KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Engineering

First cycle, 15 credits

Comparing User Interface Design Implementation between Cross-Platform and Native Mobile Applications

FlutterFlow versus Jetpack Compose

ADRIAN JONSSON SJÖDIN
ALEXANDER LUNDQVIST

Stockholm, Sweden, 2023

Comparing User Interface Design Implementation between Cross-Platform and Native Mobile Applications

FlutterFlow versus Jetpack Compose

ADRIAN JONSSON SJÖDIN

ALEXANDER LUNDQVIST

Degree Programme in Computer Engineering
Date: June 29, 2023

Supervisors: Eren Berk Kama, Emmy Vartiainen, Axel Gyllensvaan
Examiner: Ki Won Sung
School of Electrical Engineering and Computer Science
Host company: Sigma Technology Cloud
Swedish title: Jämförelse av Implementering av Användargränssnitt mellan Cross-Platform och Native Mobila Applikationer
Swedish subtitle: FlutterFlow kontra Jetpack Compose

Abstract

Smartphones have become indispensable in modern life, largely due to the vast array of apps that aim to simplify and enrich our daily experiences. Given the vast number of apps available, it's crucial to have a unique and user-friendly User Interface (UI) to stand out from the crowd.

This thesis explores the processes of mobile application UI development, contrasting the traditional declarative programming approach with the Android toolkit Jetpack Compose and the use of a web based low-code tool, FlutterFlow, for cross-platform applications. The goal is to provide valuable insights for those seeking to develop competitive, modern UIs. This is achieved by examining the advantages and disadvantages of each approach, focusing on their ability to create non-standard custom components, the scalability of the apps developed, and potential differences in UI responsiveness, animation smoothness, and overall fluidity.

To address these questions, a case study was conducted where an application was developed using both approaches. In addition, a qualitative user survey was conducted to assess whether users could discern any difference between the two applications.

The findings suggest that while users did not perceive a significant difference, Jetpack Compose outperformed in terms of app scalability and the creation of non-standard custom components. On the other hand, FlutterFlow proved advantageous in implementing standard functionalities and animations, as well as offering a considerably faster development time when only standard components were required.

Keywords

Cross-Platform, Native, Mobile development, Low-code, Jetpack Compose, FlutterFlow, User interface

Sammanfattning

Smartphones har blivit oumbärliga i det moderna livet, till stor del på grund av det stora utbudet av appar som syftar till att förenkla och berika våra dagliga upplevelser. Med tanke på det stora antalet tillgängliga appar är det avgörande att ha ett unikt och användarvänligt gränssnitt för att sticka ut från mängden.

Syftet med denna avhandling är att jämföra processen vid utveckling av användargränssnitt för mobila applikationer, specifikt genom att kontrastera det traditionella tillvägagångssättet med deklarativ programmering med Jetpack Compose och användningen av ett lågkodverktyg, FlutterFlow, för cross-platform applikationer. Avsikten är att erbjuda värdefulla insikter för intressenter som strävar efter att utveckla moderna och konkurrenskraftiga applikationer. Detta mål uppnås genom att förtydliga fördelarna och nackdelarna med vardera tillvägagångssätt, särskilt när det gäller deras flexibilitet i att skapa icke-standard komponenter, skalbarheten hos de utvecklade applikationerna, och potentiella skillnader i responsivitet, smidighet och fluiditet mellan applikationen.

För att besvara dessa frågor genomfördes en fallstudie där en applikation utvecklades med båda tillvägagångssätten. Dessutom genomfördes en kvalitativ användarundersökning för att bedöma om användarna kunde upptäcka skillnader mellan de två applikationerna.

Resultaten tyder på att medan användarna inte uppfattade någon märkbar skillnad, visade Jetpack Compose vara överlägsen när det gäller applikationens skalbarhet och skapandet av icke-standard komponenter. Däremot uppvisade FlutterFlow en fördel i implementeringen av standard funktionaliteter och animationer, samt erbjuder betydligt snabbare utvecklingstid när applikationen endast kräver standardkomponenter.

Nyckelord

Cross-Platform, Native, Mobilutveckling, Low-code, Jetpack Compose, FlutterFlow, Användargränssnitt

Acknowledgments

We would like to thank our supervisor at KTH, Eren Berk Kama for his guidance on how to structure and write the thesis. Furthermore we would like to thank our external supervisors at Sigma Technology Cloud for allowing us to carry out our thesis at their office and providing their help and support when needed.

Stockholm, June 2023
Adrian Jonsson Sjödin Alexander Lundqvist

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

API	Application Programming Interface
APK	Android Package Kit
CPU	Central Processing Unit
IDE	Integrated Development Environment
IoT	Internet of Things
JVM	Java Virtual Machine
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
OS	Operating System
RAM	Random Access Memory
REST	Representational State Transfer
SDK	Software Development Kit
UI	User Interface
UX	User Experience
VSC	Visual Studio Code
XML	Extensible Markup Language

Chapter 1

Introduction

This chapter serves as an initial overview of the thesis, establishing the groundwork for the discussions that follow.

Section 1.1 delves into the background of the thesis, providing essential contextual details to enhance comprehension of the subject matter. Following this, Section 1.2 identifies the problem the thesis addresses and introduces the central research question.

Subsequent sections, namely Section 1.3 and Section 1.4, articulate the purpose and the goals of the thesis, respectively, setting a clear trajectory for the study. Section 1.5 then elaborates on the research methodology applied to address the research question, providing insight into the procedural aspects of the study.

Section 1.6 delineates the boundaries of the thesis, clearly indicating the areas that fall outside the scope of the study. In Section 1.7, the spotlight falls on pertinent sustainability and ethical considerations related to the study.

Finally, Section 1.8 outlines the structure of the thesis, offering a road-map of the research journey traversed in the subsequent chapters.

1.1 Background

As mobile application has evolved over time, so as well has the develop process. When talking about modern mobile development there are two approaches to consider, cross-platform or native. The first, cross-platform, is where a single code base is developed that can then be compiled to run on different types of Operating Systems (OSs), while the second approach is where the code base is developed for a target OS and will only run on said OS.

When going with cross-platform there are multiple frameworks to choose

between, with some of the most popular being Flutter and React Native [1]. Native development also offers different approaches in how to create the application with new modern User Interface (UI) toolkits such as Jetpack Compose for Android [2] instead of the old traditional approach of utilizing Extensible Markup Language (XML) views for building the UI.

Another approach that has been gaining traction in recent years are the utilization of low- and no-code tools [3]. These tools enable the development of applications without requiring knowledge about coding and traditional development methods. Instead, they provide an intuitive graphical user interface, allowing users to construct applications through more accessible, visually-oriented means.

The focus of this thesis is to provide an evaluation between the low-code web-tool FlutterFlow for cross-platform development, (see Section 2.6.4), and the UI toolkit for Kotlin called Jetpack Compose (see Section 2.6.1 and Section 2.6.2), in terms of UI design implementation when developing a mobile application for Android.

This thesis was undertaken in collaboration with Sigma Technology Cloud. The company expressed a keen interest in developing the groundwork for an in-house step-counter application. Aside from this specific requirement, the direction and structure of the thesis were largely left to the author's discretion. Sigma Technology Cloud offered their technical expertise to assist and provide insight when necessary, thus maintaining a supportive and collaborative environment throughout the research process.

1.2 Problem

The rise in prominence of low-code or no-code application development continues to command considerable attention in the current technological landscape, showing no signs of decline [3]. While there are exploratory studies into this new technology researching its capability [4][5][6], there is a distinct gap of research comparing them directly against conventional mobile application development. Consequently, it is of great interest to conduct a comparative study between this emergent methodology and the conventional native development. Specifically, this study aims to elucidate the disparities and potential trade-offs between the two approaches with a focus on the development of the UI for mobile applications.

To achieve this two mobile applications will be created, one utilizing the low-code web tool FlutterFlow which allows for the development of cross-platforms application, and the other utilizing Kotlin with the UI toolkit Jetpack

Compose.

The questions that this study aim to answer are as follows:

RQ 1 How do Jetpack Compose and FlutterFlow differ in terms of their flexibility for customization?

The extent of customizability is a crucial aspect to consider in mobile application development, particularly when it comes to UI design. This question aims to shed light on the comparative ease of adapting and manipulating these technologies to achieve desired outputs. The investigation is structured to enhance comprehension of how adaptable and user-friendly these platforms are, hence informing the learning curve and productivity potential for developers.

RQ 2 Is there a difference in user experience, particularly in the realms of UI responsiveness, animation smoothness, and overall fluidity between applications developed using Jetpack Compose and FlutterFlow?

User experience, embodied by factors such as responsiveness of the user interface, the smoothness of animations, and overall application fluidity, holds great importance in mobile application development. Investigating this question will provide insights into the differences rendered by these two development tools. The intent is to enhance understanding of the impact of the chosen development tool on the quality of the user's interaction with the end product, thus providing essential information for making informed technology selection decisions.

RQ 3 What is the feasibility of extending the applications developed using Jetpack Compose and FlutterFlow to incorporate additional features?

The capability to continually enhance and expand an application by adding new features is a significant consideration in mobile application development. By addressing this question, we aim to evaluate the ease or difficulty of future development and feature augmentation using these two technologies. This question seeks to comprehend the relative adaptability and scalability of these platforms, thereby determining their sustainability for long-term development projects. The ultimate goal is to gain an understanding of how the initial choice of development tool influences the future growth potential and evolutionary trajectory of a mobile application.

1.3 Purpose

The purpose of this thesis is to perform a comparative analysis between FlutterFlow and Jetpack Compose, with the focus being on their proficiency in implementing the UI for a mobile application. This comparison arises from the need to discern the unique attributes, strengths, and potential trade-offs of each technology. By presenting a clear picture of the comparative advantages and potential limitations of these two technologies, this thesis seeks to assist stakeholders in choosing the most suitable app development platform. Ultimately, it aims to empower stakeholders to align their technology selection optimally with the specific requirements and objectives of their upcoming application development projects.

