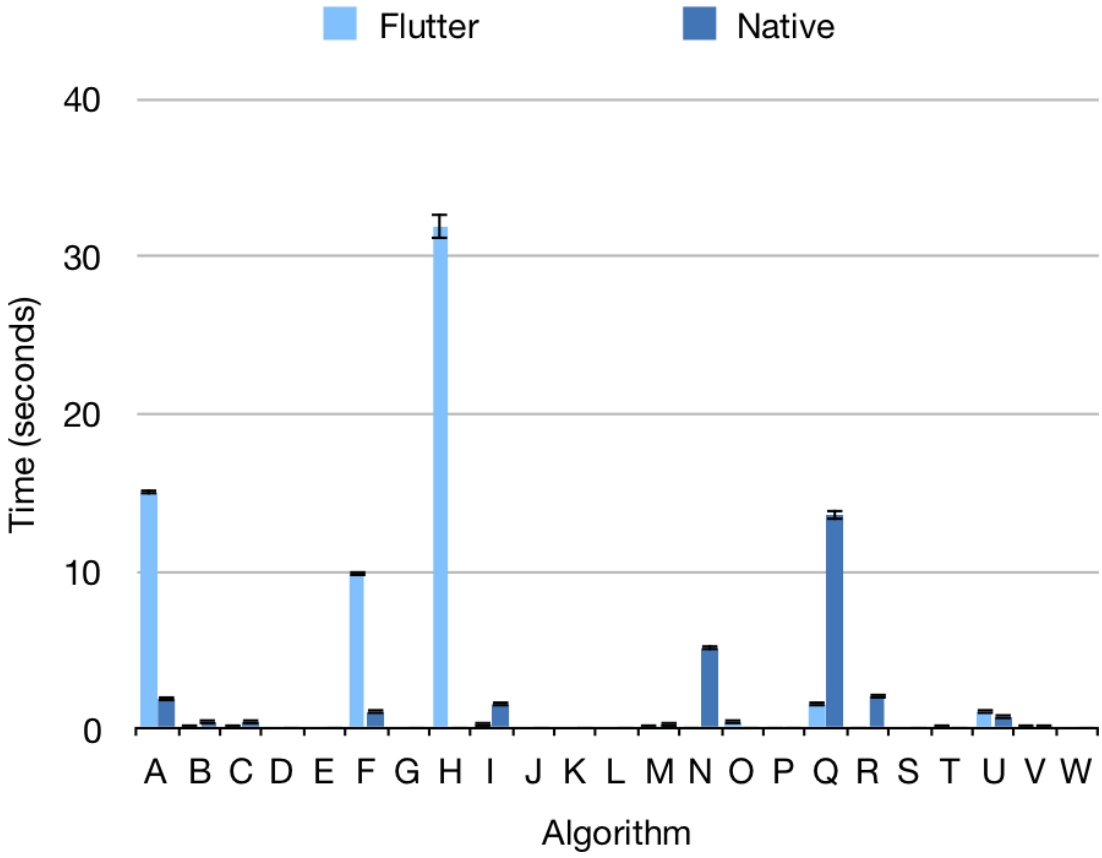


Figure 5.18: Rosetta Code Algorithms - Before Optimisation

Algorithms A took significantly longer for the Flutter implementation. This was because the algorithm requires numbers past Dart's MAXINT value. As a result the algorithm used a BigInt object, which according to the Dart website [44] are not optimised and this likely explains why it took so much longer than the Native implementation.

Algorithm F took significantly longer for the Flutter implementation. This was because the algorithm used a growable list. According to Posva [45], growable lists are slower in Dart because when accessing an element the algorithm has to first check the length of the growable list, then read the reference to the backing store and finally can it then access the requested element from the backing store. After I changed `List<int> arr = new List<int>.generate(10000, (i) => 0)` to `List<int> arr = List(10000)` the time taken dropped to around 0.4 seconds, well within a second of the Native implementation.

Algorithm H took significantly longer for the Flutter implementation. As with Algorithm F, this was because a growable list was used. After changing this to a fixed-length list the algorithm took around 0.2 seconds for the Flutter implementation.

