Expertise
and insight
for the future

Thanh Tran

Flutter Native Performance and Expressive UI/UX

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

04 April 2020

Author Title Number of Pages Date	Thanh Tran Flutter Native Performance and Expressive UI/UX 34 pages + 1 appendices 04 April 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Mobile Solutions
Instructors	Lasse Diercks, CEO Kari Salo, Principal Lecturer

Flutter Native Performance and Expressive UI/UX explores the technology behind Flutter highly performant and dynamic user interface system. The thesis aims to highlight the claims Flutter made about its technology and research those claims. One of the goals of the study is to compare Flutter application's visual and resource performance with existing technologies. The project was made in collaboration with Blidz, an e-commerce company hoping to improve its current Cordova client by migrating its implementation to Flutter. Before doing so, Flutter must be tested and compared to its Cordova counterpart by implementing a proof of concept.

The methodology chosen for this study was to utilize Flutter internal performance profiler as well as Android studio profiler to gather performance data from both Cordova client and Flutter proof of concept. As a result, the research would gain enough insights to determine the outcome: whether Flutter is worth redirecting resources to fully migrate from an old system.

The results obtained were satisfactory with some caveats. Flutter provides a significant improvement over its predecessor. Performance of Flutter constantly exceeds expectation while still delivering a dynamic and smooth UI/UX experience. In comparison to Cordova, Flutter proves to be a positive step towards the future for Blidz. However, the scale of the thesis does not reflect the current cross platform ecosystem where there are more established technologies. More research is needed before a concrete conclusion can be made to determine if Flutter is better than its competitors.

Ultimately, the thesis achieved its goal to profile the performance of the Flutter proof of concept. The study indicated that Flutter's performance is, in fact, better than that of Cordova considering its development and UI.

Keywords	Flutter, Cordova, UI/UX, performance, Dart

Contents

List of Abbreviations

PoC	Proof of Concept
UI	User Interface
UX	User Experience
SDK	Software Development Kit
AOT	Ahead Of Time
JIT	Just In Time
VM	Virtual Machine
UX	User Experience
XML	Extensible Markup Language
API	Application Programming Interface
OS	Operating system
APK	Android application package
BLoC	Business Logic of Component
Fps	Frames per second

1 Introduction

Mobile application development is always on the rise and more so in recent years when more and more people have access to smart devices. As such, the need for mobile development frameworks also increases significantly. Traditionally, native development using native technologies such as Swift and Kotlin/Java, for iOS and Android respectively, is preferred as they offer more functionalities, smooth integration and fast performance. The reason is that these technologies target and are supported by their respective platforms. Moreover, native development comes with a dedicated development environment (Android Studio and XCode) with a suite of tools. However, being platform-specific is also their drawback. Most business entities aim to reach as many users as possible, which means they need to develop applications for both platforms. Such a goal leads to the need to have two dedicated development teams working on the same project. Business entities have realized that the cost of developing and maintaining native applications for both platforms can be overwhelming. Therefore, cross-platform solutions have been introduced constantly. These technologies offer flexibility, financial stability and optimized workflow. Unfortunately, they come as a cost of the quality of the application, specifically performance.

Flutter, developed by Google, is the newest cross-platform technology that aims to eliminate the performance and user interface downfall of its predecessor. Flutter claims to have achieved native performance, easier implementation and expressive user interface and user experience. However, there are a few studies specifically aimed at profiling Flutter performance and its UI functionalities. The goal of this thesis is to take a look at the claims the Flutter team makes in terms of performance and UI/UX as well as how Flutter achieves those claims with its architecture and technology. More importantly, the thesis aims to analyze Flutter application performance and rendering in comparison with a similar application built in Cordova. The project is done in collaboration with Blidz. Blidz is a digital shopping company which offers a new online shopping experience. It does so by providing products along gamified models instead of a traditional pay-and-get model. Currently, Blidz is using a Cordova-based application for mobile and web clients. However, Blidz have noticed a significant drop in performance as the mobile client gets bigger in scale, which limits its UI potential. Therefore, the main goal of this thesis is to evaluate

if it is beneficial for Blidz, in terms of performance and UI/UX, to completely migrate its old client to Flutter.

2 Flutter and Dart

2.1 Cross-platform application SDK

Flutter is a User Interface (UI) toolkit developed and maintained by the Flutter team at Google. At its core, Flutter is a Software Development Kit (SDK) created to build native-performance application for mobile devices (iOS and Android) as well as for web and desktop application. Currently, Flutter for web is in beta version and desktop is in technical review stage, and therefore, this thesis will focus on the mobile side of Flutter. [1.]

Flutter for mobile development focuses on providing tools for developers to ship mobile applications that have the look and feel of native applications. Moreover, Flutter aims to do so by having developers interacting with a single codebase for any platform. Having learned from other previous technologies, Flutter boasts a rich set of functionalities as well as tools to, in theory, boost development efficiency. In later parts of the thesis, these tools and functionalities will be discussed to determine whether their benefits are better compared to other standards within the industry, such as React Native. [1.]

2.2 Dart overview

Flutter uses Dart as its main programming language. To understand why it was the first choice by the Flutter team, a deeper look into the core of Dart is needed. First of all, Dart is both Ahead Of Time (AOT) and Just In Time (JIT). According to Dart platform specification [2], during native development, Dart uses JIT and Virtual Machine to compile its code into native ARM and x64 machine code. JIT compilers are exceptionally fast because they compile during execution and. JIT allows for more development-friendly functionalities. One example is the Hot Reload feature of Flutter. Hot Reload takes advantage of JIT to almost instantly reflect changes in the code to the UI without resetting the current state of the application, which in turn significantly improves the development cycle and

allows for a smoother experience for developers. However, JIT is not ideal for production because JIT usually results in slower and segmented execution due to its nature of analyzing and compiling during the period, which would lead to negative user experience (UX). This is when Ahead Of Time (AOT) is needed. In AOT, codes are compiled before execution and while it typically increases development time significantly, AOT allows applications to have a smooth execution. Due to the nature of Dart, it makes use of both types of compilers, which leads to fast development cycle and performant production-level applications. [2, 3.]

Another advantage of Dart is that it is expressive and eliminates the need for layout languages like JSX in React/React Native or XML. Dart, combining with the richness of Widgets, creates a code interface that is easy to understand and visualize. [3, 4.]

Dart selects and combines functionalities and features of other commonly used languages such as Java, Swift, JavaScript and Kotlin. Dart utilizes async-await, interfaces, generics and, most notably in version 2.7, extension method. Given how Dart looks similar to many popular languages, it is easy to learn and understand, which in turn attracts more developers to Flutter or any technologies that utilize Dart. [4.]

2.3 Native performance

Performance has an incredibly broad definition. In general, performance refers to the perception of users of the application and how the application affects their experience. In a technical aspect, performance consists of visual performance, resource management (RAM, CPU and so on), input-output speed, network speed and more [5]. Due to the broad scope of performance, this thesis will focus solely on visual performance, which refers to the smoothness of the UI, and resource management, which refers to how resources are allocated.

According to the Flutter team, Flutter aims to and has achieved “native performance”. Native performance means to have a consistent frame rate similar to that of the specification of the target device. As Flutter team states, applications built with Flutter, given the correct development optimization, will reach 60 frame per seconds (fps) or 120 fps for devices that have 120Hz capability [5]. They also reach the CPU and RAM usage

comparable to native applications. In addition, Flutter applications have smooth UI and steady frame rates or there is no visible animation jank or frame drops. These claims are backed by the fact that Flutter uses Dart. As mentioned in the previous section, Dart utilizes both AOT and JIT for compiling. The advantage of AOT is that it avoids a communication bridge. Dart compiles directly into native machine code, which in turns avoids problems that other dynamic languages have [2]. This fact, in theory, boosts the performance of Flutter significantly. [5.]

Flutter UI tool kits consist of prebuilt UI components called widgets. These widgets are provided out of the box. The widgets represent common UI components seen in mobile devices like Lists, Buttons. The usage of widgets is crucial to the overall performance of the application. They are pre-optimized by the Flutter team and are recommended so they usually achieve a certain desired performance. For example, ListView widget only expose visible components while disposing non-visible ones to reduce resources needed to render them [6]. Such optimization contributes the overall performance of the application. Yet, those optimizations happen at the widget level. Flutter UI rendering is built upon a performance-focused foundation from the start, which surrounds widgets. [1.]

Flutter UI actually consists of three hierarchies co-existing and connected. They are: Widget tree, Element tree and Render Object tree [7]. While during development, developers usually only see and interact with widget, hence widget trees. Widgets are provided with configuration, properties on what it should contain or look like. When the app is run, or when a widget is built, Flutter creates three separate hierarchy as described above under the hood. Widget tree will receive and store properties passed. Element tree will determine the widget's location, manage its relationship (which widget holds it and which widget it holds) and most importantly, manage its state. Render object tree will render UI properties, sizing and painting the widget as well as laying out its children. The reason for such division is for separation of concern, which in turns leads to performance optimization [7]. It optimizes by analyzing old and new widgets after every rebuilding to determine which elements of those hierarchies need to be created, replaced or disposed of. Elements that pass the comparison will be reused. Such mechanics allow Flutter to only build what is necessary while reusing old elements, which in turn saving resources by not building abundant elements. The Flutter team helpfully demonstrates in their widget rendering video, at 18:05 minutes, that even after replacing SizedBox

widget with Padding widget, which have different runtime type and render type, their child (Image widget) is kept across rebuilding [7]. Flutter observes the changes and determine that only SizedBox widget changes so only it needs to be rebuilt and nothing else. Such behavior drastically improves performance and resource usage. [7, 8.]

2.4 User interface and user experience

2.4.1 End user experience

As discussed in the previous sections, the core building block of Flutter is widget. The main functionality of a widget is to describe a certain element visually in the UI given its configuration and state. On the development side, widget provides a handful set of application programming interface (also known as API) in an expressive manner for developers to visualize and configure widget in an expected manner. Flutter team also states that they take inspiration from React (which uses a similar concept called Component) while building Flutter. The native performance section discussed the performance optimization of widget. This section will focus on how a widget provides UI and follows UX guidelines.