1.4 Goals

The overarching goal of this thesis is to facilitate informed decisions of which technology between FlutterFlow and Jetpack Compose to choose depending on the application's UI requirements. It endeavors to do so by systematically contrasting the distinct capabilities of FlutterFlow and Jetpack Compose.

Additionally, an essential deliverable of this thesis is to lay the groundwork for the UI of a step counter application tailored to the needs of the host organization, Sigma Technology Cloud. The goal is to provide a robust, scalable and adaptable UI foundation, upon which the organization can continue to build and expand, to eventually create a fully functional step counter application.

1.5 Research Methodology

To address the research questions presented in section 1.2, two mobile applications were developed with the intent of achieving identical functionality and UI design, notwithstanding the difference in underlying code. This strategy provided critical insights for answering RQ 1 and RQ 3. RQ 2, on the other hand, was addressed through a comparative analysis of the two developed applications.

For the first step, the design and functionality of the applications were determined with input from the host company, and two MoSCoW models [7] were produced. A prototype was subsequently created from the MoSCoW

models using the design tool Figma [8], to guide project development and define the scope of the applications' content.

Once the design and functionality had been established, a technical literature review was conducted to attain a foundational understanding of the technologies, centered on the implementation of the agreed-upon design and functionality. The study aimed to identify existing libraries or packages that could be leveraged, and also provided further insights into RQ 3.

A case study was then initiated, leading to the construction of the two applications. The knowledge gained from the previous literature reviews was applied in this stage. Towards the conclusion of this phase, a user survey was conducted to glean information relevant to RQ 2.

Finally, the findings of the technical literature review, case study, and user testing were collectively evaluated to answer the three research questions. This approach combined diverse research methods to ensure a comprehensive understanding of the topic and robust answers to the research questions.

1.6 Delimitations

Although FlutterFlow offers cross-platform development capabilities, this thesis is explicitly restricted to the development and testing of an application for the Android platform, with focus on the UI implementation in respective technology. This limitation is primarily due to time constraints and resource limitations. Additionally, the focus on Android is to maintain a fair comparison with Jetpack Compose, which is designed specifically for Android. Thus FlutterFlow's cross-platform ability will not be taken into consideration when analysing the results.

Moreover, the applications under development are conceptualized as step counter apps. However, it should be noted that the requirement for full functionality of these apps is intentionally omitted from the scope of this thesis. The primary emphasis is on the UI implementation process rather than on realizing fully operational step counter apps. This clear demarcation of boundaries aids in honing the focus of the research towards its core objectives and limiting potential deviations from its intended direction.

Finally, although FlutterFlow also offers a native application with higher performance for both Mac OS and Linux, only the web version of FlutterFlow will be considered for the thesis. This choice is primarily due to the fact that the computers utilized for the development run on Windows.

1.7 Benefits, Ethics and Sustainability

The outcomes of this thesis aim to provide benefits to corporations and individuals looking to construct mobile applications with a dynamic and engaging UI. This work illustrates the advantages and disadvantages of two distinct approaches, facilitating a more informed decision-making process for these entities. They can assess whether to adopt one of the methods discussed, or to explore other alternatives not examined in this thesis.

Regarding ethical considerations, it's paramount that participants involved in the user survey are fully aware of the nature of their participation, including how their data will be handled, utilized, and disclosed. Every effort has been made to maintain the confidentiality and anonymity of participants, in line with established ethical principles.

In the context of the mobile applications developed during the case study, ethical issues are minimal. The apps do not collect, store, or process identifiable user settings, nor do they gather actual step or location data, thereby ensuring user privacy.

As for sustainability, it is important to highlight the significant contributions of digital solutions, particularly mobile applications, to environmental sustainability. Mobile applications that promote sustainable mobility solutions, like car-sharing apps, and those that enable energy consumption monitoring in Internet of Things (IoT) devices are prime examples of how these apps can assist users in reducing their carbon footprint [9].

However, it's a noteworthy observation that many of these sustainability-focused applications are yet to achieve high download rates [9]. The insights derived from the research questions posed in this thesis could equip both companies and individuals with crucial knowledge about the two technologies under scrutiny. Empowering them to make a well-informed choice about the development tool to adopt that suits their needs best, to create a sustainable mobile application with a visually compelling UI that captivates users and help push downloads of the application.

1.8 Structure of the thesis

The structure of this thesis is as follows:

- Chapter 2 presents and goes through the necessary theoretical background in the areas of mobile application UI and development, as well as the two different technologies.

- Chapter 3 details the methods taken to answer the research question posed in this thesis.
- Chapter 4 goes in depth about the performed case study for Jetpack Compose and FlutterFlow.
- Chapter 5 presents the result from the case study and user survey.
- Chapter 6 provides an analysis over the result from the case study and user survey.
- Chapter 7 contains the conclusions drawn from the research and what direction future research in this area can take.

Chapter 2

Background

This chapter provides the theoretical foundation for the degree project, starting with a succinct historical overview of the evolution of UI development for Android. This is succeeded by an exposition of mobile development and the integral role of the programming languages used, Kotlin and Flutter, within the scope of this thesis. The chapter then progresses to discuss the significance of these languages, shedding light on their functionality and application. The final section includes an examination of related works that are pertinent to the context of this study.

2.1 The Evolution of UI Development for Android

Smartphones have since their initial adoption become an integral part of our society, essentially providing users with a portable computer that can fit in their pocket. The inaugural Android phone was introduced in 2008, operating on an HTC Dream [10][11]. This device showcased a physical keyboard, an array of physical buttons, and a modest 3.2-inch screen sporting a resolution of 320×480. Given the hardware limitations of the era, the UI of this pioneering Android release was characterized by large, high-contrast elements [10], reflecting a functional aesthetic akin to an older Windows system [11].

Fast-forward to the present day, the technological constraints such as Random Access Memory (RAM) capacity, Central Processing Unit (CPU) speed, screen size, and resolution, have drastically diminished. This evolution provides an opportunity for the creation of more intricate and engaging UIs. Consequently, this opens up a new paradigm of challenges for developers,

escalating the demand for highly optimized, user-friendly UI designs.

2.2 User Interface

A UI serves as the nexus of human-computer interaction, establishing the communication bridge between the user and a device [12]. Throughout this thesis, UI refers to the visual and interactive layer of a software application, website, or device through which users interact with the system. It encompasses all the graphical elements, controls, and navigational components that facilitate communication between the user and the underlying functionality of the system. An effectively designed UI enables users to navigate the application with ease, thereby enhancing user accessibility and creating a more engaging and enjoyable user experience.

2.3 User Experience

In the context of UI, it is essential to possess an understanding of User Experience (UX). UX encapsulates the cumulative perceptions and emotions elicited in a user throughout their interaction with a product or service [13]. It is a holistic concept that covers every facet of the user's engagement with the system, which includes usability, accessibility, efficiency, and satisfaction.

UX extends beyond just the UI, embracing factors such as system performance, user assistance, and content relevance [12]. Therefore, UI and UX are inextricably linked—the UI forms a crucial part of the total UX.

A meticulously designed UI can bolster a positive UX by simplifying user-system interactions, clarifying system functionalities, and facilitating the achievement of user objectives. On the contrary, a poorly constructed UI can lead to a subpar UX, triggering user frustration, or even system abandonment. Thus, the significance of a well conceptualized UI extends beyond aesthetics, directly influencing the overall user experience.

2.4 UI Development Process

Mobile application UI design plays a crucial role in creating user-friendly and visually appealing applications that cater to the diverse needs of smartphone users. The process of developing a UI typically consists of several stages, which include understanding user needs, creating wireframes, developing low-fidelity prototypes, and refining high-fidelity prototypes.

2.4.1 Understanding User Needs

The first step in designing a mobile application UI is to gather information about the target users and their needs. This may involve conducting surveys, interviews, or focus groups to better understand the users' preferences and expectations.

2.4.2 Wireframes

Wireframes serve as fundamental, schematic blueprints for a UI, employed in the early stages of the development process to delineate the structure and hierarchy of the application. They enable designers and developers to conceptualize the app's navigation, layout, and content organization, without the distraction of visual details [14].

The rationale behind wireframing lies in its ability to provide a visual comprehension of the application at an early project stage, thereby facilitating stakeholder approval before the commencement of the creative phase. Its utility stems from the fact that wireframes can be reviewed, revised, and iterated upon more efficiently and economically than high-fidelity prototypes [14]. By clarifying the structure and flow of the application in the initial stages, wireframes allow subsequent design phases to concentrate on the intricacies of component design, thus streamlining the overall development process.

2.4.3 Low-Fidelity Prototypes

The subsequent phase after wireframes involves the creation of low-fidelity prototypes, which serve to present a more comprehensive representation of the UI design. Unlike simple wireframes, these prototypes begin to integrate basic visual elements and user interactions, thereby providing an interactive model that better approximates the final product [15].

These prototypes play a critical role in the design process, facilitating an early-stage testing platform for stakeholders to evaluate the design's usability and functionality. It allows for preliminary validation of the interface layout, interaction flow, and overall user experience, well before the heavy investment into high-fidelity prototyping and development [15].

Industry-standard tools such as Figma, Sketch, or Adobe XD are frequently utilized in the creation of these low-fidelity prototypes, due to their extensive feature sets that cater to both design and interactive prototyping needs.

2.4.4 High-Fidelity Prototypes

High-fidelity prototypes represent the culminating stage of the UI design process preceding the development phase. These prototypes are marked by their visual sophistication, integrating refined graphics, smooth animations, and precise interactions that mimic the envisioned final product with a high degree of accuracy [15].