Algorithm N took significantly longer for the Native implementation. The algorithm

involves calculating the one millionth hamming number [46]. Before the one millionth hamming number both Dart and Kotlin's MAXINT value is exceeded. Therefore the BigInt object has to be used for both languages, which is inefficient. However, the Flutter implementation used a custom class to represent the numbers without using the BigInt object. After the one millionth number was calculated it was converted to a BigInt. This meant that the Flutter implementation did not have to keep a list of BigInts and so it was faster than the Native implementation.

Algorithm Q took significantly longer for the Native implementation. This was because Kotlin's `pow` method was slower than Dart's. To fix this I wrote my own `pow` method in Kotlin. After doing so the Native implementation took about half as long as before and is now within a second of the Flutter implementation.

After making these changes, all algorithms, except for A and N, had both implementations within one second of each other. The new algorithm performance graph can be seen in Figure 5.19.

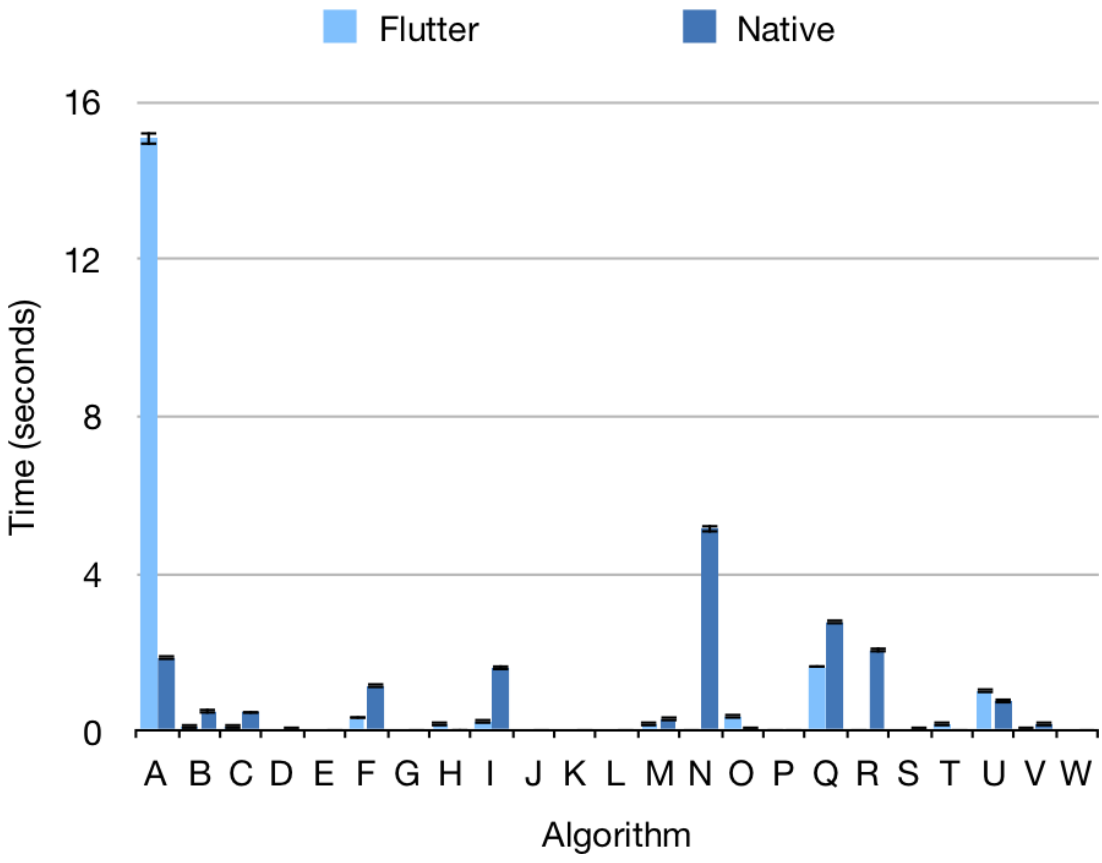


Figure 5.19: Rosetta Code Algorithms - After Optimisation

Overall, this case study shows that the run time of both Native and Flutter development is typically very fast with complex algorithms, and that both languages have their own unique quirks that developers need to take into account when writing and optimising algorithms for Android devices.