In the current version of Flutter, two sets of UI design that the Flutter follow: Material Design and Cupertino. Widgets that follow Cupertino have the look at feel of iOS elements and are more suitable for iOS devices, although there are no strict rules or guidelines about mixing different styles. On the opposite end, Material design is built to inherit the style of Android elements. The two sets of guidelines are designed and tested by their respective founders which can be found in Material Guidelines for Material Design by Google [9] and Human Interface Guidelines by Apple [10]. For a brief overview, these guidelines provide conventions and guidance to make sure the application are visually fluent and pleasant. They also implement certain rules to help follow UX guidelines like typography, environment, layout, navigation and so on. While these guidelines simply guide developers on the best conventions, the fact that Flutter directly implements them into its widgets significantly reduce development time by removing the need for manually building custom widgets. One specific example is Card widget. As stated in the documentation, card follows Material to provide a sheet of information in an enclosure [11]. The documentation defaults elevation of card to 1 (the base elevation is 0), having it

floats above the background an amount states by Material design [9]. Elevation is only one of many attributes that card has which almost entirely eliminates the need for developers to manually design and create the widgets, hence shorter development time. One other benefit of having Flutter following a specific guideline is to ensure the consistency of the elements throughout the application [12].

While Flutter does provide two different sets of native UI widgets, generally, it mostly follows brand-specific design principle [12]. Brand-specific is a design pattern that prioritizes branding consistency. In the context of Flutter, it is generally easier and faster to create one consistent UI based on the branding of the application for all devices instead of separating the design according to the native OS of the device. One example of the latter case is the direct competitor of Flutter – React Native. While the comparison will be fully discussed in the following section, briefly, React Native renders the look of components (the equivalent of widget) based on which device is in use. Consequently, the same dialog box in the code will look differently on Android and iOS devices. In the case of Flutter, widgets render the same for both platform from one singular code. Such implementation allows developers and designers working with Flutter the ability to make a coherent, predictable application for all platforms to have brand-focused and brand-specific design. While this is not strictly a rule as Flutter does provide platform-specific widgets, it does require developers to have additional condition logic to render correctly to the devices. This opens up some potential problems which will be discussed in a later section. [13].

UX, on the other hand, is a more complicated matter than UI for cross-platform technologies like Flutter. According to ISO 9241-210:2019 definition of user experience:

> user's perceptions and responses that result from the use and/or anticipated use of a system, product or service [14]

UX is entirely based on human perception and is different case by case so the guidelines on UX are subjective. However, there are conventions that are generally agreed upon. For example, an application should allow users to navigate with ease, to read and understand content regardless of their condition, to minimize the amount of actions required to get a user to do a certain task, and so on. UX also considers application performance and the smoothness of actions. Flutter allows the ability to tackle these obstacles. Flutter

architecture optimizes performance greatly in theory. The technology also follows Material Guidelines and also implement it directly into widgets. These built-in UX optimization reduces time for specific UX design. However, Flutter can only tackle low-level UX concept. As stated before, UX is a case-by-case evaluation. UX also has to take into consideration dynamic aspects such as disability, demographics, and human emotion. Such aspects cannot be quantified and predicted by the Flutter team and the responsibility falls on the entity using Flutter. [15, 16, 17.]

2.4.2 Developer user experience

The previous section discussed how Flutter provides user-friendly experiences for end-users through its integrated system. This section will focus on how the Flutter tools affect developers when developing UI, logic and the overall experience with the Flutter SDK.

To help convey the points, the thesis will focus on two user survey done by the Flutter team: the Q1 of 2019 survey [18] and the Q4 of 2019 survey [19]. One of the key features of Flutter is the cross-platform implementation. Material and Cupertino sets of widgets for Android and iOS respectively are created to provide the look and feel of their platform. Material set with its well-documented and rich functionalities impressed developers with satisfaction rate at 88% for Q1 and reached 92.1% at the end of 2019. Cupertino widgets, however, did not perform as well with only 66% satisfaction rate in Q1, though Flutter team improved the system around Cupertino widgets, which boosted its rate to 71.2%. Flutter team did and continues to do really well in terms of UI design, functionality, documentation and performance for Material widgets, which is considered crucial for the majority of developers. [18, 19]

Another major, if not the most important, feature of Flutter is hot reload. In theory, this feature should attract more developers due to how Dart compiling its code, giving it an edge over native technologies. This is demonstrated in Figure 1 below.

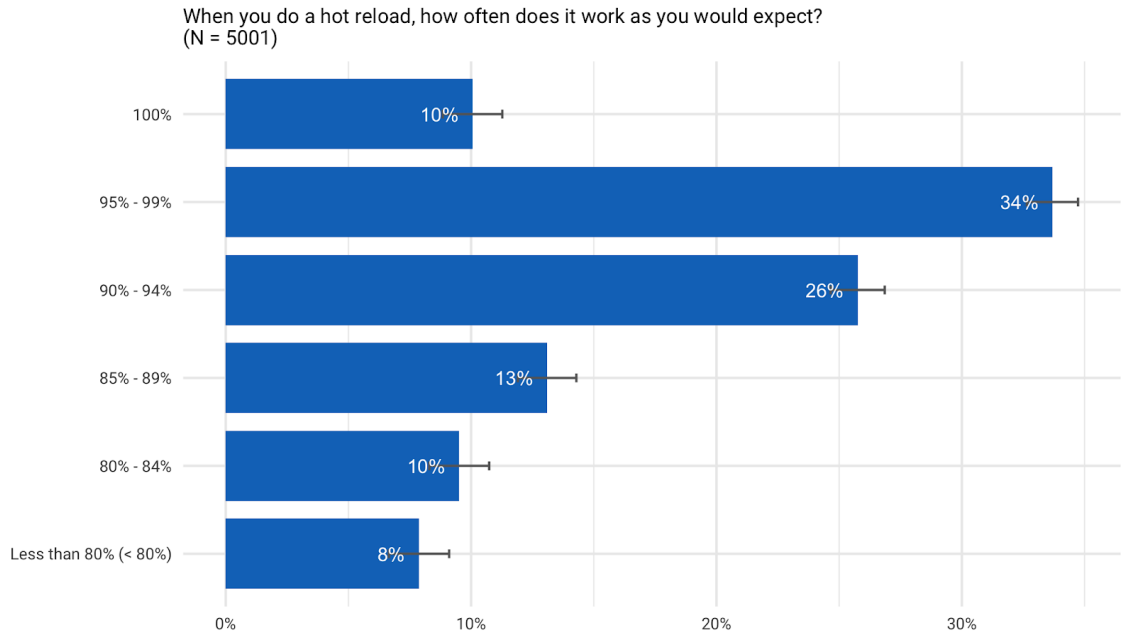

Figure 1. Hot reload working as expected statistics [19].

According to Figure 1, 70% of all respondents have seen hot reload behaving as it should 90% or higher of the time. One major point of feedback for hot reload is when it does not work, users have difficulty knowing why.

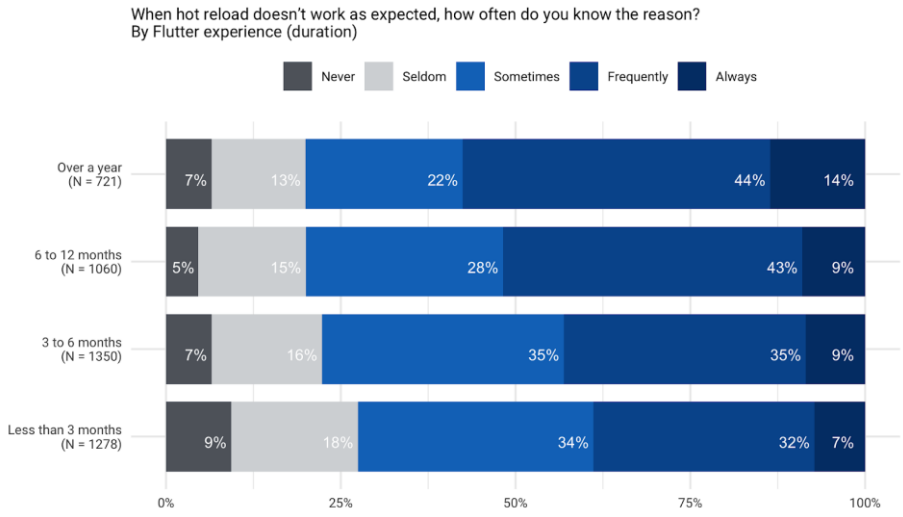

Figure 2. Hot reload failure statistics [19].

Figure 2 shows that at least 20% from all respondents seldom or never know why hot reload does not work. That is a notable issue the Flutter team needs to solve due to the

fact it can be a deciding factor for many developers looking to switch to or stay with Flutter. Most problems are known limitations but barely documented. Others are scalability issues or new bugs. A comparison with other cross-platform technologies. [19.]

Generally, developers are satisfied with Flutter. Flutter consistently maintains its satisfaction rate at 90% and above. In Q4 of 2019, the rate stays at 93%. Another notable increase is the number of respondents. In Q1 of 2019, only 1,961 people responds from the survey with 42% of that number only uses Flutter less than 3 months. Respondent number increases to 6,343. Although, one thing to take into consideration is how the surveys are being delivered. Another point is the Q4 survey correlates with the official first stable release of Flutter. There is no evidence to determine respondent increase is in correlation or in causation with the release. However, there is no denying Flutter is becoming more popular among developers given its stable 90% satisfaction rate and increasingly high 87,9k stars on GitHub as of this writing. [18, 19.]