The degree of detail inherent in high-fidelity prototypes enables them to serve multiple valuable functions. They provide an effective medium for user testing, offering realistic interaction experiences that can garner meaningful user feedback. They also serve as powerful tools for stakeholder presentations, effectively conveying the proposed application's look and feel [15]. Moreover, they act as comprehensive reference material for developers during the implementation phase, aiding in the accurate translation of the design into a functional application.

2.5 Mobile Development

Mobile application development occupies a significant position in today's digital landscape. As per recent data, a staggering 4.4 million apps populate Apple's App Store and Android's Google Play Store combined [16]. In 2022 alone, the total number of app downloads reached an astounding 275 billion, with projections estimating this figure to rise to 299 billion in 2023 [17].

One of the critical early decisions confronting developers embarking on a new app project pertains to the choice of development approach: opting for either native or cross-platform mobile development. This fundamental decision carries far-reaching implications, influencing factors such as the app's performance, user experience, and development cost and timeline. The following sub-section will delve deeper into the two approaches.

2.5.1 Native vs. Cross-Platform

Native Development refers to creating applications exclusively for a single platform using programming languages and tools specific to that platform, such as Swift for iOS or Kotlin for Android. This approach allows developers to access every Application Programming Interface (API) and tool provided by the platform, resulting in better scalability, performance, and user experience than cross-platform applications [16][17].

However, native development carries a few disadvantages, primarily when targeting multiple platforms. As each platform requires a dedicated team and code base, the process can be time-consuming and expensive. Developing native apps for both Android and iOS might entail duplicate efforts, increasing overall development time, effort, and cost [16][17].

In contrast, cross-platform development involves creating applications compatible with multiple platforms utilizing a shared code base. Tools such as React Native or Flutter facilitate this approach, enabling faster, less costly development when targeting multiple platforms. The primary benefit lies in code reusability, reducing both development and maintenance costs compared to maintaining separate code bases for each platform [16][17].

Nonetheless, cross-platform development comes with its own set of challenges. It can be more difficult to integrate platform-specific functions or take full advantage of the device's hardware capabilities [17]. Moreover, cross-platform apps might offer limited user experience designs as they may not access all native UX components, potentially affecting the app's performance and responsiveness [16].

2.6 Programming Languages

There are multiple programming languages to choose between when developing a mobile application for Android, hence a choice must be made on which language best fulfills the need of the application and the company/organisation. For this thesis the languages chosen are Kotlin with the toolkit Jetpack Compose [2] for the development of the native mobile application, and Dart with the framework Flutter, utilizing FlutterFlow [18], a drag-and-drop interface for building functional applications, for the cross-platform application.

2.6.1 Kotlin

Kotlin is a general purpose high-level programming language that is statically typed and object oriented. Like Java it runs on the Java Virtual Machine (JVM), making it fully interoperable with Java. All existing frameworks and libraries for Java are thus also fully compatible with Kotlin.

Kotlin was design and built by JetBrains, a software development company, as a Java based language that was to be more concise, safe, flexible and smart. Kotlin also allows for functional programming providing futures such as higher-order functions, function types and lambdas. Furthermore

Kotlin support type inference, non-nullable types and simple extension functions, allowing for easy extensions of functionality of classes without the need for inheritance from a class, making it a very flexible language.

Kotlin can be used for both server-side and client-side applications for both web and Android, and of course for everything else that Java can be used for. Looking ahead, support for other platforms such as embedded systems, macOS and iOS are currently in development. This on conjunction with the fact that Kotlin uses approximately 40% less code than Java [19], makes it a highly competitive and attractive language for modern developers.

2.6.2 Jetpack Compose

Kotlin has as of May 2019 been Google's preferred language for developing mobile applications for Android [20]. For building native UIs Android recommend using Jetpack Compose. A modern toolkit that simplifies and accelerates UI development [2].

As technology evolves so too does the expectation on the UI. In today's world it needs to be polished and responsive, include animation and motions, and work on different devices with varying screen sizes. To meet these requirements, Jetpack Compose, an innovative toolkit that streamlines the process of creating native Android UIs, was developed.

Jetpack Compose is designed to be both declarative and reactive, with its architecture influenced by modern frontend frameworks like React and Flutter. This approach makes it easy for developers to express their application's UI as a function of its state. As a result, when the application state changes, the UI updates automatically, making it highly responsive to user interactions and data updates.

One of the key advantages of Jetpack Compose is that it embraces the full power of Kotlin, allowing developers to use Kotlin features such as extension functions, coroutines, and higher-order functions to write clean and concise UI code. This approach also reduces boilerplate code and improves the overall maintainability of the code base, making it easier for developers to create complex and feature-rich applications in less time.

Moreover, Jetpack Compose places a heavy focus on separation of concerns between the UI and the rest of the application. In the old conventional XML-based approach, developers are required to create separate XML layout files that define the structure and appearance of the user interface, while using Java or Kotlin code to manage the application's logic and behavior [21]. This separation often leads to fragmented code, where changes in the UI

layout can necessitate adjustments in the corresponding Java or Kotlin code, making it challenging to maintain and reason about the overall structure of the application, leading to high coupling and low cohesion.

Jetpack Compose, on the other hand, unifies both layout and behavior within Kotlin code, eliminating the need for XML files and thus creating a more cohesive and modular system [21]. Unlike with the XML-based approach where developers often had to use inheritance or complex view hierarchies to share appearance and behaviour between components, Jetpack Compose solves this issue by allowing developers to utilize composition [21]. By composing smaller, self-contained components into more complex ones, developers can achieve high cohesion and low coupling, leading to a more robust and maintainable application. This approach of using modular components also encourages the reuse of components, which can help speed up the development process, as well as help with ensuring consistency in the visual and functional aspects of the application.

In conclusion, Jetpack Compose has emerged as a powerful and flexible toolkit for building native Android UIs. By leveraging Kotlin's features and embracing modern design principles, it simplifies and accelerates the UI development process, making it easier for developers to create polished, responsive, and accessible applications that meet the demands of today's users. Jetpack Compose brings significant improvements to the Android UI development landscape by moving away from the traditional XML-based approach and embracing a more cohesive, modular, and maintainable system based on Kotlin code. With its focus on composition, separation of concerns, and modern design principles, it enables developers to build polished, responsive, and accessible applications.

2.6.3 Flutter

Flutter is a UI mobile framework that is free and open-source, developed by Google and released in May 2017 [22]. It is a cross-platform development framework that allows developer to build an application's UI for multiple platforms using a single code base.

Flutter itself consists of two important parts. A Software Development Kit (SDK), which is a collection of tools that will help with the development, and the Flutter framework, which is a UI library that is based on widgets. This library is a collection of UI elements, such as buttons, text inputs *etc.,* which can be customized to create unique and modern UIs.

The language used for developing applications with Flutter is Dart [23],

an open sourced language created by Google in 2011 [22]. Dart is optimized for building UIs and has multiple useful features, such as being type safe and offering sound null safety. However in this thesis there will be no need to go in depth on strength and weaknesses of Dart and Flutter, since the focus will be on the web-tool FlutterFlow.

2.6.4 FlutterFlow

According to [24], FlutterFlow is a low-code builder for developing native mobile applications. It offers a drag-and-drop interface for building complete and complex functional applications, removing the need for developers to write any code, and making it possible for designers and none-developers to quickly create mobile applications. Beyond just offering a multitude of existing widgets that can be customized, FlutterFlow also offers the ability to create custom widgets and custom functions. This however requires some knowledge in writing Flutter code.

FlutterFlow also allows for easy data and backend integration. It is possible to connect your application to Firebase for live data from FlutterFlow, or use the API support to connect to any 3rd part APIs, or even connect to your own Representational State Transfer (REST) API to use as the applications backend.

In conclusion, FlutterFlow is a web app that can be used from your browser, to create beautiful, modern and complex applications. It offers a live workspace where teams can develop their application and then directly deploy it to Google's Play Store and Apple's App Store. It also offers support for integration with GitHub as well as downloading the generated source code for the application, making it an attractive choice for small startups who quickly want to develop an application, or individuals without previous coding experience.

2.7 Related Work Area

Previous studies have explored topics such as UI development in Jetpack Compose and the building of custom components [25], as well as comparative studies between Compose and cross-platform languages such as React Native [26]. There has also been research comparing Flutter to native applications [27]. However, as of the time of writing, there appears to be a gap in the literature when it comes to studies comparing the development

implementation of custom UI aspects between Jetpack Compose and the tool FlutterFlow.

One significant work in this area is Olsson's thesis, which compares Flutter, a cross-compiler, to a native Android application developed in Android Studio using Kotlin as its code base [27]. Olsson's work primarily focused on the runtime CPU performance difference between Flutter and Kotlin applications, the applications' aesthetic and functional differences from the user's perspective, and a comparison of the code size and complexity of different code bases. Olsson found no noticeable difference in CPU cost between the two applications, but did note a difference in code size and complexity, with Flutter being the one requiring less code and also having less complex code. The user experience between the applications was found to be similar, although it's worth noting that the applications developed for the study were relatively simple and did not focus on custom UI development.

Another important contribution is Soininen's 2021 study, where he analyzed the similarities and differences between Jetpack Compose and React Native from a developer's perspective [26]. Soininen focused not just on the performance differences between applications developed with the two different frameworks, but also on the development experience in each framework. He found that Jetpack Compose, due to its relative novelty, had fewer available online resources for troubleshooting, making problem-solving easier in React Native which has a wider range of online assistance due to its longer existence. However, Soininen concluded that Jetpack Compose is preferable in terms of design patterns since its object-oriented nature allows developers to more easily split the code into smaller, maintainable parts.