5.6 Conclusion

The following can be concluded:

- Flutter is better at optimising the performance of some UI components, while Native is better at optimising other UI components.
- Flutter apps seem to have a slightly larger APK size (around 3MB).
- Flutter apps take around 0.2 seconds longer to startup.
- Flutter compilation time is around 11 seconds faster when it comes to compilation time from scratch.
- Kotlin and Dart are very good at handling complex algorithms when Ahead-of-Time compiled, but require careful implementation.

Overall, the results show that Google's claims that Flutter provides native Android performance holds up because the performance differences are very small and sometimes even go in Flutter's favour.

Chapter 6

Conclusion

This study aimed to compare Android development using the Android SDK with the Flutter SDK. The following objectives were met:

- Examine what evaluation metrics are important for comparing mobile SDKs.
- Examine what features should be tested when comparing mobile SDKs.
- Develop an app in both SDKs that incorporates most of these features.
- Compare the ease-of-development of both apps.
- Compare the performance of both apps.
- Compare the performance of other simple apps.
- Compare the performance of algorithms compiled with both SDKs.

Results showed that the Flutter SDK is very intuitive and easy to develop with - often more intuitive than the Android SDK because much boilerplate code is abstracted in Flutter. Although, some features are lacking and require a lot of research or programming to solve this is to be expected from such a new SDK and should improve over time. Results also indicated that Flutter matches and sometimes exceeds the Android SDK's performance.

Based on this research Flutter proves to be an excellent choice for developing Android apps, but developers need first to investigate if Flutter can do what they need. If not, then they need to decide if it is worth the effort to have to write code for the Flutter Engine or Embedder to incorporate that feature for the Flutter framework.

6.1 Future Work

Evaluate Other Metrics
As mentioned in Section 2.4 the key evaluation metrics are user experience, software platform features, performance, and security. This study evaluated two of these: software platform features and performance. However, as mentioned in Section 3.1 not

every software platform feature was tested and the features tested did not cover every possible aspect. For example, when evaluating animation, we drew a moving line on the screen; however, there are many other types of animations, for example, as transforming images or drawing other shapes. Not every possible performance metric could be tested, though as per Section 2.4, the two most important metrics were tested. Further research could include:

- User Experience
- Other Software Platform Features
- Other Performance Metrics
- Security

The Same Study for iOS Development
As Flutter also compiles to iOS, an additional study could compare Flutter with Native iOS development using Swift/Objective-C.

Other Cross-Platform SDKs
Flutter could also be compared with other SDKs that compile to either Android or iOS. One such SDK of particular note is Kotlin/Native [47] which compiles Kotlin code to iOS. Thus, a study could compare Kotlin and Dart for both iOS and Android development.

Bibliography

[1] Gartner. Gartner says worldwide sales of smartphones recorded first ever decline during the fourth quarter of 2017. https://www.gartner.com/en/newsroom/press-releases/2018-02-22-gartner-says-worldwide-sales-of-smartphones-recorded-first-ever-decline-during-the-fourth-quarter-of-2017. Accessed: 2019-01-20.

[2] Sensor Tower. Global app revenue grew 23% in 2018 to more than $71 billion on the app store and google play. https://sensortower.com/blog/app-revenue-and-downloads-2018. Accessed: 2019-01-20.

[3] Altexsoft. Pros and cons of xamarin vs native mobile development. https://www.altexsoft.com/blog/mobile/pros-and-cons-of-xamarin-vs-native/. Accessed: 2019-01-25.

[4] Netrananda Maharana. Cross platform mobile apps and its pros and cons - netrananda maharana - medium. https://medium.com/@netranandamaharana/cross-platform-mobile-apps-and-its-pros-and-cons-9c257ec64e94. Accessed: 2019-01-25.

[5] Medium. Why we are not cross-platform developers. https://medium.com/pixplicity/why-we-are-not-cross-platform-developers-fd7ef70e976d. Accessed: 2019-03-31.

[6] Flutter. Flutter - beautiful native apps in record time. https://flutter.io/. Accessed: 2019-01-14.