2.5 A comparison with other cross platform technologies

Flutter is only one of the newest when it comes to cross platform technologies. Before Flutter exists Ionic, Xamarin, Apache Cordova and the most popular – React Native. In order to have an extensive and precise comparison, multiple frameworks need to be compared to Flutter plus a significant sample size must be examined. However, due to the scope and the general goal of the thesis, this section will only take into consideration React Native, which is the most prominent framework to date excluding Flutter, and Apche Cordova, which is the framework that is used for comparison in the project section of the thesis, and native technologies such as Kotlin and Java for Android and Swift for iOS as the base for all comparison. One thing to note is that React Native and Flutter compiles to native elements while Apache Cordova wraps the application in a web view to leverage the existing web code into mobile environment.

2.5.1 Advantages

One of the most talked about feature of Flutter is hot reload. Hot reload is crucial for speeding up development process and maintaining smooth user experience for mobile

developers. Hot reload on Flutter updates the UI in milliseconds while still keeping the previous state of the app. This allows a more linear and focus experience. Both Xcode and Android Studio, development environment for native mobile apps, have reload features but they take minutes to compile and often reset states. React Native also has hot reload of its own.

Performance is emphasized by Google when it comes to Flutter. Due to architectural-level performance optimization, Flutter, in theory, should boast native-level performance. However, Flutter lacks the appropriate study to back up this claim. Researching this subject can be particularly difficult because the researcher needs to have a large pool of test subjects and a controlled environment. An independent experiment was conducted by building the same application in five different platform, including native iOS, Android and React Native [20]. The experiment focused on CPU usage of the device. Flutter seems to be very economic and is comparable to native technology while React Native struggles. Another independent experiment dives deeper into resource usage of Flutter and React Native [21]. This experiment builds a timer application for Flutter, React Native, native and test it on multiple devices. Flutter, once again, impresses with its CPU usage similar to its native counterpart while React Native struggles. One interesting finding is that while Flutter excels with CPU usage, it falls into the same flaw as React Native: its memory usage is relatively high compared to native. However, Flutter still performed better than React Native – its direct competitor. So overall, this is a positive sign for Flutter going forward. It is worth to note that these experiments are not perfect and small in scale. [20, 21.]

Combining all advantages Flutter has, according to a survey on StackOverflow, Flutter is one of the most loved frameworks and is the most loved mobile development frameworks while React native, which one of the most mature and powerful libraries, falls behind and even appears high in the list of most dreaded along with Cordova. [22.]

2.5.2 Potential flaws

There are reasons why Flutter, with its increasing popularity, still boasts a relatively low application count. One immediate potential issue is how young Flutter is. Flutter only got announced and released in 2017 with its stable version released in late 2019. Due to the

age of Flutter, the number of third-party libraries can be limited and lack solutions for smaller but crucial issues. While there is no way to correctly determine the number of libraries, in comparison with React Native, there is a general agreement among developers that Flutter has a long journey to reach the same level of maturity. Even if the interest rate is high, the amount of active and constant users is still low compared to React Native. Moreover, JavaScript is a more widely used and familiar language than Dart, which contributes more or less on why developers hesitate to try Flutter. Fortunately, Flutter is becoming more prevalent so it is almost only a matter of time before it becomes mature enough.

Another potential issue is how Flutter handles cross platform UI. As stated in the UI/UX section, Flutter is using a design structure called “brand-specific”. This allows for a single code base to use either Material or Cupertino, or even both, to provide a consistent UI, tailored for brand design across platforms. The problem arises when the need goes beyond brand-specific design. One example is when an entity desires to have a cross platform application adhering to their brand but also wanting to differentiate itself between platforms. For instance, they want their dialog to appear Material on Android and Cupertino on iOS devices. This can be done in Flutter but with extra logic so if the application increase in scale, it also significantly increases the effort put into dividing logic. React Native, on the other, renders UI platform-specifically so there is no further effort needed. However, it is a debate whether this is an actual flaw, while it might just be a limitation at best. Anyhow, it is determined on a case-by-case basis and depends on the need of the developers.

Final point of interest is the Flutter application size. Cross platform frameworks generally produce higher size applications. This is due to the fact that when shipped, cross platform frameworks also include functionality libraries, framework configurations and more. Flutter suffers from the same problem. An article by Dharmin M compares minimal application between native technologies such as Kotlin and Java with React Native and Flutter [28]. The results show that for native tools, they only produced less than 1 MB of space while React Native and Flutter produced 7 MB and 7.5 MB respectively, which are 7 times higher than a native application. However, Flutter team consistently reduces the minimal app size as documented in [29]. According to the Flutter team, they have reduced the minimal app size to 4.06 MB for Android, which is a significant drop from 7.5

MB. Comparing this statistic to React Native, it is a win for Flutter. Nevertheless, it is still an issue for Flutter and for cross platform technologies in general.

3 Implementation of Flutter Proof of Concept

3.1 Introduction, design, libraries and tools

The thesis is written in collaboration with and under supervision of Blidz – an e-commerce company specializing in providing shopping experiences through a gamified model. Currently, Blidz is using mobile client built with Cordova. The Cordova client exhibits certain performance restrictions due to its nature of wrapping the whole application in a webview. Blidz chose to create a Proof of Concept (PoC) from Flutter to assess its performance and UI benefits compared to the old client.

Since PoC is meant to make an initial assessment whether Flutter is more performant, the project will only cover the most crucial functionalities yet will still represent the look and feel of the old client. PoC will contain 4 main views: Dashboard, DuoDeal, DealFreeze and FlashSale (see Appendix 1)

Dashboard, which is the landing view of the application, contains 3 main elements: animating banners, unlimited scrolling grid and a tab bar. Dashboard displays all products as well as promotion information. The goal of PoC is to allow Dashboard to scroll smoothly, to display smooth and automated animations for banners and to allow transitions. The next 3 views are grouped as one category called DealDetails. The DealDetails display specified information for a single product as well as providing various gamified functionalities and checkout. Each of these views have unique elements that are doing complex animations, which might slow down the performance of the application. DealFreeze has an ever-changing price point as well as an animating freeze button. DuoDeal automates displaying selected list of participants as well as their dedicated animated timer. FlashSale is least demanding out of the three with only an animated timer. All of the images of the views listed are design files and are not the exact implementation of PoC. Throughout the thesis, specific changes made to the design will be discussed if

they are due to performance issue or UI limitation of Flutter. The development also decided to use Material sets of widgets as they are more stable, have more variety over their Cupertino counterparts and falls in the demographic of the company, which is mainly Android users.

Aside from designs, certain decisions were made in terms of tools, development environment and libraries. The project was conducted using Visual Studio Code as it is one of the development environments supported by Flutter. Flutter and Dart SDK also contains a suite of helpful tools to aid development. One of such tools is the Dart DevTools which is used consistently throughout the process of development and post-development. It allows for elements inspection, hot reload and profiling. Along with Dart DevTools, Android Studio is used for performance profiling as it allows for profiling of built Android application package (APK), which is crucial for profiling Cordova application. In addition, it is more consistent to have the same set of data measured by the same tool for both the Cordova app and the Flutter app. For debugging and development, iOS simulator is the main tool as it is rather stable compared to Android emulators. Moreover, iOS devices cover more consistent, widely used and wide-ranging form factors. However, since the application is cross-platform, Android simulators are also occasionally used for testing.

In addition to development tools, a suite of libraries was also discussed about among the development team members. Flutter does not have a definitive state management system like native application. There are numerous state management systems developed by third-party developers and even the Flutter team themselves. Internally, Flutter provides InheritedWidget to propagate static data. However, the scope of the project exceeds capability of InheritedWidget. Some of the more prominent state management system are Business Logic of Component (BLoC), ScopedModel, Redux and Provider. Provider was chosen as the state management for this project. Provider is specifically documented by the Flutter team as their chosen method in conjunction with BLoC. It is also rather simple and flexible. In addition, Dio package is used in conjunction with http package for http requests.

3.2 Architecture and development

Due to the complications of Blidz application model, most business logic and backend communication were handled by development lead. The author was tasked to deliver UI elements as well as to incorporate real-time data into the UI. In addition, the author was also responsible for optimizing UI performance.

Since the design is dynamic and complicated in certain parts, UI development was done in view-by-view principle. The principle means development went through each individual view rather than implementing all minor widgets at once. With each view, the design was broken down into smaller elements to determine the amount and scale of widgets. In addition, elements needed to be analyzed to determine whether a custom widget was needed or a provided widget from the base package was sufficient enough. However, there was a general principle that prebuilt widgets were used if possible, to optimize performance as well as speed up production. Naturally, such principle can be difficult to follow given the flexible and dynamic nature of the design.

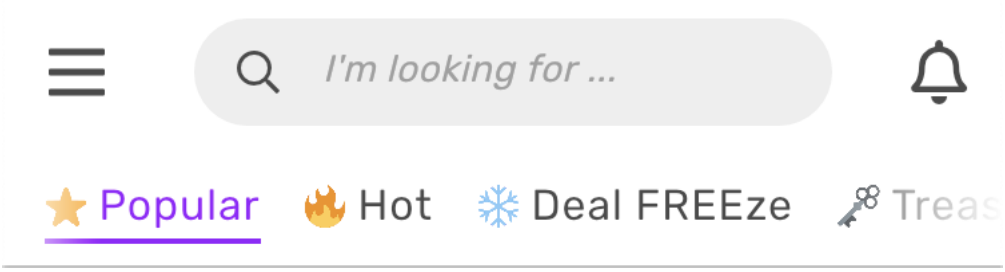

Figure 3. Header of Dashboard view

For example, figure 3 describes the header for Dashboard. Flutter provides a default AppBar widget and TabBar widget which allows for such design. However, further analysis found that the fading effect when scrolling and the icon preceding the text is not supported within the TabBar so a custom solution was chosen. Fortunately, Flutter also provides a mixin to allow for TabBar custom build.