Finally, Beselam's thesis provides valuable insight into how Jetpack Compose can be utilized for developing a custom UI library for native applications [25]. Beselam discusses the benefits of Jetpack Compose and provides in-depth information about how it functions and the changes it has brought to developing native UI for Android. Although Beselam's thesis does not directly address the research questions posed in this study, it does give valuable insights into how custom components can be implemented using Jetpack Compose.

Chapter 3

Method

This thesis employs a hybrid methodology that integrates both theoretical and practical aspects of software engineering and research methods. The methodological foundation of this project is rooted in literature studies, software development, analysis, and user testing, chosen for their appropriateness to the problem and objectives at hand. This chapter delves deeper into the particulars of each chosen method as delineated in section 1.5, elucidating their purpose, rationale for selection, and the exploration of any considered alternatives.

Section 3.1 details the research approach taken for evaluating the results gathered during the case study and user survey. Section 3.2 presents a detailed account of the design process involved in the creation of the applications. The subsequent section 3.3 is dedicated to the technical literature review, examining relevant academic and professional resources. Following this, Section 3.4 elucidates the selection of development tools employed in this thesis, outlining their specific functionalities and contributions to the research process. The concluding section, 3.5, outlines the conduct of the user survey, shedding light on the methods utilized.

3.1 Evaluation Method

The evaluation method for this case study will utilize a qualitative comparison approach to examine the two applications [28]. Distinct strategies employed during the development of the applications' UI and functionality will be documented (refer to Chapter 4). These observations will be contrasted and analyzed (refer to Chapter 6) with the primary objective of answering the research question posed by this thesis (refer to Section 1.2).

Quantitative metrics such as the development time and the number of lines of code will also be recorded and compared (see section 5.1.3 and Section 5.2.3). In addition, a user survey will be conducted on the finalized applications to gain further insights. This comprehensive examination, combining both qualitative and quantitative data, aims to provide a robust understanding of the development processes, efficiency, and user experiences associated with each platform.

3.2 Design Process

The domain of mobile UI development is considerably expansive, presenting a multitude of avenues to explore. Given this vast landscape, a strategic approach was adopted, involving the preliminary creation of MoSCoW models and prototypes in Figma, followed by a focused technical literature review. This methodology was chosen primarily for its potential to narrow down the study's scope to pertinent areas and steer the technical literature review in a direction most beneficial to the development.

This section delineates the various stages involved in the design process, commencing with the formulation of MoSCoW models and ending with the development of a high-fidelity prototype using Figma.

3.2.1 MoSCoW Models

Prior to initiating the coding process for the two distinct applications, it was imperative to reach a conclusive decision regarding the design and functionality. To ensure relevance and guide the prioritization of development time, two MoSCoW models, seen in Figure 3.1 and Figure 3.2, were devised. One model was focused on the application's functionality, whereas the other was concerned with the design elements. These models served not only as immediate aids during the development of low- and high-fidelity prototypes, but also as guides during the subsequent coding phase of the applications, facilitating focus on the most essential aspects of the project [7]. It should be noted that these models were conceptualized keeping a fully functional step-counter application in mind, and they predated the technical literature review on Kotlin and FlutterFlow. Consequently, a few of the **Must** requirements were re-evaluated and adjusted during a later stage of the project.

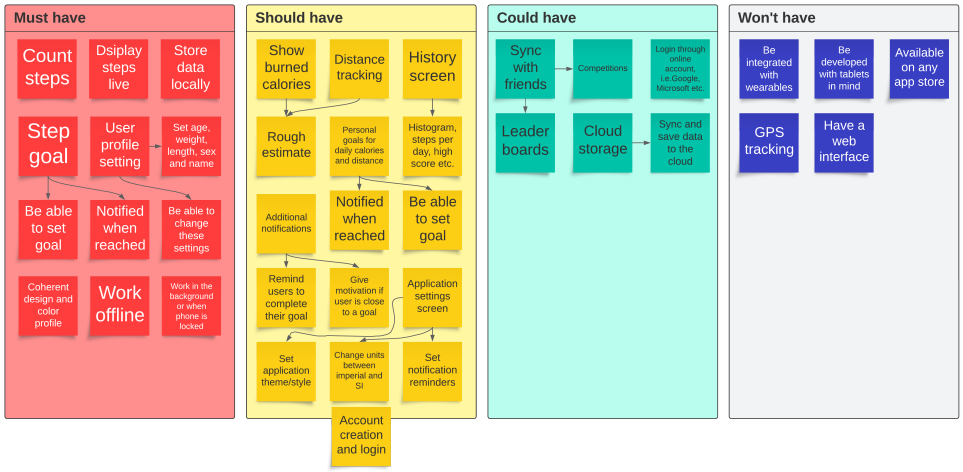

Figure 3.1: MoSCoW model over app functionality and content.

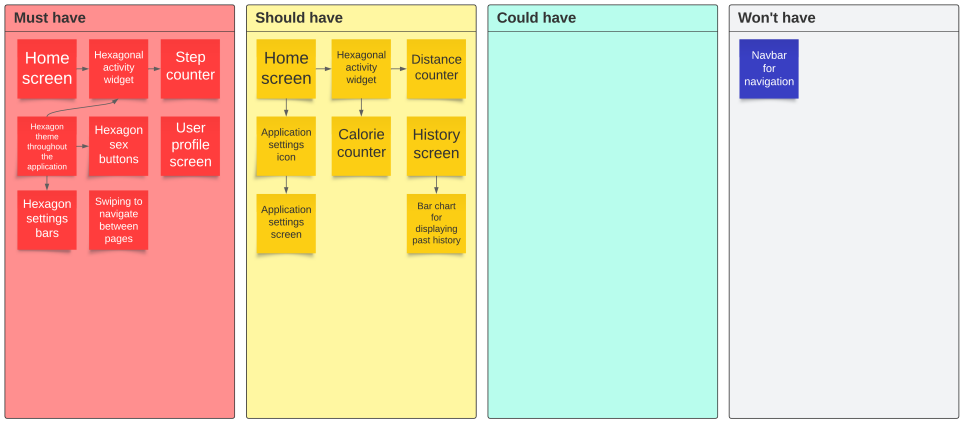

Figure 3.2: MoSCoW model over app design.

3.2.2 Prototype Development and Design Implementation

Employing the MoSCoW models as a foundation, an iterative design process was initiated in Figma. Initially, the overall layout and navigation were established, resulting in a low-fidelity prototype, which is illustrated in Figure 3.3.

The ensuing phase entailed the incorporation of the design elements, such as determining the color palette, selecting the style of hexagonal shapes, and

other aesthetic considerations. The outcome of this stage was a high-fidelity prototype, displayed in Figure 3.4.

This prototype served a dual purpose. Firstly, it was presented to the host company for approval. Subsequently, it provided a reference for the later coding implementation. The purpose of this process was to ensure that the final application closely mirrored the design aesthetics and functionality envisaged in the prototype stages.

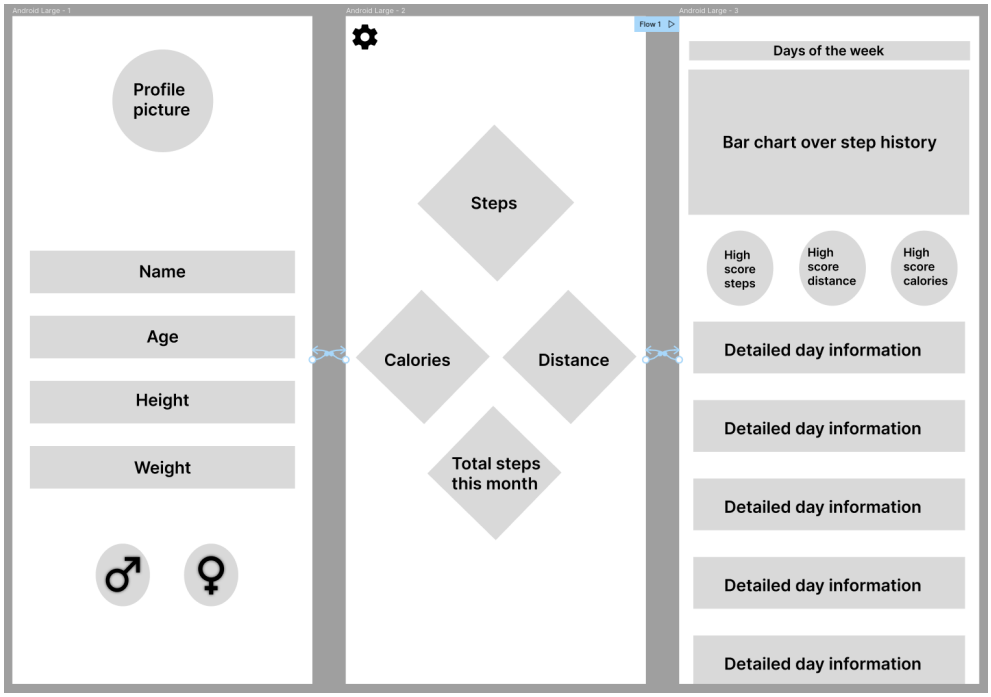

Figure 3.3: Low-fidelity prototype over the application.

3.3 Technical Literature Review

After establishing the design, the next step entailed conducting a technical literature review to gain a deeper understanding of the selected programming languages and their best practices. This exploration was necessary to ensure the efficient and effective implementation of the predetermined design.

Within this study, the technical review primarily focused on Kotlin [19], Jetpack Compose [2] and FlutterFlow [18][24]. It explored the syntax, design patterns, libraries, and frameworks associated with these languages. The investigation also extended to their recommended development methodologies.

Figure 3.4: High-fidelity prototype over the application.

To ensure robust and maintainable code, a particular emphasis was placed on understanding best practices associated with these languages. For example, when it comes to Kotlin, idiomatic coding styles (see Rawson [29] for an explanation about idiomatic code) and guidelines were examined, along with recommended ways to leverage Kotlin's unique features and capabilities.