[7] Wikipedia. Flutter (software) - wikipedia. https://en.wikipedia.org/wiki/Flutter_(software). Accessed: 2019-01-25.

[8] P. Smutny. Mobile development tools and cross-platform solutions. In *Proceedings of the 13th International Carpathian Control Conference (ICCC)*, pages 653–656, May 2012.

[9] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. Comparing cross-platform development approaches for mobile applications. In *WEBIST*, 2012.

[10] Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. A survey and taxonomy of core concepts and research challenges in cross-platform mobile development. *ACM Comput. Surv.*, 51(5):108:1–108:34, November 2018.

[11] Skia. Skia graphics library. https://skia.org. Accessed: 2019-03-30.

[12] Flutter. Technical overview. https://flutter.dev/docs/resources/technical-overview. Accessed: 2019-03-30.

[13] Albert Staszak. Swift versus f# for ios development, 2018. Honours project report, School of Informatics, University of Edinburgh.

[14] Wenhao Wu. React native vs flutter, cross-platforms mobile application frameworks, 2018. Metropolia Ammattikorkeakoulu.

[15] I. Dalmasso, S. K. Datta, C. Bonnet, and N. Nikaein. Survey, comparison and evaluation of cross platform mobile application development tools. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 323–328, July 2013.

[16] Alex Sullivan. Examining performance differences between native, flutter and react native mobile development. https://thoughtbot.com/blog/examining-performance-differences-between-native-flutter-and-react-native-mobile-development-take-two. Accessed: 2019-02-24.

[17] Tim A. Majchrzak and Tor-Morten Grønli. Comprehensive analysis of innovative cross-platform app development frameworks. In *HICSS*, 2017.

[18] Android Documentation. Android developer documentation. https://developer.android.com/docs/. Accessed: 2019-01-13.

[19] Google. Google trends. https://trends.google.com/trends/?geo=US. Accessed: 2019-01-25.

[20] NPM. google-trends-api. https://www.npmjs.com/package/google-trends-api. Accessed: 2019-01-25.

[21] Android Documentation. Layouts. https://developer.android.com/guide/topics/ui/declaring-layout. Accessed: 2019-03-31.

[22] Google. Android fundamentals 02.2: Activity lifecycle and state. https://codelabs.developers.google.com/codelabs/android-training-activity-lifecycle-and-state/index.html?index=..%2F..android-training. Accessed: 2019-01-25.

[23] Google. Line charts | google developers. https://google-developers.appspot.com/chart/interactive/docs/gallery/linechart. Accessed: 2019-03-15.

[24] Dart Packages. Flutter package | webview_flutter. https://pub.dartlang.org/packages/webview_flutter. Accessed: 2019-03-15.

[25] Dart Packages. android-youtube-player. https://github.com/PierfrancescoSoffritti/android-youtube-player. Accessed: 2019-03-15.

[26] Dart Packages. Flutter package | youtube_player. https://pub.dartlang.org/packages/youtube_player. Accessed: 2019-03-15.

[27] Android Documentation. Androidx. https://developer.android.com/jetpack/androidx. Accessed: 2019-03-15.

[28] Android Documentation. Migrating to androidx. https://developer.android.com/jetpack/androidx/migrate. Accessed: 2019-03-15.

[29] Dan Lew. The reality of migrating to androidx. https://blog.danlew.net/2018/11/14/the-reality-of-migrating-to-androidx/. Accessed: 2019-03-15.

[30] Android Documentation. Volley overview. https://developer.android.com/training/volley/. Accessed: 2019-03-15.

[31] JetBrains. Anko - pleasant android application development. https://github.com/Kotlin/anko. Accessed: 2019-03-31.

[32] Android Documentation. Save key-value data. https://developer.android.com/training/data-storage/shared-preferences. Accessed: 2019-03-15.

[33] Dart Packages. Flutter package | shared_preferences. https://pub.dartlang.org/packages/shared_preferences. Accessed: 2019-03-15.

[34] Dart Packages. Dart package | http. https://pub.dartlang.org/packages/http. Accessed: 2019-03-15.