Figure 4. Banner carousel in Dashboard view

Another example is the automated banners, which can be seen from figure 4, PageView widget from Flutter covers the needed behavior of the banners so no custom solution was needed. While most designs can be achieved given the appropriate amount of time but ultimately, the goal of PoC was to determine whether Flutter is as performant as other technologies. With such goal in mind, certain design choices have to be reduced or discarded altogether to preserve time resources and reach the goal as soon as possible

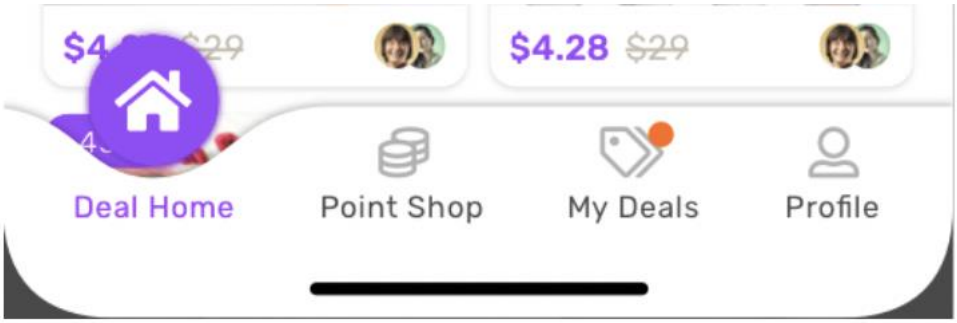

Figure 5. Bottom navigation bar in Dashboard view

Figure 5 shows the bottom navigating bar from the original design. The idea was to animate transition between tabs with advanced animation. Since Material widgets does not allow such customization, a custom was needed. However, such feature would take a few days to one week without any discernable benefits and could be a potential performance obstacle (due to animation). So ultimately, this design was set back to a traditional navigation bar until further development.

With the development process properly mapped out, the architecture of the project was also visualized. Figure 6 describes resource organization of the project:

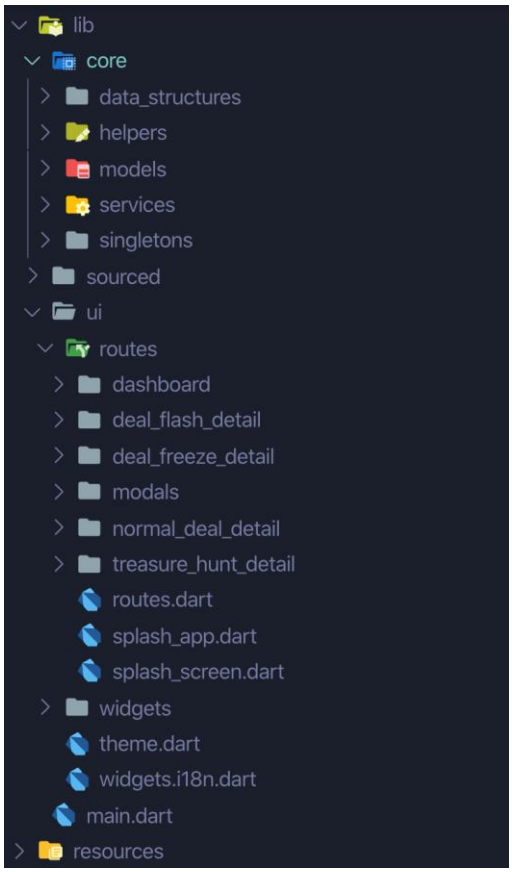

Figure 6. Folder structure of proof of concept

As figure 6 shows, UI logic and business logic are properly segmented with reusable widgets organized in one folder and individual views have their own folders. Models hold object states, which were implemented with Provider, along with Singletons to hold information for each session and backend communications. Overall, files organization is not that different from other cross-platform technologies like React Native and Angular with a heavy focus on reusable components and views.

3.3 Performance profiling

Performance profiling was done by two tools: Flutter Dart DevTools and Android Studio internal profiling tool. While profiling with Android OS was rather simple for both the Cordova application and the Flutter application, iOS was not possible for the Cordova application. The author has consulted with the development lead of Blidz and came to the conclusion that performance profiling for the Cordova application should only be done

on Android. The reason being the Cordova app was outdated and unable to update with the new requirement of Xcode. While such problem can be fixed, since development has now moved fully into Flutter, it was not recommended to spend resources on the old client. However, it should not affect much of the result since this is an analysis between Cordova and Flutter and not native and Flutter. Therefore, for the first performance test, the comparison was made only for APKs between Flutter and Cordova. For the second performance test, Android GPU profiler was used for Cordova-based app and Dart DevTools GPU profiler was used for the PoC. The reason being that Cordova cannot utilize Dart DevTools and the Flutter application was unable to display GPU metrics using Android GPU profile.

The first performance test was conducted using Android Studio profiling tools. For the Flutter application, a profile version was used. A brief description of versions of a Flutter application needs to be had. Flutter provides three types of build versions: debug, profile and release. Debug version focuses on debugging, has data collection turned on and compilation is optimized for development but it is not optimized for execution, which results in janky animations, low frame rates. This version is mostly used during development. Profile version is meant for performance profiling. This means that data collection is off except for essential information needed for profiling, code tests and similar processes are disabled to emulate real usage performance. Finally, release build is optimized for performance with fast execution, minimal footprint size and debugging data collection is completely off. Therefore, profile version is best used for performance profiling.

For the Cordova app, a minified test version that most resembles release application with all the profiling data still intact. The test was conducted on Galaxy S8+ device using Android version 9, One UI version 1, Exynos 8895 CPU and 4GB RAM. Android Studio profiling tools specifically measures resource usage of applications over an amount of time. There are four specifications in Android profiles, which are CPU usage (percentage), RAM usage (megabytes and gigabytes), Network usage (MB/s) and Energy (light, medium to heavy). The test was performed with the following process: Navigate to and stay in each view for exactly 20 seconds. During these 20-second periods, perform specific tasks for each specific view. For Dashboard, wait for banner animations, scroll, change tabs and click on the first item of each important category. For other views, swipe

the header thumbnails, wait for animations, scroll to the bottom of the information list. The whole process took 1 minutes and 40 seconds.

Figure 7. General performance graph of Cordova client

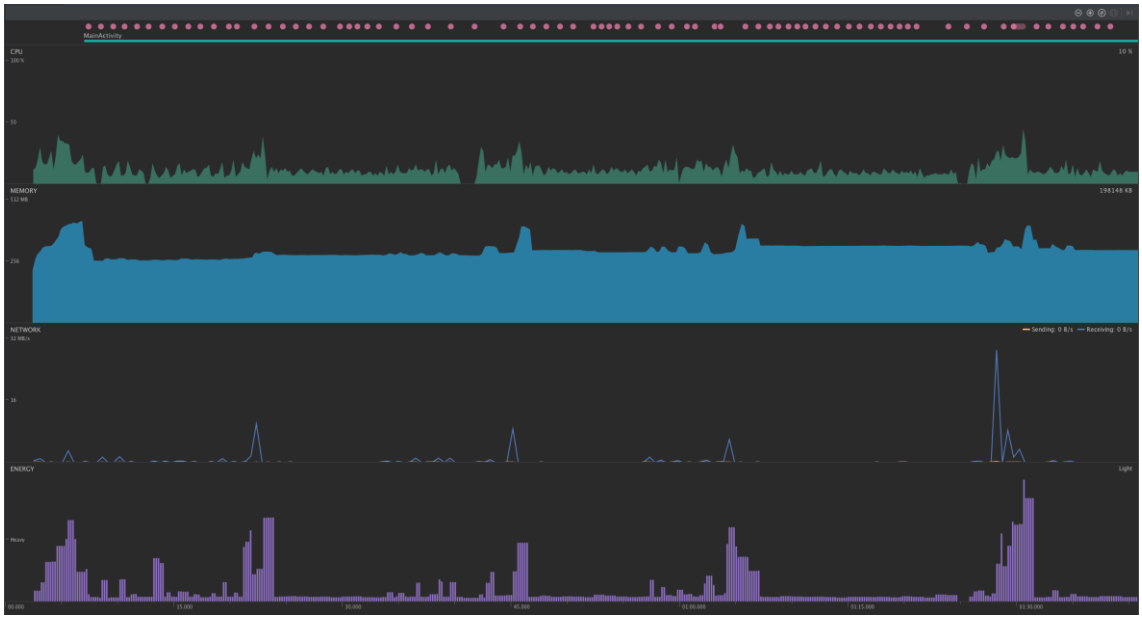

Figure 8. General performance graph of Flutter proof of concept

Figures 7 and 8 describe the overall performance data of the Cordova application and the Flutter application respectively. The graphs show all specifications and their totaled

data over the period of 1 minute 40 seconds. At the top, click events are showed by displaying red dots for clicks and red stripe for holds (like scrolling). The click events graph is not reflective of the actual events delivered so they are not considered in this performance test. The green graph is the CPU usage over time in percentage. CPU is responsible for handling executions and calculations within the application. The blue graph is RAM usage. RAM holds temporary information that the CPU needs to execute certain actions. If CPU and RAM usage is constantly high, the application will stutter. The purple bar graph is potential energy usage. High energy usage, which usually comes from high resource usage, reduces battery life. The line graph is network usage.