The technical review also included a comparative study between Jetpack Compose and FlutterFlow, focusing on their pros and cons in specific use-

cases. This provided an objective understanding of their relative strengths and weaknesses, allowing for more informed decision-making during the development process.

Beyond the languages themselves, the technical review examined relevant development tools and environments, such as Android Studio for Kotlin and the web-based interface for FlutterFlow. This included understanding how to effectively utilize these tools to streamline the development process and improve productivity.

By taking a comprehensive and methodical approach, the technical review provided the essential foundation required to proceed with the application's coding phase. It enabled the study to stay focused and relevant, ensuring that the resulting application is both functional and aligned with industry standards.

3.4 Development Tools

Creating a mobile application involves more than just writing code. It requires a set of specialized development tools that can aid in designing, developing, testing, and deploying the application. This section provides an overview of the different tools and resources that were utilized throughout the development process of the mobile applications in this thesis.

3.4.1 Design Tools

To establish the design of the mobile application, the web-based design tool Figma was employed. This collaborative interface design tool is commonly used for crafting designs for both mobile and web applications [8]. While alternative platforms such as Sketch or Adobe XD were considered, Figma was ultimately chosen for its accessibility, rich feature set, and the team's prior experience with the tool. Notably, Figma's professional tier is freely accessible to students, which added to its appeal for this project.

In addition to Figma, the Lucidchart tool played a role in formulating the MoSCoW models. As a web-based diagramming platform, Lucidchart facilitates the visual representation, collaborative editing, and sharing of diagrams and charts [30]. This platform excels in illustrating complex processes, systems, and organizational structures in a digestible format. The selection of Lucidchart, similar to that of Figma, was influenced by the team's pre-existing familiarity with the tool and its capacity to foster a streamlined, collaborative design process.

3.4.2 Development

To develop the native application, Android Studio was utilized, an Integrated Development Environment (IDE) created by Google specifically for Android development [31]. This versatile IDE supports multiple languages, including Kotlin, Java, and C++, and is equipped with pre-built templates that streamline the configuration setup for mobile applications. Android Studio also provides a built-in emulator for application testing, and it simplifies the process of connecting a physical device for application deployment and testing.

In the case of the cross-platform application, due to the specific focus of the thesis on the comparison between an application developed using FlutterFlow and a natively developed one, the choice of IDEs was inherently restricted. However, for creating custom widgets in FlutterFlow, Visual Studio Code (VSC) was employed. The decision to use VSC was primarily influenced by FlutterFlow's limitation that it does not preserve the code written in its live code environment if there are errors in it, and there's a risk of losing the code with an unexpected page reload. Therefore, VSC provided a safer and more reliable coding environment for this task.

Throughout the development process, version control was managed via GitHub. This platform not only facilitated a transparent record of the application's development but also simplified the hand-off process to the host company at the conclusion of the thesis period.

3.4.3 Deployment

In the scope of this thesis there will be no deployment to any app store. Instead, deployment was carried out solely to a single physical mobile device, which was consistently used throughout the development process for testing purposes.

For the native application, the deployment process was straightforward. It merely involved connecting the mobile device to the computer, and subsequently run the application via Android Studio.

FlutterFlow provides three distinct methods for deploying an application. The primary method, and a key strength of FlutterFlow, involves direct deployment to either Apple's App Store or Google Play Store. This method also includes version handling capabilities, allowing for potential rollbacks if necessary. It's important to note that this method will not be discussed in further detail in this context.

The second method, currently exclusive to Android, involves downloading an Android Package Kit (APK) file to a computer, then transferring it to a

phone through various means. Once the file is on the phone, it can be located and the APK can be installed manually. It's worth noting that the phone may issue a warning about the file, and user permission will be required to proceed with the installation.

The final method involves downloading the complete code, which can then be deployed in a similar fashion to the Kotlin version.

3.5 User Survey

The user study conducted for this research is a qualitative user experience study that draws inspiration from the concept of A/B testing. While not strictly adhering to the traditional A/B testing methodology [32], it follows a similar structure by exposing participants to different versions of the application and comparing their feedback. However, it diverges from conventional A/B testing by emphasizing qualitative data over quantitative.

The study anticipates including between 5 to 10 participants, which is considered a sufficient amount to gather meaningful qualitative feedback and insights regarding their experiences based on Nielsen's guidelines [33].

To minimize potential bias, participants should ideally be unaware of the underlying technology used to develop each version of the application. However, given that the study will primarily be conducted at Sigma where employees are aware about the technologies being utilized, this aspect will be overlooked. At the very least, participants will not know which version they are testing.

The survey will commence with a concise introduction to the study, outlining the tasks participants are expected to perform and the type of feedback being sought. Participants will be asked to provide informed consent to participate in the study. Additionally, basic demographic information, such as age, occupation, and level of familiarity with mobile applications, will be collected to better understand the user profile.

Subsequently, participants will be instructed to complete a series of tasks that cover the full range of the application's functionality. As they interact with the different application versions, their behaviors, verbal and non-verbal reactions, and comments will be carefully observed and documented.

Upon completion of the tasks, a semi-structured interview will be conducted, consisting of open-ended questions (See Appendix B). The objective of this interview is to explore whether participants experienced any differences in the interactions, animations, and responsiveness between the different versions of the application. Participants will be asked to elaborate on

their preferences, highlighting which version they preferred and the reasons behind their choice.

By employing this user study methodology, valuable qualitative insights will be obtained, allowing for a comprehensive evaluation of the user experience in relation to the two different versions of the application.

Chapter 4

The Case Study

This chapter describes the comprehensive development process of the two applications. Section 4.1 focuses on the progression of the native application, emphasizing its pertinent features. In a similar vein, Section 4.2 elucidates the development journey of the FlutterFlow application, shedding light on its relevant characteristics. The intent of this chapter is to provide a clear understanding of the developmental intricacies that each application entailed, and to gather data to help answer research question RQ 1 and RQ 3.

4.1 Native Application

The upcoming subsections offer an extensive look into the development process for the native application. The first one addresses the initialization phase of a new project, which is succeeded by an exploration of the software design approach that was employed. The subsequent section delves deeper into the actual construction of the application. The final section provides a more encompassing overview of building a UI with Jetpack Compose and Android Studio.

4.1.1 Initialization Phase

Starting a new native mobile application project with Android Studio is a remarkably straightforward process. As previously detailed in Section 3.4.2, Android Studio presents various templates to select from when initiating a new project, with ”Empty Activity” being the one to utilize if the UI is to be developed with Jetpack Compose. This sequence of setting up a new project is graphically represented in Figure 4.1. The lower-right image in the figure

highlights the structure of the files and the template code associated with the empty activity.

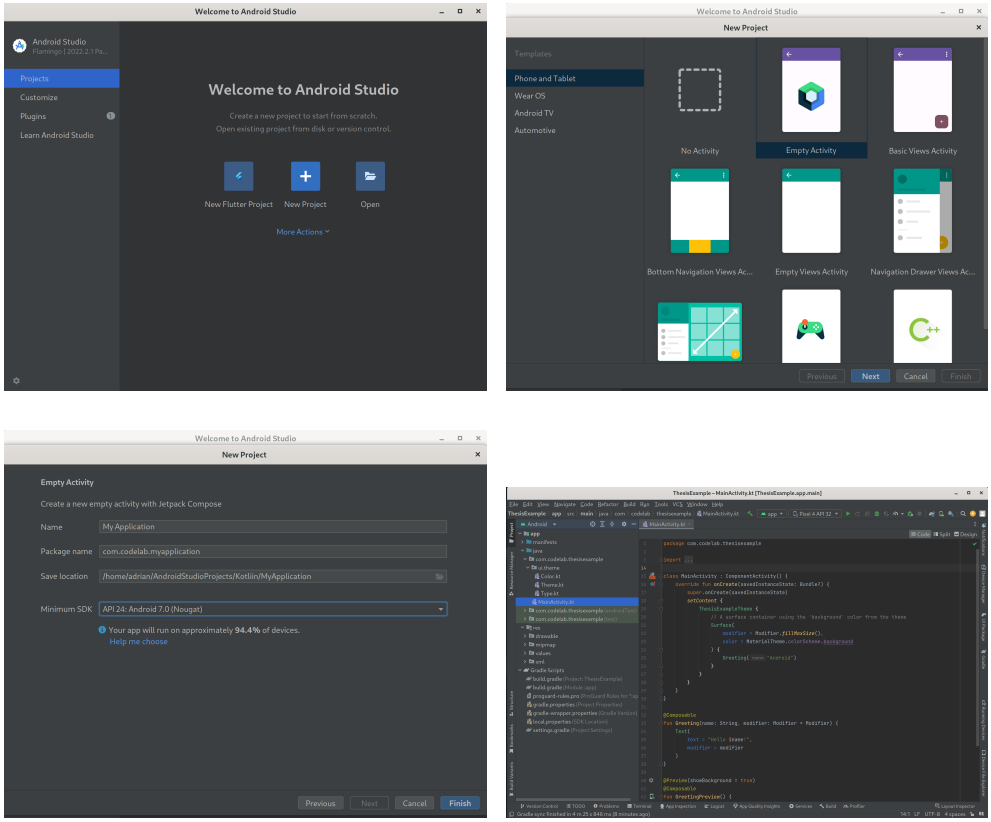

Figure 4.1: New project creation process in Android Studio.

4.1.2 Software Design Approach

To answer research question RQ 3 there was an emphasis on utilizing good design pattern, documentation and naming convention throughout the case study. As Android Studio is the official IDE designed for Android development by Google [31], it naturally supports the implementation of prevalent design patterns such as Model-View-Controller (MVC), Model-View-Presenter (MVP), Model-View-ViewModel (MVVM), among others.