[35] Google. Introducing google play instant, a faster way to try apps and games. https://blog.google/products/google-play/introducing-google-play-instant-faster-way-try-apps-and-games/. Accessed: 2019-03-15.

[36] Mockito. Mockito. https://github.com/mockito/mockito. Accessed: 2019-03-23.

[37] Dart Packages. Dart package | mockito. https://pub.dartlang.org/packages/mockito. Accessed: 2019-03-23.

[38] Android Documentation. Espresso. https://developer.android.com/training/testing/espresso. Accessed: 2019-03-31.

[39] Github. Scrolling and tap not working in flutter driver. https://github.com/flutter/flutter/issues/16991. Accessed: 2019-03-23.

[40] StackOverflow. Stackoverflow trends. https://insights.stackoverflow.com/trends. Accessed: 2019-02-24.

[41] Rosetta Code. Rosetta code. https://rosettacode.org. Accessed: 2019-02-24.

[42] Android Documentation. Android profiler. https://developer.android.com/studio/profile/android-profiler. Accessed: 2019-02-26.

[43] Android Documentation. Android logcat. https://developer.android.com/studio/debug/am-logcat. Accessed: 2019-02-26.

[44] John McCutchan. Numeric computation. https://www.dartlang.org/articles/server/numeric-computation. Accessed: 2019-03-31.

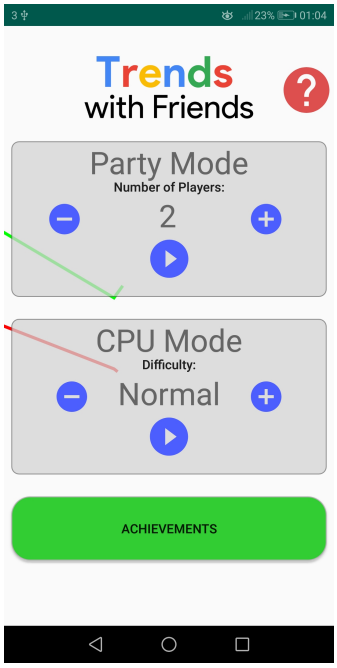
[45] Ivan Posva. Is there a performance benefit in using fixed-length lists in dart? https://stackoverflow.com/questions/15943890/is-there-a-performance-benefit-in-using-fixed-length-lists-in-dart. Accessed: 2019-03-31.

[46] Rosetta Code. Hamming numbers - rosetta code. http://rosettacode.org/wiki/Hamming_numbers. Accessed: 2019-03-31.

[47] JetBrains. Kotlin/native. https://github.com/JetBrains/kotlin-native. Accessed: 2019-03-31.