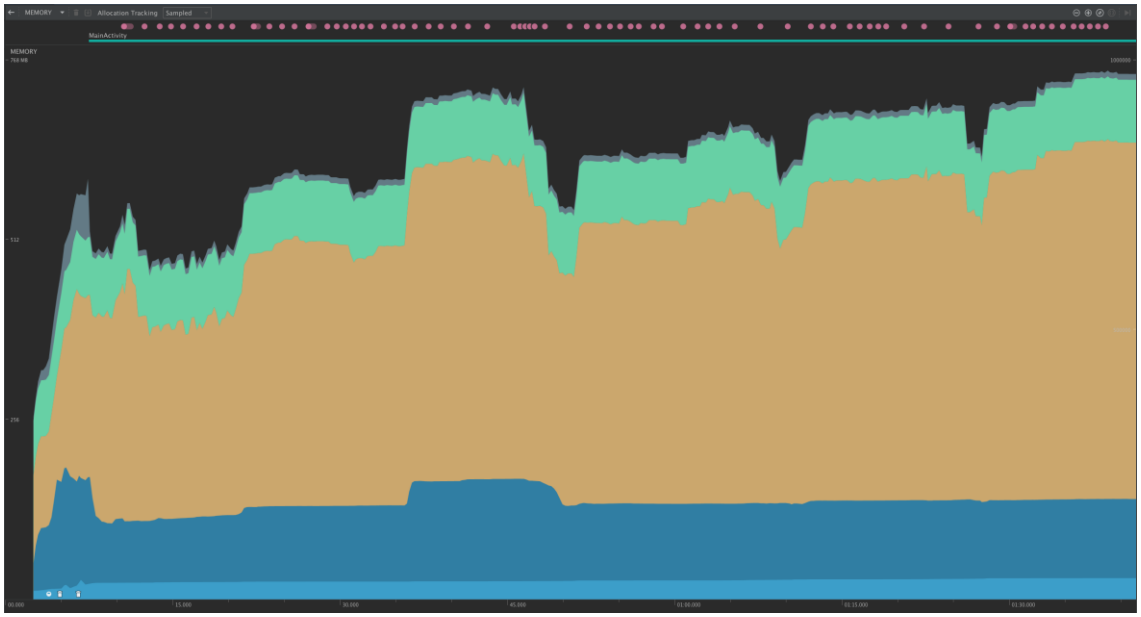

Figure 9. RAM usage graph of Cordova client

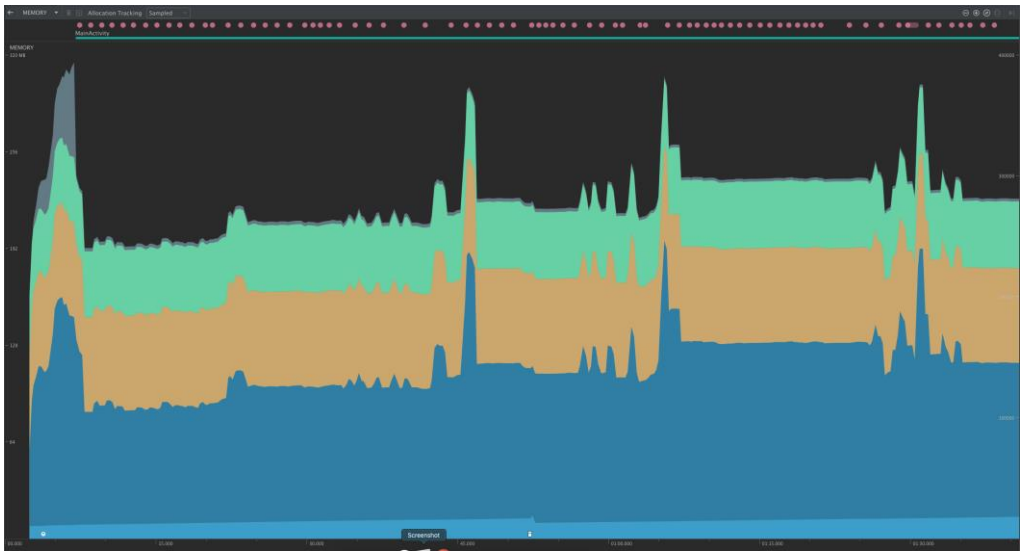

Figure 10. RAM usage graph of Flutter proof of concept

Figures 9 and 10 describe the detailed RAM usage data of the Cordova and Flutter application respectively. These graphs give a closer look at how RAM distribution is handled in each platform. While there are 6 categories of resources that utilize RAM, there are 4 that are notable in the graphs. They are native (blue), graphics (orange), code (green) and others (gray). Native is the memory from objects that are allocated from native code. This is prevalent because how different these cross-platform handles compiling codes into native languages. Graphics displays memory used for drawing UI. Graphics memory does not display GPU dedicated memory (if present). Code is memory dedicated to resources and codes such as fonts, images, libraries and so on. [25.]

Detailed graphs of CPU, Energy and Network do not provide any additional useful information so their general graphs will be sufficient enough.

The second test was done using the same hardware and software configuration as the previous test. The second test focused on the rendering of animations and widgets by observing frame rates. Flutter claims to achieve constant 60 frames per second (fps) which converts to 16ms per frame. This means that the application must render a frame at most every 16ms.

Figure 11. Frame rate graph of Flutter proof of concept

Figure 11 shows every frame rendered over a window of measurement. For each frame, there is a bar indicating the time it took to render. Each bar is a combination of the time CPU and GPU takes to render a frame. There is a line at 16ms mark. Any frame that is over that line will be colored red to indicate that it does not meet the standard.

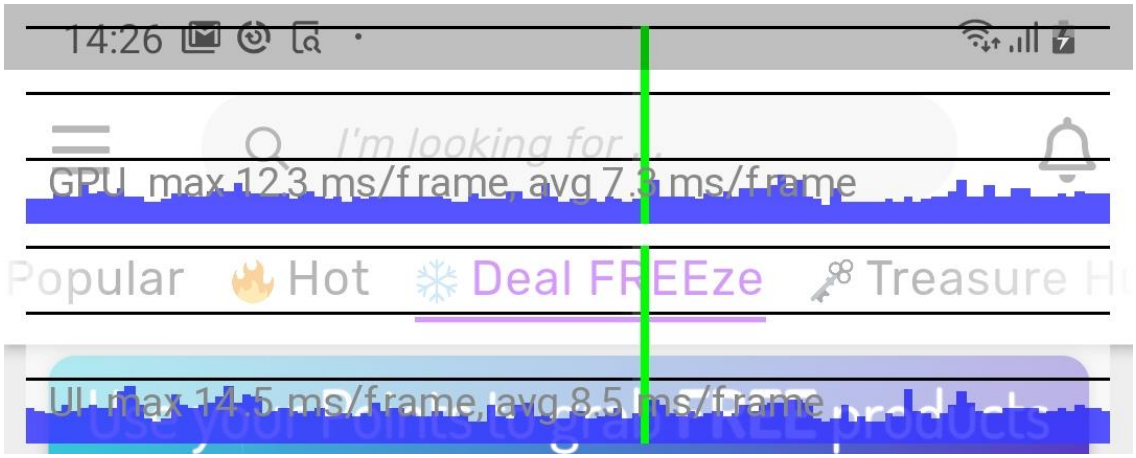

Figure 12. GPU and CPU graph overlay of Flutter proof of concept

Figure 12 is an overlay directly on the testing device provided by timeline tool. It shows max frame rates of CPU and GPU as well as their average frame rates.

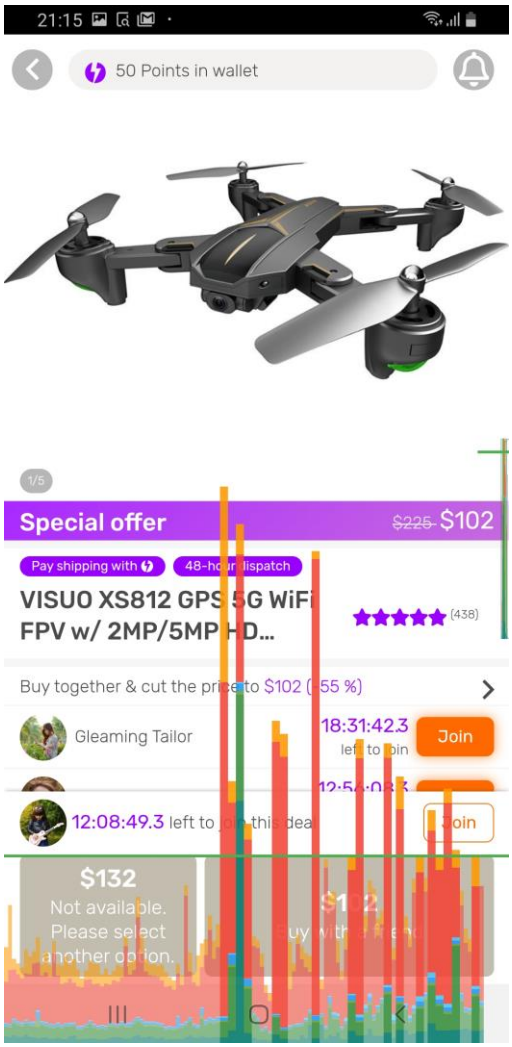

Figure 13. UI rendering overlay of Cordova application

Figure 13 illustrates the graphics overlay of the Cordova client. The overlay was the result of Android GPU profiler. The vertical bars represent rendering speed of each frames. The green horizontal line marks the 16ms point. Similar to Dart DevTools, any bars that exceeds the horizontal line does not meet 60 fps.

4 Proof of Concept Analysis

4.1 Performance

For the first test, Flutter exhibited promising results, although not without some red flags. In figure 7 and 8, Flutter performed better on all specifications compared to Cordova with the exception of CPU usage. RAM usage of Flutter overshadowed Cordova. Flutter used RAM in an average of around 180-200MB while Cordova averaged up to 500-600MB, which almost tripled in usage compared to Flutter. Note that Android Studio offers no way to calculate average numbers as well as highest numbers. These are based upon a series of data and calculated estimation. Peak RAM usage of Flutter reached only about 270MB of RAM while for Cordova, that number peaked at 638.1MB. A closer inspection of RAM usage according to figure 9 and 10 shows that there is a significant difference of how RAM is distributed among its categories.

For Cordova, heavy RAM usage is seen in graphics category, which contributed to its unusual high numbers. Graphics took up 229.4MB of RAM in the beginning and consistently increases without any sign of reducing the further into the application and peaked at 431.1MB. Interestingly, Flutter managed to keep a static graphics RAM usage of 53.8MB across 1 minute 40 seconds. Low graphics RAM usage might be due to the fact that Flutter utilizes Skia 2D graphics rendering engine. With Skia, most UI drawings use dedicated GPU resources instead of RAM, which put less strain on RAM. In Flutter, native is the category with the most RAM used which fluctuates around 70-100MB but even so, it is comparable with 80-100MB of Cordova native. RAM usage in both spiked significantly when there was a screen transition but Flutter managed to stabilize rather quickly and kept the numbers consistent while Cordova increases. The biggest spike in numbers from both platforms was when the app first launched where resources were loaded. Overall, Flutter did an extremely well job in keeping RAM usage low.