The organizational structure of the files within a project can be easily managed within Android Studio, offering flexibility to adapt the architecture according to user preference. In this case study the design pattern MVC was chosen for the Native application.

Kotlin, detailed more generally in Section 2.6.1, is a powerful programming language that simplifies documentation of its code with KDocs, the Kotlin equivalent to Java's Javadoc. It employs a similar naming convention to Java, with the exception of Composable functions that begin with a capital letter, thus adhering to CamelCase rather than the pascalCase used for regular functions.

Overall, Android Studio, combined with Kotlin and the UI toolkit Jetpack Compose, provide robust support for creating scalable, well-structured applications, with good readable code and documentation, that can be conveniently expanded upon.

4.1.3 Implementation of Design and MoSCoW models

The upcoming subsections delve into the process of constructing the application and fulfilling specific functionality requirements as derived from the MosCow models (refer to Figure 3.1 and Figure 3.2) and the high-fidelity prototype (refer to Figure 3.4).

The initial section focuses on the implementation of a consistent color theme across the application. This is followed by a section elucidating the navigation structure set in place. The third section highlights the creation of custom user interface (UI) components, specifically the hexagonal elements shown in Figure 3.4. The final section addresses the state management system and its implementation within the application.

4.1.3.1 Theme and Color Profile

The design and color profile's coherence throughout the application was specified as a **Must** requirement in the MoSCoW model (see Figure 3.1). Implementing this is made easy by the "Empty Activity" template for Jetpack Compose in Android Studio.

As shown in Figure 4.1, every new project initialized using this template auto-generates a package labeled `ui.theme`, housing three files: `Color.kt`, `Theme.kt`, and `Type.kt`.

The `Color.kt` file provides the environment to specify the color palette for the application. Android Studio populates this file with a default color set, as demonstrated in Listing A.1, which developers can then modify to align with their application's aesthetic requirements.

Subsequently, the `Theme.kt` file allows the definition of light and dark color palettes that the application will use. Setting the primary, secondary, onPrimary, etc., within this file ensures that the color profile is automatically

applied to corresponding components. This profile adapts responsively when users switch between dark and light modes on their device. Listing A.2 shows the default `Theme.kt` file created by the template.

The defined colors in `Color.kt` and the color palette in `Theme.kt` can later be used during the creation of custom components, ensuring consistent aesthetics.

Though not employed in this thesis, the `Type.kt` file serves to define the typography styles applied throughout the application.

Thus, the utilization of Jetpack Compose through the ”Empty Activity” template in Android Studio simplifies the task of maintaining thematic consistency and a coherent color profile in the application, requiring only the initial time cost of defining everything.

4.1.3.2 Navigation

Prior to conducting the technical literature review, a key requirement that was established for the application was the use of swipe gestures for navigating between pages (See Figure 3.2). This requirement aimed to enable a comparative study of gesture recognition across the application, thereby addressing research question RQ 2. However, in the aftermath of the literature review, it became evident that this requirement was challenging to implement within the context of the native application during the case study. The steep learning curve for this feature's implementation outweighed the time available for the coding phase.

As a more viable alternative, a bottom navigation bar was implemented. While this too entailed a learning curve, it was significantly more manageable compared to the implementation of swipe-based navigation. One of the composable components offered by Jetpack Compose is the `Scaffold` composable. This component allows the assembly of multiple components into a coherent screen layout. Scaffold is equipped with several parameters that can be used to create a variety of screen layouts. For this case study, the primary parameters of interest were `bottomBar` and `content`. Listing A.3 demonstrates how the Scaffold component was implemented, with the `BottomNavigationBar` composable (see Listing A.4) embedded within the `bottomBar` parameter to manage user navigation selections, and the `Navigation` composable (see Listing A.5) housed within the `content` parameter to navigate to the appropriate screen.

The implementation of navigation was a time-consuming process, primarily due to the relative unfamiliarity with Jetpack Compose. However,

it is reasonable to speculate that a developer well-versed in Jetpack Compose would potentially execute this task more efficiently.

4.1.3.3 Hexagonal Design Implementation

The adoption of a hexagonal theme throughout the application was a pivotal design decision, reflecting the host company's design ethos, and offering an opportunity to answer research question RQ 1 by comparing the complexities of creating said graphical components.

Upon an extensive review of the technical literature, it became clear that there were no ready-to-use libraries or packages to facilitate the implementation of the hexagonal design shown in Figure 3.4. The solution resided in utilizing Jetpack Compose's composable `Canvas` [34]. `Canvas` is a component that enables complex custom visuals to be drawn directly onto the screen [35]. It functions by accepting a path or shape and subsequently rendering it. `Canvas` can also modify properties such as color, stroke, and path effect, and transform the Canvas with operations like rotate, scale, and translate [35].

Listing A.6 demonstrates how Canvas is employed to draw a hexagon. It uses the path returned by the function `drawCustomHexagonPath(size)`, which outlines a hexagon. The properties can then be adjusted, rounding the corners and filling in the hexagon.

This strategy of defining a path and then using `Canvas` also facilitated the creation of the hexagonal bars displayed in Figure 3.4, responsible for user settings. While `Canvas` is an incredibly potent tool, capable of drawing limitless shapes, its intricate nature can pose challenges to novices.

The implementation of the hexagon and hexagonal bars consumed a significant portion of the development timeline. It is worth noting that the completion of this task might not have been possible without Firoz Memon's question posted on Stack Overflow [36], coupled with Phil Dukhov's answer on rounding the corners [37]. These resources proved to be instrumental in the successful creation of the hexagonal design features within the application.

4.1.3.4 Incorporating Application Logic and State Management

In order to make the application functional and responsive, particularly with regards to user interactions and having persisting settings even when the application transitions to the background, it was necessary to introduce application logic and manage state appropriately. As depicted in Figure 3.1, these where some of the application's **Must** requirements.

According to Android's official documentation, state in an app refers to any value that may change over time [38]. When utilizing Jetpack Compose to develop the UI, anytime the state used by a composable is updated a recomposition will be triggered. This means that a composable needs to be explicitly told the new state in order for it to be updated correctly.

Though direct state modifications within the composable responsible for the UI component are possible through the use of Jetpack Compose's `remember` API (as seen in Listing A.7), it is advisable to shift the state responsibilities and associated logic to a dedicated state holder class [38].

State holders serve as repositories for storing application state, allowing the app to access and read this stored state as necessary [39]. When additional logic is required, state holders assume the role of intermediaries, facilitating access to the data sources that host the relevant logic. Through this process, the state holder effectively delegates the execution of logic to the appropriate data source [39].

To achieve this one can utilize a `ViewModel` [40], which exposes the state to the UI and encapsulates corresponding business logic. Importantly, it also caches the state, ensuring its persistence through configuration changes. This functionality eliminates the need for the UI to refetch data during navigation between activities, or upon returning to the application after backgrounding it [40].

Given the complexity associated with managing permissions and accessing the hardware of the phone for step detection, a decision was made to simulate steps with hardcoded data that increments from zero up to a predefined goal. Consequently, there was no ViewModel implemented for this task. Nevertheless, a ViewModel and a data class were implemented to provide the functionality of setting and updating user-related settings, as displayed in Figure 3.1. The specifics of this implementation can be observed in Listing A.8.

4.1.4 Declarative UI Design with Jetpack Compose

As touched upon in Section 2.6.2, Jetpack Compose offers a declarative method to build a UI, similar to building a UI with React while utilizing a framework such as Tailwind or Bootstrap. Jetpack Compose consists of three main layout components, `Row`, `Column` and `Box`, that is used to structure the UI. These components serve as the structure for the UI, accepting various parameters that influence the children placed within them. For instance, these parameters could dictate the alignment of content (either horizontally

or vertically), the padding applied between child elements, and the amount of space that child elements are permitted to occupy [41].

The approach implemented by Jetpack Compose makes UI construction feel intuitive, particularly for developers who have had prior experience with web development. In addition, if Android Studio is used as the development environment, Jetpack Compose offers a beneficial feature: the ability to use `Preview`. This allows developers to both visualize and interact with a composable under development, all without the need for an emulator or a physical device. An illustration of this feature can be found in Figure 4.2.

However, it's important to note that frequent rebuilds may be required when using the `Preview` feature. Despite being faster than reinstalling the application onto a device, it introduces an additional step in the development workflow. Moreover, there might be discrepancies in the color rendering within the preview, which might not precisely match the specified colors for the UI components.

To summarize, Jetpack Compose, when paired with Android Studio, substantially simplifies layout design, mirroring the experience of designing a web-page with a modern web framework.

4.1.5 Animations with Jetpack Compose

Time constraints limited the extent of animations incorporated into the application. Nonetheless, the flip animation applied to hexagons, which display activity tracking data on the front and reveal a backside where the goal can be changed, illustrates the functionality of animations in Jetpack Compose effectively. This operation leverages a public library called Flippable [42] (refer to Listing A.9). This library provides a composable, denoted `Flippable`, which generates a card-like flip view. This view enables the display of distinct composables on its front and back sides and allows for the declarative assignment of flip animation handling instructions, including duration and direction parameters. At its core, the Flippable library is built upon Jetpack Compose's Animation APIs.