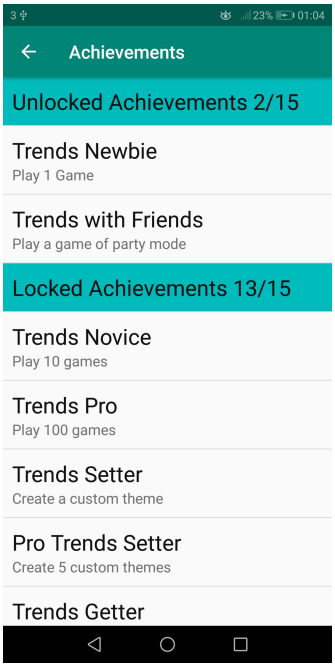
Appendix A

Native Trends Screenshots

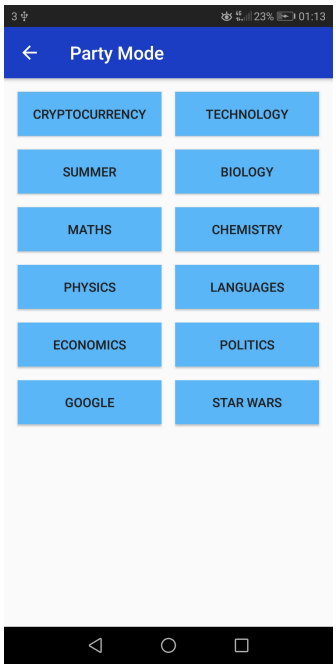


a) MainActivity

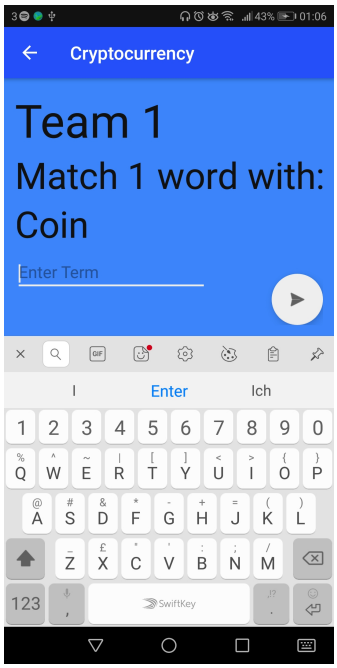
b) AboutActivity



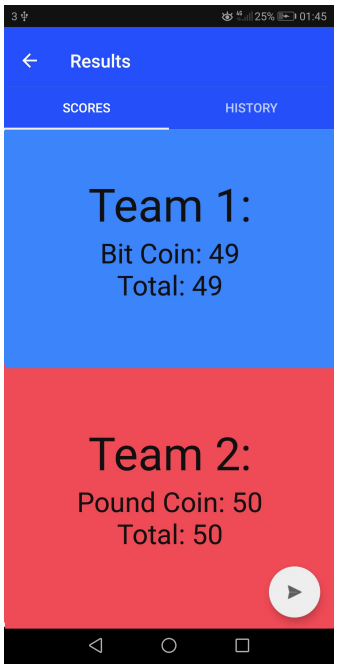
c) AchievementsActivity



d) ThemesActivity



e) GameActivity



f) ResultsActivity - Tab 1

g) ResultsActivity - Tab 2

h) FinalResultsActivity

Appendix B

Flutter AnimatePath Function

Dart - animatePath.dart

```
void animatePath(Canvas canvas, List<Offset> points, Color lineColor
    ) {
    var paint = Paint()
      ..color = lineColor
      ..style = PaintingStyle.fill
      ..strokeWidth = 4.0;
    //The distances between all points
    List<double> difsX = [];
    List<double> difsY = [];
    for (int i = 0; i<points.length-1; i++) {
      difsX.add(points[i+1].dx-points[i].dx);
      difsY.add(points[i+1].dy-points[i].dy);
    }
    int numOfLineChanges = 2*(points.length-1);
    int numOfLineChangesOver2 = points.length-1; //Needed as
        dividing ints converts them to doubles
    double timeBetweenLineChanges = 1/numOfLineChanges;

    ///For loop loops for half the number of line changes because
    ///half the line changes are creating, the other half is
        deleting
    for (int i = 0; i<numOfLineChangesOver2; i++) {
      ///Line Creation
      if (_animation.value < timeBetweenLineChanges*(i+1) &&
          _animation.value > timeBetweenLineChanges*i) {
        ///Draw Previously Created Lines
        for (int j = i; j>=1; j--) {
          canvas.drawLine(points[j-1], points[j], paint);
          canvas.drawCircle(points[j], 2.0, paint);
        }
        ///Line currently being created
        canvas.drawLine(
            points[i], //First Point
            Offset(
              points[i].dx+(difsX[i]*(_animation.value-
                  timeBetweenLineChanges*i)*numOfLineChanges),
              points[i].dy+(difsY[i]*(_animation.value-
```

```
                timeBetweenLineChanges*i)*numOfLineChanges),
            ), //Second Point
            paint //Color
        );
      }

      ///Line deletion
      if (_animation.value < timeBetweenLineChanges*(i+1+
          numOfLineChangesOver2) && _animation.value >
          timeBetweenLineChanges*(i+numOfLineChangesOver2)) {
        ///Draw Lines not deleted yet
        for (int j = i+1; j<numOfLineChangesOver2; j++) {
          canvas.drawLine(points[j], points[j+1], paint);
          canvas.drawCircle(points[j], 2.0, paint); //width was 5.0
        }
        ///Line currently being deleted
        canvas.drawLine(
            Offset(
                points[i].dx+(difsX[i]*(_animation.value-
                    timeBetweenLineChanges*(i+numOfLineChangesOver2))
                    *numOfLineChanges),
                points[i].dy+(difsY[i]*(_animation.value-
                    timeBetweenLineChanges*(i+numOfLineChangesOver2))
                    *numOfLineChanges)
            ),
            points[i+1],
            paint
        );
      }
    }
  }
```