In Android Studio profiler, energy consumption data is an estimation based on resource usage rather than directly measuring energy. While it might not show real-time energy consumption, it does give a general view of how each platform handles their resources, which in turn translate to energy. The profiling also saw low energy usage across the board for Flutter with occasional spikes. The graph consistently shows below light level

energy consumption but sometimes jumps to heavy level. The graph also shows how energy spikes directly corresponded to spikes in CPU, RAM and network, which was due to screen transitions and active click events. In comparison, Cordova found it difficult to stay under medium level and at the same time, does not seem to correspond with click events. Finally, CPU consumption is the most unexpected data. While peak CPU usage of Flutter only reached 44.2% while Cordova reached 50.1%, on average, Flutter performed poorly compared to Cordova, although only by a small margin. Cordova consistently kept CPU usage below 10% with only a few spikes above but Flutter struggled to keep CPU usage below 10% and consistently went over 15%. This result was unexpected because Flutter was thought to have better CPU utilization. However, noted by the development lead of the project, one of the views were not fully optimized, which might lead to memory leakage, resulted in high CPU usage. Generally, the difference was not as marginal as RAM usage so overall, Flutter performed exceedingly better than its Cordova counterpart.

The second test took a look at how Flutter handles rendering frames. To reiterate, Flutter team indicates that Flutter can achieve native-like performance, which means constant 60fps on most devices and even 120fps on high refresh rate devices. Therefore, for this test, every frame must render on average at most 16ms and without any visible animation jank at any point of time. As figure 11 shows, Flutter had a constant below 16ms rendering time for every frame. There were a few noticeable spikes due to view transition and data acquisition. Overall, however, the Flutter client seem to handle well under even the most demanding view. A deeper look into data in Figure 12 shows that on average, GPU took on 7.3ms per frame and CPU took 8.5ms per frame.

Together, the whole system took 15.8ms per frame on average, which coincided with the claim from the Flutter team that Flutter can achieve native-like performance on average. However, there were some noticeable spikes. As the graph indicates, there was a significant drop in frame rate at some point, which reached up to 28.8ms that frame. There was most likely due to the current implementation and not because of Flutter. In contrast, figure 13 shows that the majority of the view rendering exceeds 16ms per frame and struggle to stay below that mark for most of the test. In general, Flutter did an excellent job at ensuring 60fps throughout the testing.

4.2 Ease of implementation

The development of the PoC took approximately 2 months from the concept until finalizing and testing. That period was short and fast even though PoC was developed for both iOS and Android. One of the major advantages of Flutter is the ability to build a Minimal Viable Product (MVP) and this project greatly demonstrated that ability. From the beginning, little setup was needed compared to other technologies. Flutter SDK provides all of the prebuilt package, ships along with a suite of useful tools without much need for configuration.

Hot reload is one of the most discussed features of Flutter. After constant usage, hot reload did significantly increase development time though not without some caveats. Throughout the first half of development, hot reload worked as it should with every change immediately reflected on the screen. It had an exceptional level of reliability. Whether the change was UI based or logic based, hot reload delivered consistently. However, towards the end half of development, it showed signs of weakness. As the project grew, hot reload started to struggle in certain views, especially in product views where it refused to work. The problem with hot reload was not that it was inconsistent but rather it was uncommunicative. Most of the time it did not work, the development team failed to understand why. While Flutter documentation is great for the most part, documentation for hot reload seems lack luster. It was rather clear that the error was from the implementation of the view but being unable to identify the problem eventually led to speculating hot reload itself. Such problem might not occur to other projects or at all. Overall, hot reload is reliable but only under heavy monitoring and care.

Widgets and Dart are two other parts of Flutter that were praised by the community. While this section will not specifically discuss the UI/UX of widgets, which will be in the next section, it will analyze how easy widgets are to use. Documentation of widgets are excellent and well-maintained by the Flutter team. Within the development environment, VSCode Intellisense combined with Flutter SDK provides a smooth experience when it comes to prebuilt widgets. They provide live documentation displaying available properties as well as their use.

```
child: Column(
  mainAxisSize: MainAxisSize.min,
  children: <Widget>[
    Flexible(
      flex: 6,
      child: Stack(
        children: <Widget>[
          _buildThumbnail(),
          _buildDiscountTag(context),
          _buildIndicatorElements(context),
        ], // <Widget>[]
      ), // Stack
    ), // Flexible
    Expanded(
      flex: 1,
      child: Padding(
          padding: EdgeInsets.all(kSpacing8),
          // child: _buildNormalBottom(context),
          child: _buildNormalBottom(context)), // Padding
    ), // Expanded
  ], // <Widget>[]
), // Column
), // InkWell
), // Card
); // VisibilityDetector
```

Figure 14. Sample snippet of Flutter Intellisense

Due to the nature of Flutter, UI elements can be highly nested with deep hierarchy. The VSCode Flutter extension handily documented brackets and scopes accordingly as demonstrated in Figure 14. Moreover, Dart is a highly descriptive language which blends comfortably with widgets and removes any need for layout language. Dart also shares a lot of functionalities and features with other languages like JavaScript, Java and Swift, which allows for easy transitions from other platforms. However, widgets and Dart are not without their flaws. The fact that they are descriptive and hierarchical can lead to extremely deep widget tree. It can be overwhelming when debugged even with the useful widget tree tool from the SDK.

One of the predicted downfalls of Flutter is community support. While Flutter gained itself a large following, the fact that Flutter is young provides a challenge. During development, there were times where a package was needed. Such package is widely used for other well-established platforms such as React Native and Ionic but are not yet available for

Flutter. One example is the FirebaseUI. FirebaseUI provides an interface for authentication of other services like Google, Apple, email and so on. It has support for web based, Kotlin and Swift. Even though Firebase is a part of Google, which is similar to Flutter, Flutter is still not supported by FirebaseUI. Although authentication is not a part of PoC, it will be a crucial feature in the future. The fact that no obvious authentication package is available creates a problem that is not hard to solve. While the community has evolved in a fast pace, it has not kept up with the demands.

Dart DevTools was crucial for the development of this project. Inspector tool is the most reliable aspect of Flutter. Inspector provides a detailed description for each element on screen. It also comes with a variety of debugging tools like slow animations to carefully analyze an animation cycle, debug paint to see hidden information right on screen. Along with inspector, Dart DevTools also provides CPU, GPU and Memory profiler along with hot reload, restart and logging.

Overall, the experience developing the PoC was improved significantly due to attention to detail of Flutter. The whole process took 2 months developing for both Android and iOS, which would have been much longer if the development is split into native Android and native iOS. Most of the problems faced were solved either by the rich documentation from the Flutter or from the ever-growing community. There were a few systematic problems that was immediately addressed on the Github issue page of Flutter and solved during SDK updates.

4.3 UI/UX

The final version of PoC achieved the design with few compromises. Most compromises came in the form of time management issue rather than because of Flutter. One advantage of Flutter over other technologies is the brand-specific design principle. This falls in line with the project design as it aimed to have a unified look across all platforms. The design was also influenced by Material Design guidelines. Flutter already included a set of Material widgets to use so the process of integrating the design into Flutter was rather smooth.

The original design specified some seemingly complex and heavy animations. One such example is the Deal Freeze category in the Dashboard

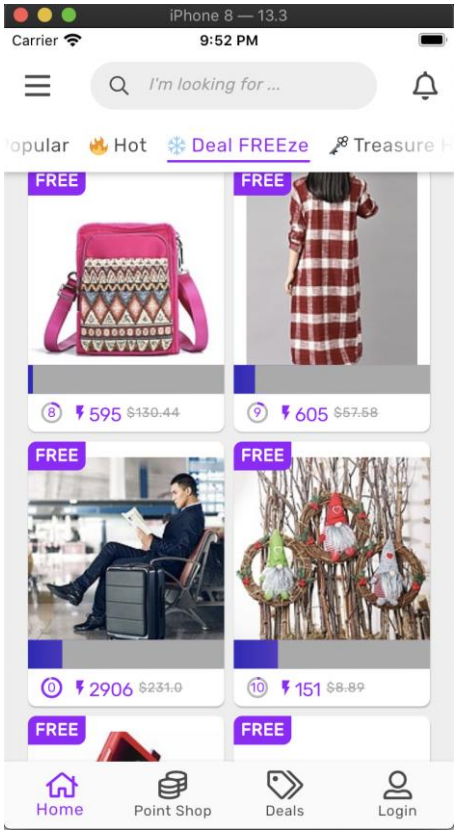

Figure 15. Deal freeze category in Dashboard view

Figure 15 shows how each of the deals are laid out in card form. Each of these cards has a timer which constantly updates every frame. Such operation does not pose much of a performance threat to each individual card but as a whole view where there are unlimited number of cards, the view will normally have a performance hit. However, the GridView automatically handles optimization by disposing non-visible elements. Therefore, no further optimization is needed from developers to achieve a constant 60fps. In Cordova, such optimizations must be done manually or use an external package.

Hero widget is another example of Flutter shortening the development process. The idea behind the product cards was that they will lead to their respective product detail view. The design specified that each card will expand into the new view. Such transition is

common among mobile application. Flutter provides that out of the box under a widget called Hero.

```
return Hero(
  tag: widget.deal.id + "_tag",
  child: Scaffold(
    body: Center(child: CircularProgressIndicator()),
    appBar: AppBar(),
  ), // Scaffold
); // Hero
```

Figure 16. Sample snippet showing Hero widget

Developers only need to wrap Hero around two connecting widgets or views, provide them with a key and Flutter automatically animates the transition. The animation is also optimized within Hero so there is minimal performance footprint.

There are a lot of instances of Flutter providing flexible widgets to ease the development strain. However, compromises needed to be made to adhere to the design choices. One of them was the tab bar.