Jetpack Compose offers an extensive suite of Animation APIs, many of which are available as declarative composable functions, similar to the `Flippable` composable [43][44]. Figure 4.3 presents a diagram outlining Jetpack Compose's Animation APIs and the appropriate usage contexts. Numerous Animation APIs allow for parameter input to customize their behavior further [45]. The parameter, known as `AnimationSpec`, comes in several forms to facilitate various types of animations. For example, `spring`

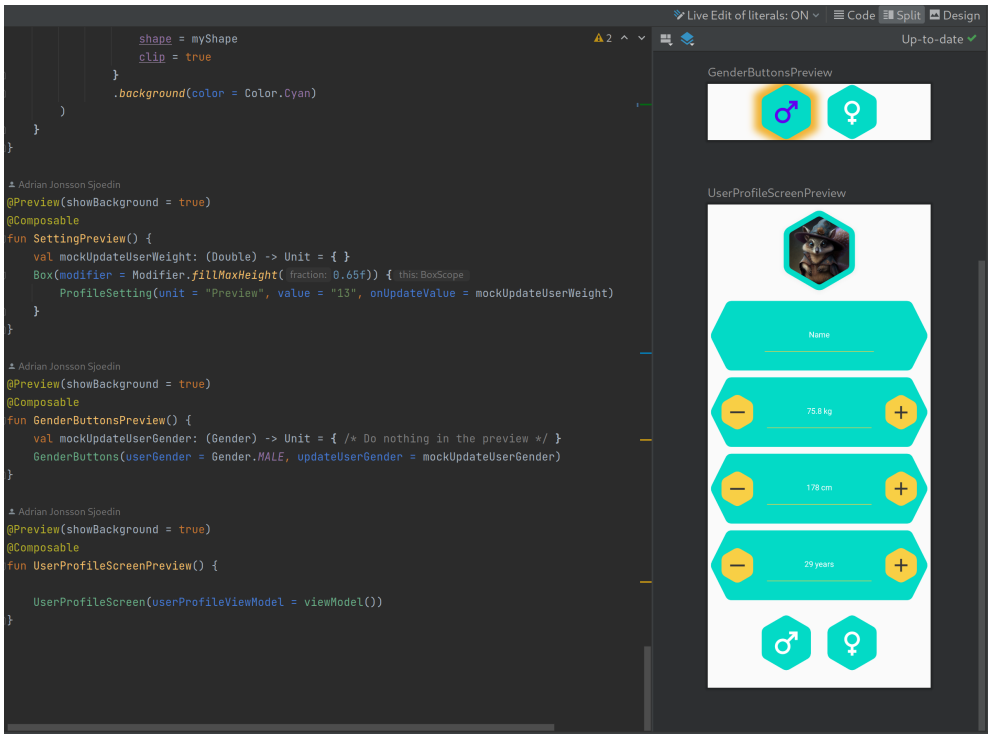

Figure 4.2: Android Studio's preview feature.

enables a physics-based spring animation between start and end values, while `keyframes` animates according to snapshot values specified at different timestamps within the animation's duration [45].

Even though the development process did not necessitate it, Android Studio supports inspection of these Animation APIs: `animate*AsState`, `CrossFade`, `rememberInfiniteTransition`, `AnimatedContent`, `updateTransition`, and `animatedVisibility`, using the Animation Preview tool [46][47]. This tool enables frame-by-frame previewing of transitions, inspection of values for all animations within a transition, and the preview of a transition between any initial and target state. Furthermore, it facilitates the inspection and coordination of multiple animations simultaneously [46]. Figure 4.4 demonstrates the features of this tool.

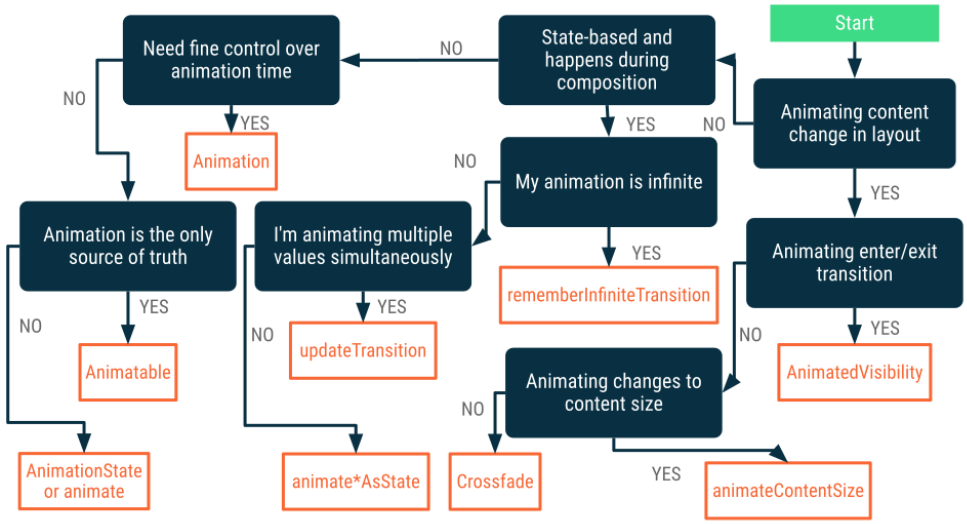

Figure 4.3: Animation flowchart diagram showing which animation API to use [44].

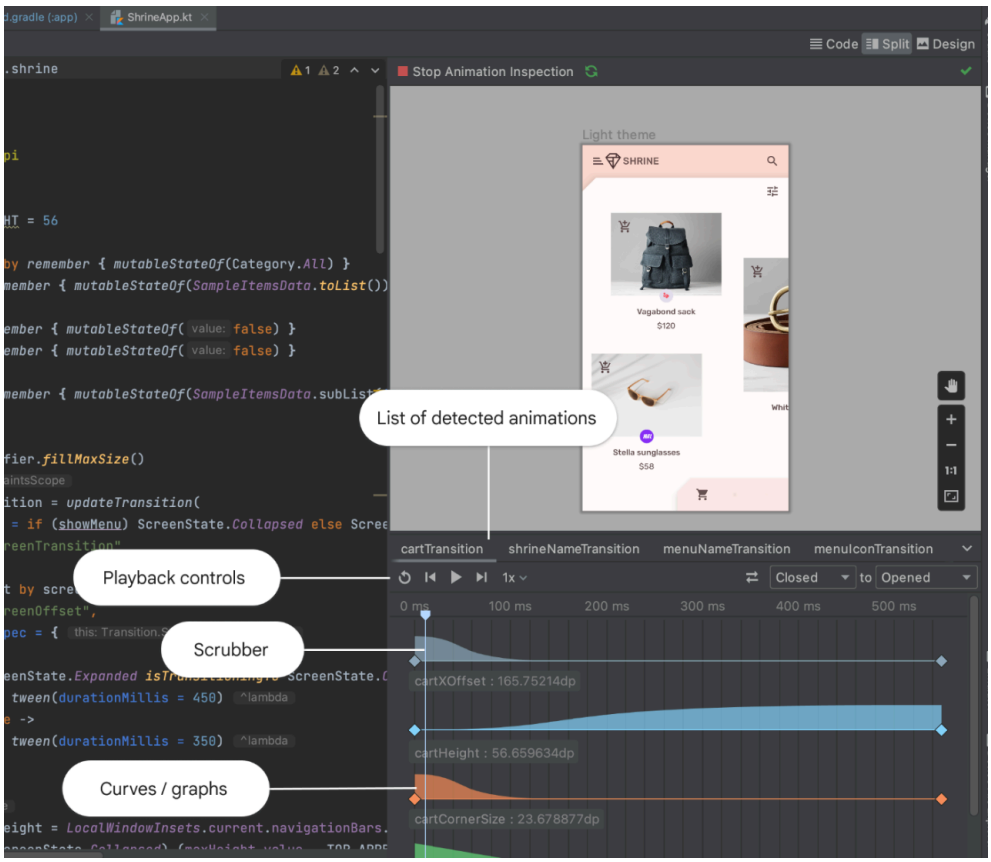

Figure 4.4: Overview of Android Studio's Animation Preview feature [46].

4.2 FlutterFlow Application

This section, mirroring the structure of Section 4.1, delves into the comprehensive development process of the cross-platform application created using FlutterFlow. The initial subsection, akin to its predecessor, explains the process of initiating a new project within FlutterFlow. The following subsection will discuss the approach to conventional software design from the context of low-code tools such as FlutterFlow. The final subsection outlines the construction process of the application.

4.2.1 Initialization Phase

Initiating a new project in FlutterFlow is remarkably straightforward and swift. Within a few minutes of creating an account and logging in, users are presented with an extensive assortment of native FlutterFlow templates. In addition, an in-built marketplace provides an opportunity to explore a wider range of templates and custom components. However, paralleling the Kotlin implementation, this study will utilize the Blank App, as the primary focus lies on the development of the UI. The process is illustrated in the figure Figure 4.5.

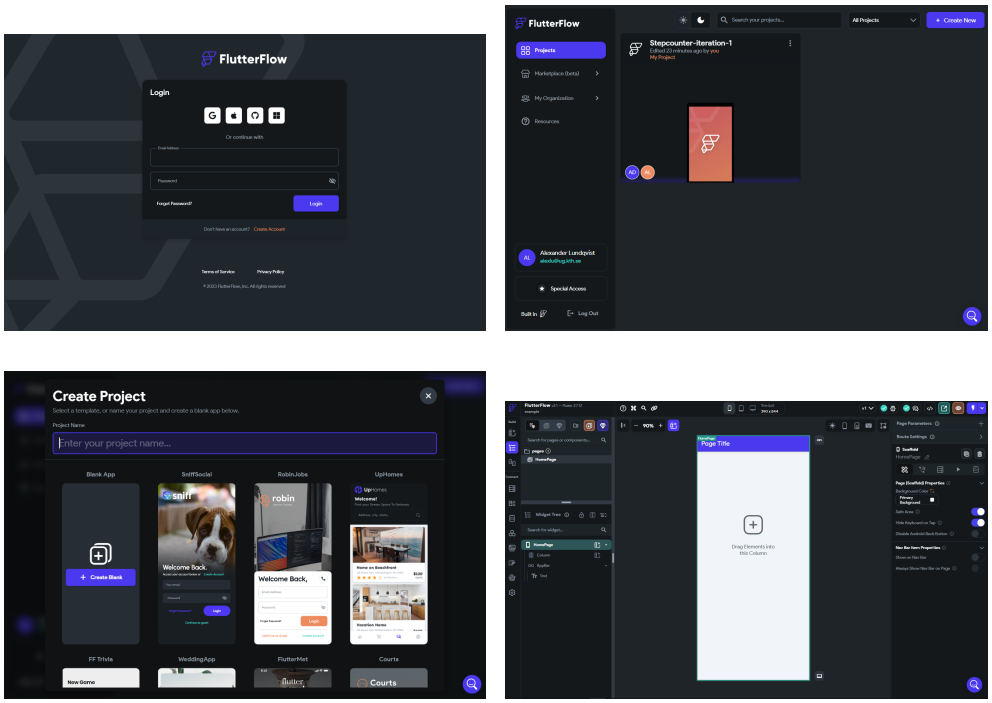

Figure 4.5: New project creation process in FlutterFlow.