Appendix C

Android XML Transitions

XML - slide_in_left.xml

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android" >
    <translate
        android:duration="500"
        android:interpolator="@android:interpolator/decelerate_quint
            "
        android:fromXDelta="100%p"
        android:toXDelta="0%p" >
    </translate>
</set>
```

XML - slide_in_right.xml

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android" >
    <translate
        android:duration="@android:integer/config_mediumAnimTime"
        android:interpolator="@android:interpolator/decelerate_quint
            "
        android:fromXDelta="0%p"
        android:toXDelta="100%p" >
    </translate>
</set>
```

XML - slide_out_left.xml

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android" >
    <translate
        android:duration="500"
        android:interpolator="@android:interpolator/decelerate_quint
            "
        android:fromXDelta="0%p"
        android:toXDelta="-100%p" >
    </translate>
</set>
```

XML - slide_out_right.xml

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android" >
    <translate
        android:duration="@android:integer/config_mediumAnimTime"
        android:interpolator="@android:interpolator/decelerate_quint
            "
        android:fromXDelta="-100%p"
        android:toXDelta="0%p" >
    </translate>
</set>
```

Appendix D

HTML for WebView

HTML - graph.html

```
<html>
<head>
    <script type="text/javascript" src="https://www.gstatic.com/
       charts/loader.js"></script>
    <script type="text/javascript">
    google.charts.load('current', {'packages':['line']});
    google.charts.setOnLoadCallback(drawChart);

    function drawChart() {

      var mdate = new Date();
      mdate.setFullYear(mdate.getFullYear() - 1);

      var allVals = JSON.parse(Android.getVals());
      var numVals = Object.keys(allVals).length;

      var i = 1;

      allVals.forEach(function(element) {
        element.unshift(new Date(mdate.getTime()));
        //i++;
        mdate.setDate(mdate.getDate() + 7);
      });

      var data = new google.visualization.DataTable();
      data.addColumn('date', 'Day');
      data.addColumn('number', Android.getQuery() + " " + Android.
         getTerms());
      data.addColumn('number', '2');

      data.addRows(allVals);

      //data.addRows(Android.getVals());

      var options = {
        chart: {
          //title: 'Box Office Earnings in First Two Weeks of
```

```
            Opening',
          //subtitle: "" + Android.getQuery()
          //subtitle: "" + allVals.toString()
          //subtitle: "" +  (mdate.getFullYear() - 1)
          //subtitle: "" + numVals
          //subtitle: 'in millions of dollars (USD)'
        },
        width: 900,
        height: 500,
        vAxis: { viewWindow: { min: 0, max: 100 } },
        //hAxis: { title: 'Hello', ticks: [5,10,15,20] }
      };

      var chart = new google.charts.Line(document.getElementById('
          line_top_x'));

      chart.draw(data, google.charts.Line.convertOptions(options));
    }
  </script>
</head>
<body>
<div id="line_top_x"></div>
</body>
</html>
```

Appendix E

Flutter Altered WebView Plugin

These code snippets only includes the methods that were altered or added and removes all comments. The highlighted text is the code I wrote.