Figure 17. Tab bar in Dashboard view

As figure 17 shows, the design specified the tab bar to have a fading effect while scrolling. The tab bar widget provided by Flutter does not allow for such implementation so a work around was decided to achieve the effect. Another widget called ShaderMask wrapped around the tab bar and added a shader to emulate fading. While this effect was ultimately achieved, it added another layer of performance strain as the ShaderMask is very demanding. While in Cordova, since tab bar is built manually from the beginning, such effect can be achieved with CSS without as much performance strain.

Another potential performance compromise was made to follow the design was the live timer.

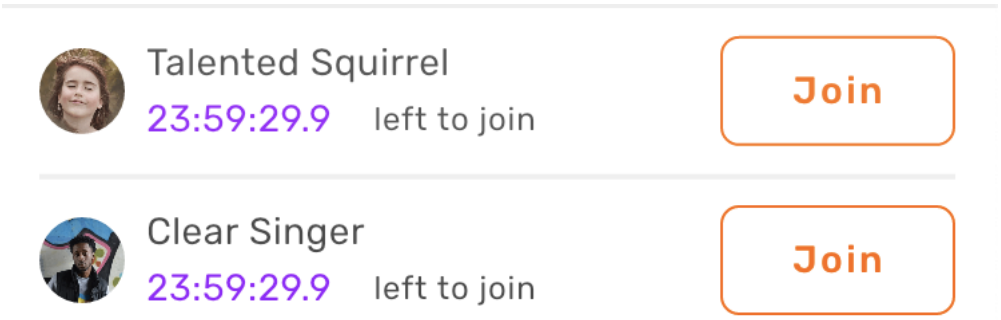

Figure 18. Participant list in DuoDeal view

The timers in figure 18 updates constantly as they count to milliseconds. This is achieved by using an animation object called Tween and AnimatedContainer. The way they work is that they check for updates every frame and either change the UI if there is an update or idle if there is not. These widgets are already optimized by idling but because these timers update so constantly that the optimization is non-existent.

Other than a few minor setbacks, the final PoC UI turned out almost exactly like the design. The theming feature of Flutter also prevents development from having hard coded values for styling and allow for a more consistent look. One key point that put the Flutter client over the Cordova client is that Flutter uses Skia rendering engine. Since most of its UI drawings are performed by the GPU, the UI constantly stays at 60fps which allows Flutter to provide much more leeway when it comes to design choices. For example, shadows and border radius performance footprint is non-existent in Flutter while Cordova sometimes struggle. Since PoC is an internal project, most of the UX testing was done by team members. Given that the development strictly follows Material guidelines, there are no foreseeable problem when it comes to UX.

4.4 Comparison with existing application

After evaluations from the previous sections, Flutter client is a definite improvement on the old Cordova client. In terms of resource utilization, Flutter is much better in handling

RAM and energy optimization. Specifically, Flutter propagate graphics handling to GPU, which take off much of the strain from RAM because of UI rendering and in turns resulted in one third of RAM usage compared to Cordova. Moreover, because of how optimized Flutter is when handling resources, energy consumption also saw a significant drop compared to Cordova. While CPU consumption was not as low as Cordova, the difference was not significant enough to be crucial. UI rendering from Flutter also saw a major improvement over its Cordova counterpart. Scrolling and animation were visibly smoother and less janky in Flutter. Since Skia graphics renderer from Flutter directs resources toward GPU, visual performance was greatly improved. Frame rates constantly stayed at 60fps with a few drops during heavy operations and transitions though was still impressive compared to Cordova. Moreover, Flutter and its powerful rendering engine gives more leeway in terms of design choices with minimal compromises.

The same cannot be stated for Cordova since it required much compromise to ensure an acceptable visual performance. On the development side, Flutter also improved the process significantly. Flutter SDK provides a suite of impressive tools out of the box. Specifically, hot reload, inspector and its incredibly accurate auto fill were some of the tools that greatly speed up development. Documentation from the Flutter team was one highlight throughout the whole experience. It was detailed and structured. Documentation of widgets not only lists and describes each property but it also provides examples for most common cases. Such documentation is often underrated and negated in other technologies. React, for example, is under scrutiny for its subpar documentation.

However, Flutter is not without its flaws. Cordova – being a much more mature technology – has a well-supported ecosystem. It means that Flutter has significantly less libraries and packages. While support for Flutter is increasing at a rapid pace, in niche and specific use cases, Flutter sees less solutions and choices. Such downside would be solved with time as more developers come to Flutter but, in the meantime, Flutter will need to cover for its lack of support. In addition to lack of support, Dart is also a point of weakness. Throughout this thesis, Dart is described as a great language with positive performance impact as well as being easy to adapt to and learn. However, compared to JavaScript, one of, if not, the most well-known and well-established language, Dart still needs more capabilities to convince developers to switch over. It is more so as React

Native, which is the direct competitor of Flutter, and Cordova both use JavaScript as their main language.

The point of this thesis, however, is to determine whether Flutter is more performant, easier to provide smooth UI experiences and easier to develop than Cordova. Based on the data gathered from the previous section, the client from Flutter is a definite improvement over its predecessor in all aspects. Although it is to note that such comparison is subjected to opinions. The nature of two technologies is rather different. While Flutter is a cross-platform mobile technology which utilizes its own rendering machine to draws its widget and compile its code directly to native code, Cordova is more web-based as it wraps its views in a web view. Therefore, from the architecture, Flutter already has an edge over Cordova when it comes to performance. Moreover, in order to reach a definitive conclusion, a wide scale research needs to be had. More performance straining functionalities must be observed such as Bluetooth, background services. The research mush also covers more devices from both Android and iOS under a controlled environment. The author does not have sufficient resources to conduct such throughout research. In that sense, it is inconclusive from this evaluation that Flutter is the most performant cross-platform technologies in the field. However, in the scope of this thesis, Flutter is better performance and UI/UX wise than Cordova.

5 Conclusion

The purpose of this thesis was to analyze the technology behind Flutter which contributed to its native performance and expressive UI/UX. In addition, the thesis aimed to compare and profile the Flutter proof of concept and its Cordova counterpart. As a result, the study was expected to derive a definite conclusion based on the data gathered from the comparison whether Blidz would gain benefits, from a development standpoint, from migrating its old Cordova application to Flutter. Moreover, the thesis was expected to provide a pointer to decide whether Flutter, as a cross-platform technology, is better than its current, more mature competitors such as React Native, Ionic and Cordova.

During the theory research, there seems to be an ongoing debate whether Flutter is ready to replace existing competitors, or even replace native technologies. By going

through actual development and migration of a production level application, Flutter does have a positive impact on the whole development process. It was faster to debug, faster to develop and faster to test. After performance testing, Flutter has an edge over Cordova. Resources are more optimized. Frame rate is constantly at 60fps while still maintaining a healthy energy consumption. Animation is consistently smooth. Moreover, the vast widget support from Flutter base SDK allows for more design freedom. Little compromises need to be had in order to adhere to specific design choices.

While Flutter performed great in the scope of this thesis, it is rather inconclusive whether Flutter is actually better than other technologies. The thesis only covers Cordova and Blidz old client when there are various established technologies such as React Native and Ionic. Flutter has its flaws that cannot be fixed easily. The tests were small in scale and did not reflect the market correctly. However, it is safe to assume that the future looks bright for Flutter. With time, Flutter will gain enough traction and support within the developer community. Flutter is also being backed by Google and its vast resources.

In conclusion, this thesis proved that in many ways, Flutter proof of concept improves performance, development process over the Cordova client of Blidz. It also proves to be highly beneficial and future-proof. While the decision is solely placed upon Blidz, Flutter is a positive choice moving forward.

References

1 Flutter [online]. Technical overview [cited 2020 February 16]. Available from: https://flutter.dev/docs/resources/technical-overview

2 Dart [online]. Platforms [cited 2020 February 16]. Available from: https://dart.dev/platforms

3 Flutter team. Flutter: the first UI platform designed for ambient computing [online]. 2019 December 11 [cited 2020 February 16]. Available from: https://developers.googleblog.com/2019/12/flutter-ui-ambient-computing.html

4 Dart [online]. Dart. [cited 2020 February 16]. Available from: https://dart.dev/

5 Flutter [online]. Flutter performance profiling [cited 2020 February 17]. Available from: https://flutter.dev/docs/perf/rendering/ui-performance

6 Flutter [online]. ListView class [cited 2020 February 16]. Available from: https://api.flutter.dev/flutter/widgets/ListView-class.html

7 Flutter [online]. How Flutter renders widgets [cited 2020 February 17]. Available from: https://www.youtube.com/watch?v=996ZgFRENMs

8 Flutter [online]. Widgets introduction [cited 2020 February 17]. Available from: https://flutter.dev/docs/development/ui/widgets-intro

9 Google [online]. Guidelines [cited 2020 February 23]. Available from: https://material.io/design/guidelines-overview/

10 Apple Inc. [online]. Human Interface Guidelines [cited 2020 February 23]. Available from: https://developer.apple.com/design/human-interface-guidelines/

11 Flutter [online]. Card class [cited 2020 February 23]. Available from: https://api.flutter.dev/flutter/material/Card-class.html

12 Michael T. Building beautiful, flexible user interfaces with Flutter, Material Theming, and official Material Components (MDC) [online]. 2018 May 10 [cited 2020 February 23]. Available from: https://medium.com/flutter/building-beautiful-flexible-user-interfaces-with-flutter-material-theming-and-official-material-13ae9279ef19

13 Bartosz S, Agnieszka M, Damian W. Flutter vs. React Native – What to choose in 2020? [online]. 2019 December 11 [cited 2020 March 7]. Available from:

https://www.thedroidsonroids.com/blog/flutter-vs-react-native-what-to-choose-in-2020

14 International Organization for Standardization. ISO 9241-210:2019. Ergonomics of human-system interaction — Part 210: Human-centred design for interactive systems [cited 2020 March 7]

15 Kati K, Tommi M. On Designing UX for Mobile Enterprise Apps. 2014 August [cited 2020 March 7].

16 Lai-Chong EL, Virpi R, Marc H et al. Understanding scoping and defining user experience: A survey approach. 2019 April. [cited 2020 March 7].