Dart - webview_flutter.dart

```
class JavascriptChannel {
  JavascriptChannel({
    @required this.name,
    @required this.onMessageReceived,
    @required this.data
  })  : assert(name != null),
        assert(onMessageReceived != null),
        assert(_validChannelNames.hasMatch(name));

  final String name;

  final String data;

  final JavascriptMessageHandler onMessageReceived;
}

Set<String> _extractChannelData(Set<JavascriptChannel> channels) {
  final Set<String> channelDatas = channels == null
      ? Set<String>()
      : channels.map((JavascriptChannel channel) => channel.data).toSet();
  return channelDatas;
}

Map<String, String> _extractChannelMappings(Set<JavascriptChannel> channels) {
  final Map<String,String> channelMapping = new Map();
  if (channels != null) {
    for (var channel in channels) {
      channelMapping[channel.name] = channel.data;
    }
  }
  return channelMapping;
}

class _CreationParams {
```

```
  _CreationParams(
      {this.initialUrl, this.settings, this.javascriptChannelNames,
         this.javascriptChannelData});

  static _CreationParams fromWidget(WebView widget) {
    return _CreationParams(
      initialUrl: widget.initialUrl,
      settings: _WebSettings.fromWidget(widget),
      javascriptChannelNames:
          _extractChannelNames(widget.javascriptChannels).toList(),
      javascriptChannelData:
        _extractChannelData(widget.javascriptChannels).toList()
    );
  }

  final String initialUrl;

  final _WebSettings settings;

  final List<String> javascriptChannelNames;

  final List<String> javascriptChannelData;

  Map<String, dynamic> toMap() {
    return <String, dynamic>{
      'initialUrl': initialUrl,
      'settings': settings.toMap(),
      'javascriptChannelNames': javascriptChannelNames,
      'javascriptChannelData': javascriptChannelData,
    };
  }
}

  Future<void> _updateJavascriptChannels(
      Set<JavascriptChannel> newChannels) async {
    final Map<String, String> names2Datas =
        _extractChannelMappings(newChannels);
    final Set<String> currentChannels = _javascriptChannels.keys.
        toSet();
    final Set<String> newChannelNames = _extractChannelNames(
        newChannels);
    final Set<String> channelsToAdd =
        newChannelNames.difference(currentChannels);
    final Set<String> channelsToRemove =
        currentChannels.difference(newChannelNames);
    if (channelsToRemove.isNotEmpty) {
      _channel.invokeMethod(
          'removeJavascriptChannels', channelsToRemove.toList());
    }
    if (channelsToAdd.isNotEmpty) {
      Map<String,String> mapsToAdd = Map();
      channelsToAdd.forEach((String s) {
       mapsToAdd[s] = names2Datas[s];
      });
      _channel.invokeMethod('addJavascriptChannels', mapsToAdd);
    }
```

```
    _updateJavascriptChannelsFromSet(newChannels);
  }
```

Java - JavascriptChannel.java

```
class JavaScriptChannel {
  private final MethodChannel methodChannel;
  private final String data;
  private final String javaScriptChannelName;

  JavaScriptChannel(MethodChannel methodChannel, String data, String
      javaScriptChannelName) {
    this.methodChannel = methodChannel;
    this.data = data;
    this.javaScriptChannelName = javaScriptChannelName;
  }

  @JavascriptInterface
  public String getVals() {
   return data;
  }
}
```

Java - FlutterWebView.java

```
public class FlutterWebView implements PlatformView,
    MethodCallHandler {
  private static final String JS_CHANNEL_DATA_FIELD = "javascriptChannelData";

  FlutterWebView(Context context, BinaryMessenger messenger, int id,
      Map<String, Object> params) {
    webView = new WebView(context);
    // Allow local storage.
    webView.getSettings().setDomStorageEnabled(true);

    methodChannel = new MethodChannel(messenger, "plugins.flutter.io
        /webview_" + id);
    methodChannel.setMethodCallHandler(this);

    applySettings((Map<String, Object>) params.get("settings"));

    if (params.containsKey(JS_CHANNEL_NAMES_FIELD)) {
      registerJavaScriptChannelNames(
        (List<String>) params.get(JS_CHANNEL_NAMES_FIELD),
        (List<String>) params.get(JS_CHANNEL_DATA_FIELD)
      );
    }

    if (params.containsKey("initialUrl")) {
      String url = (String) params.get("initialUrl");
      webView.loadUrl(url);
    }
  }

   private void registerJavaScriptChannelNames(List<String> channelNames,
       List<String> channelDatas) {
```

```
    for (int i=0; i<channelNames.size(); i++) {
       webView.addJavascriptInterface(
                new JavaScriptChannel(methodChannel, channelDatas.get(i),
                    channelNames.get(i)), channelNames.get(i));
    }
  }
}
```