17 Marianna O, Lai-Chong EL, Virpi R. User experience evaluation methods: Current state and development needs. 2010 January [cited 2020 March 7].

18 JaYoung L. Insights from Flutter's first user survey of 2019 [online]. 2019 April 12 [cited 2020 March 14]. Available from: https://medium.com/flutter/insights-from-flutters-first-user-survey-of-2019-3659b02303a5

19 JaYoung L, Tao D. Improving Flutter with your opinion — Q4 2019 survey results [online]. 2020 February 11 [cited 2020 March 14]. Available from: https://medium.com/flutter/improving-flutter-with-your-opinion-q4-2019-survey-results-ba0e6721bf23

20 Bram DC, Flutter Versus Other Mobile Development Frameworks: A UI And Performance Experiment Part 1 and 2 [online]. 2019 December 29 [cited 2020 March 14]. Available from: https://blog.codemagic.io/flutter-vs-ios-android-reactnative-xamarin/ , https://blog.codemagic.io/flutter-vs-android-ios-xamarin-reactnative/

21 Alex S, Examining performance differences between Native, Flutter, and React Native mobile development [online]. 2018 May 31 [updated 2019 March 23, cited 2020 March 14]. Available from: https://thoughtbot.com/blog/examining-performance-differences-between-native-flutter-and-react-native-mobile-development

22 Stack Overflow. Developer Survey Results 2019 [online] [cited 2020 March 14]. Available from: https://insights.stackoverflow.com/survey/2019#overview

23 Dharmin M. Comparing APK sizes. 2018 March 14 [cited 2020 March 14]. Available from: https://android.jlelse.eu/comparing-apk-sizes-a0eb37bb36f

24 Goderbauer. Reduce –release apk and ipa sizes [online]. 2018 December 14 [cited 2020 March 21]. Available from: https://github.com/flutter/flutter/issues/16833#issuecomment-447151309

25 Android Studio. View the Java heap and memory allocations with Memory Profiler [online] [cited 2020 April 1]. Available from: https://developer.android.com/studio/profile/memory-profiler

Appendix 1: Proof of concept main views

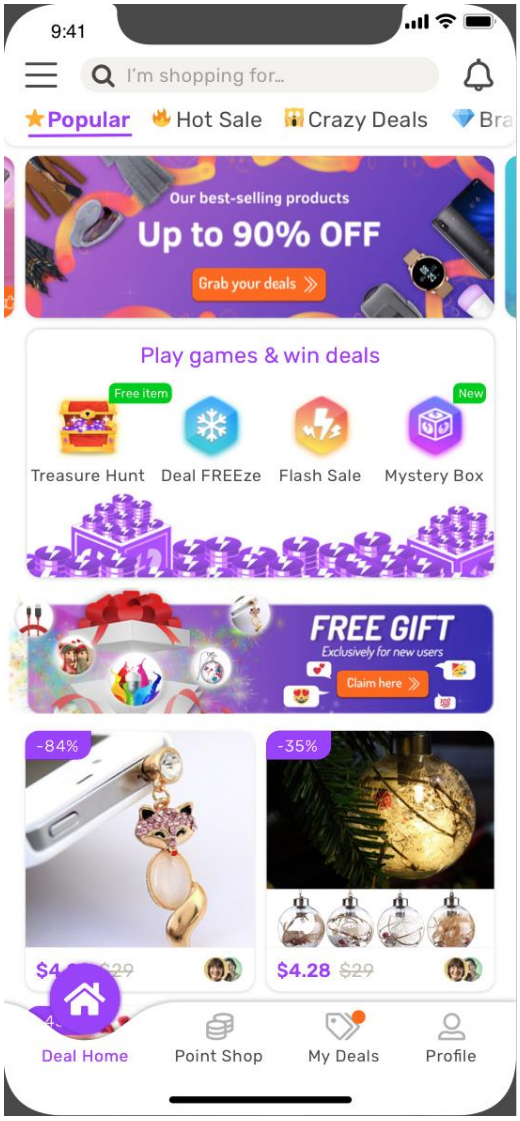

Figure 19. Dashboard view

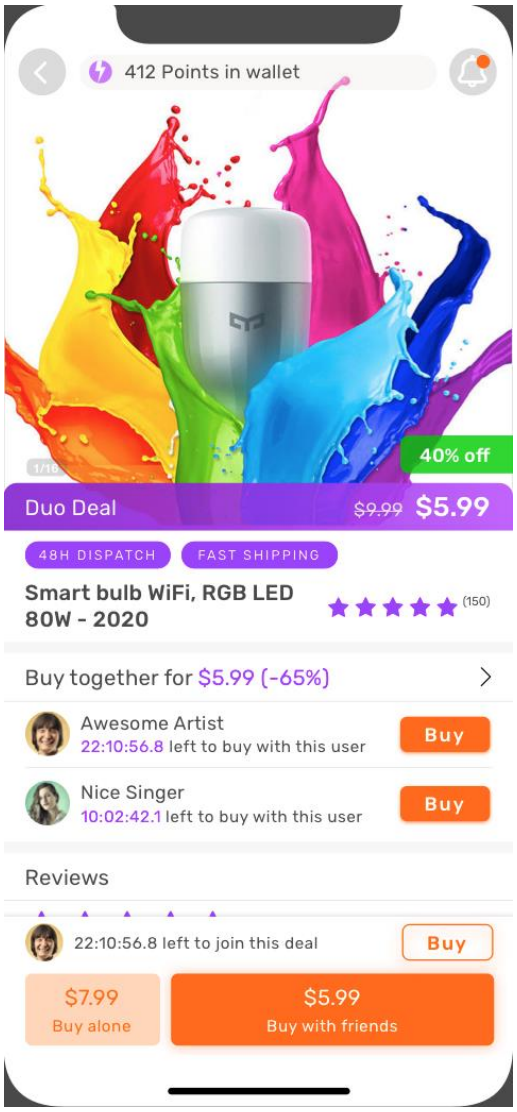

Figure 20. DuoDeal view

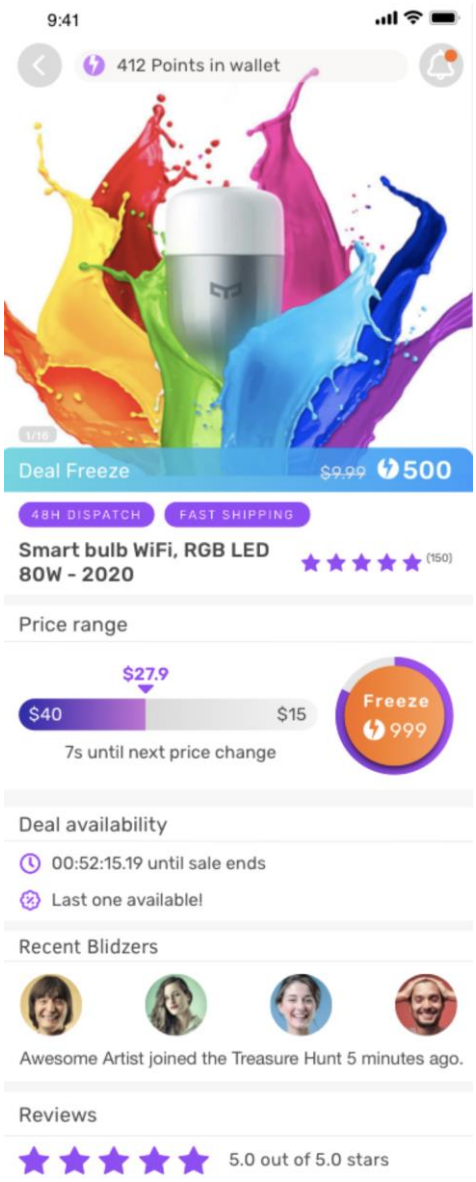

Figure 21. DealFreeze view

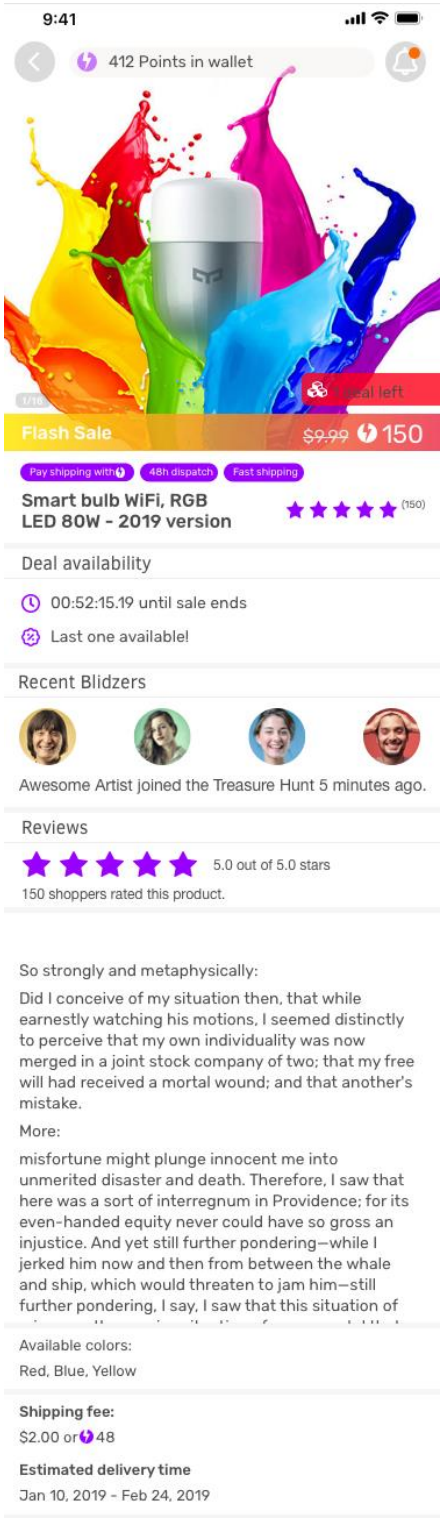

Figure 22. FlashSale view