4.2.2 Software Design Approach in a low-code tool

FlutterFlow's naming conventions, similar to Flutter, adhere to the Dart naming conventions [48]. Much like Java, this convention favors CamelCase for classes, which in FlutterFlow are represented as custom components or widgets. Moreover, variables should be named in pascalCase, and FlutterFlow will prevent any variable from starting with an uppercase letter.

In terms of documentation, the options available to developers are somewhat limited. Each widget provides the option to add documentation in the form of comments, but this is only possible for the immediate child of a page. Pages themselves do not accommodate comments. Although developers can use Dart doc comments [48] in regular Flutter, this is not an option in FlutterFlow except when writing custom functions, widgets, and actions - as these are the only instances where developers may directly interact with code.

While the Flutter framework, due to its flexibility, allows for the implementation of virtually any architectural design pattern, the same cannot be said for FlutterFlow. Unlike Android Studio, FlutterFlow does not provide immediate choices for file management. Developers have the option to sort their pages and components into folders, but these will eventually only be added to the general lib folder (See Figure Figure 4.6). Essentially, this means that developers do not have any significant control over the architecture.

Therefore, if a specific architectural pattern is crucial and the developer possesses the requisite knowledge, it is possible to implement a particular file organization by refactoring the exported code. However, this approach shifts the focus to the Flutter framework rather than the FlutterFlow tool, and hence, this method will not be considered in the analysis.

4.2.3 Implementation of Design and MoSCoW models

This subsection details the application's implementation in FlutterFlow, discussing how both the functional and design requirements as stated in 3.1 and 3.2 the as well as were met. It follows a similar structure to the earlier section Section 4.1.3.

4.2.3.1 Theme and Color Profile

Establishing a unified color scheme across the application, a requirement drawn from the MoSCoW models (see Figure 3.1), is fortunately a straightforward task in FlutterFlow. A dedicated page named Theme Settings

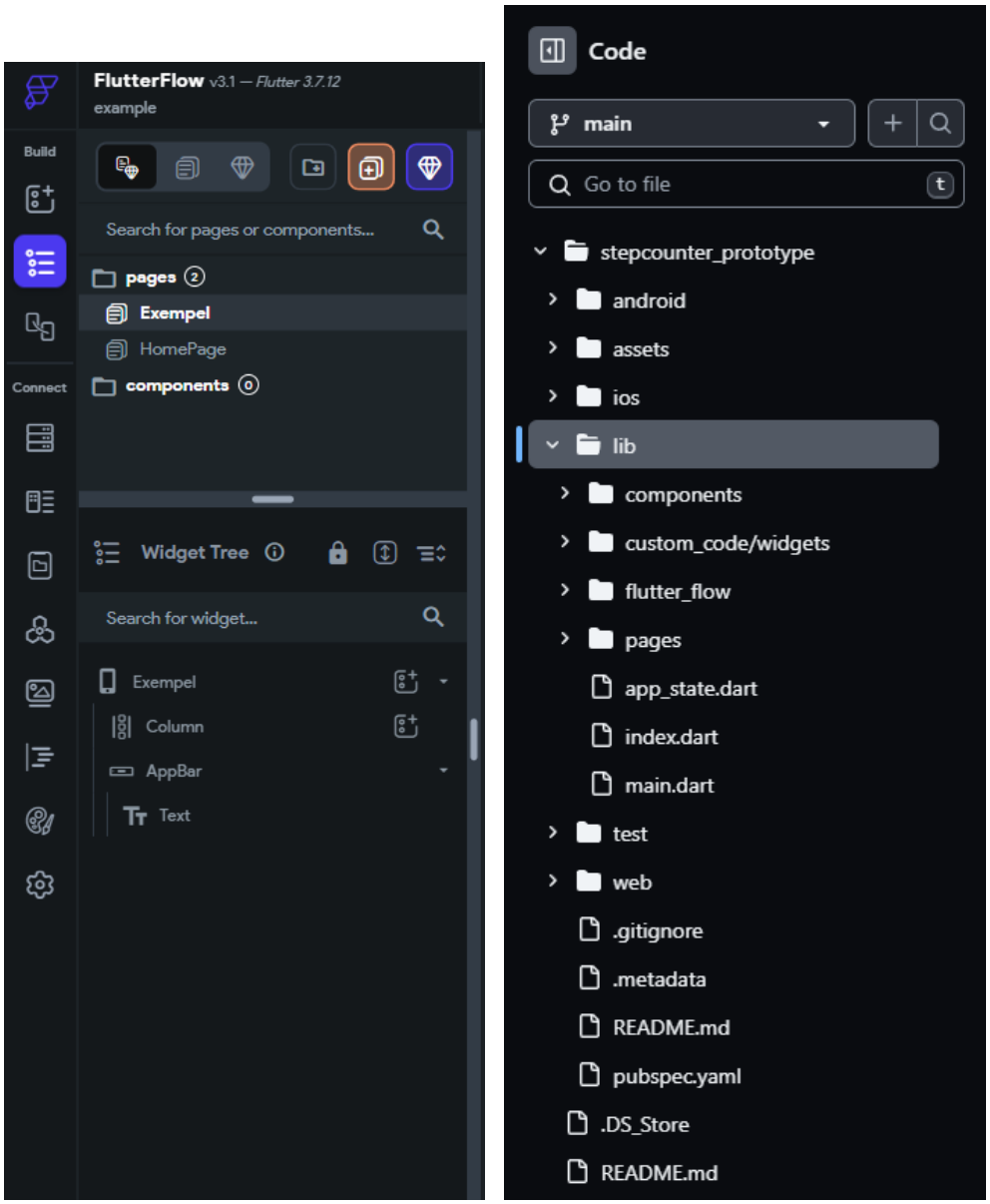

Figure 4.6: Implementation of the navbar.

exists where the Colors sub-page allows the configuration of a consistent color palette that responds to the Dark mode setting (see Figure Figure 4.7).

Similarly, the project-wide typography can be set via the Typography & Icons sub-page, where access to any Google Font is granted, and custom font families can be uploaded. This sub-page also provides the ability to upload custom icons in various formats.

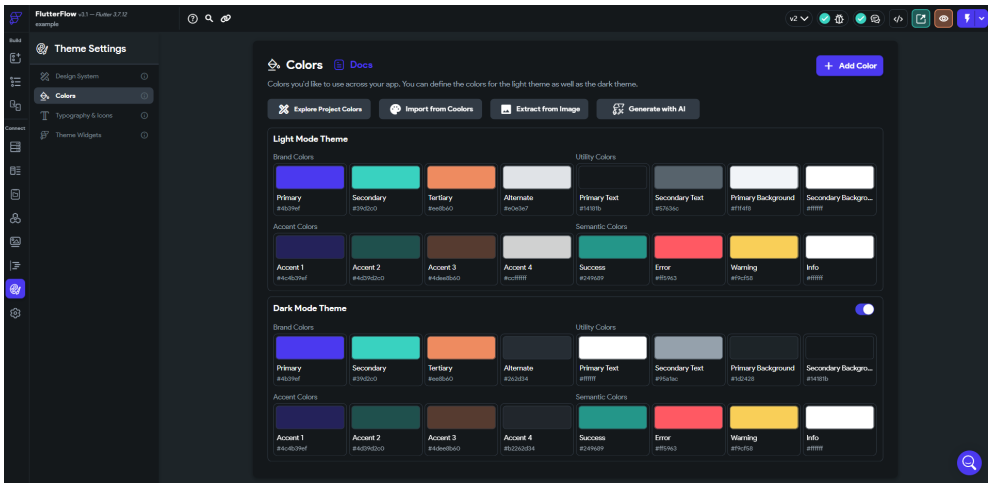

Figure 4.7: The colors sub-page in theme settings.

The Design System sub-page enables the addition of breakpoints, providing developers with a simple method to dynamically manage the layout for a highly responsive user interface. Finally, FlutterFlow facilitates a smooth transition from Figma by offering the capability to import the entire design system, inclusive of colors, typography, and theme widgets. This feature not only ensures design consistency but can also significantly speed up the development process. The Figma prototype that was developed in the beginning 3.4 did not meet up to the expectations of this functionality and thus it was not utilized for this thesis.

4.2.3.2 Navigation

As referenced in Section 4.1.3.2, the initial swipe-based navigation system was rejected in favor of a more traditional navbar. Implementing this change is remarkably easy and straightforward with FlutterFlow. The process (See Figure Figure 4.8) requires going to the App Settings page and then navigating to the NavBar & AppBar sub-page within the general category. From there, activating the navbar and checking the ”Show on Nav Bar” option for each page that needs to be displayed is all it takes.

By default, FlutterFlow employs the standard Flutter navbar which, despite being somewhat limited in terms of customization beyond colors and icons, provides a simple and functional navbar suitable for any modern application. For more advanced customization, FlutterFlow provides the option to use a Google-style navbar, which offers additional options such as animations

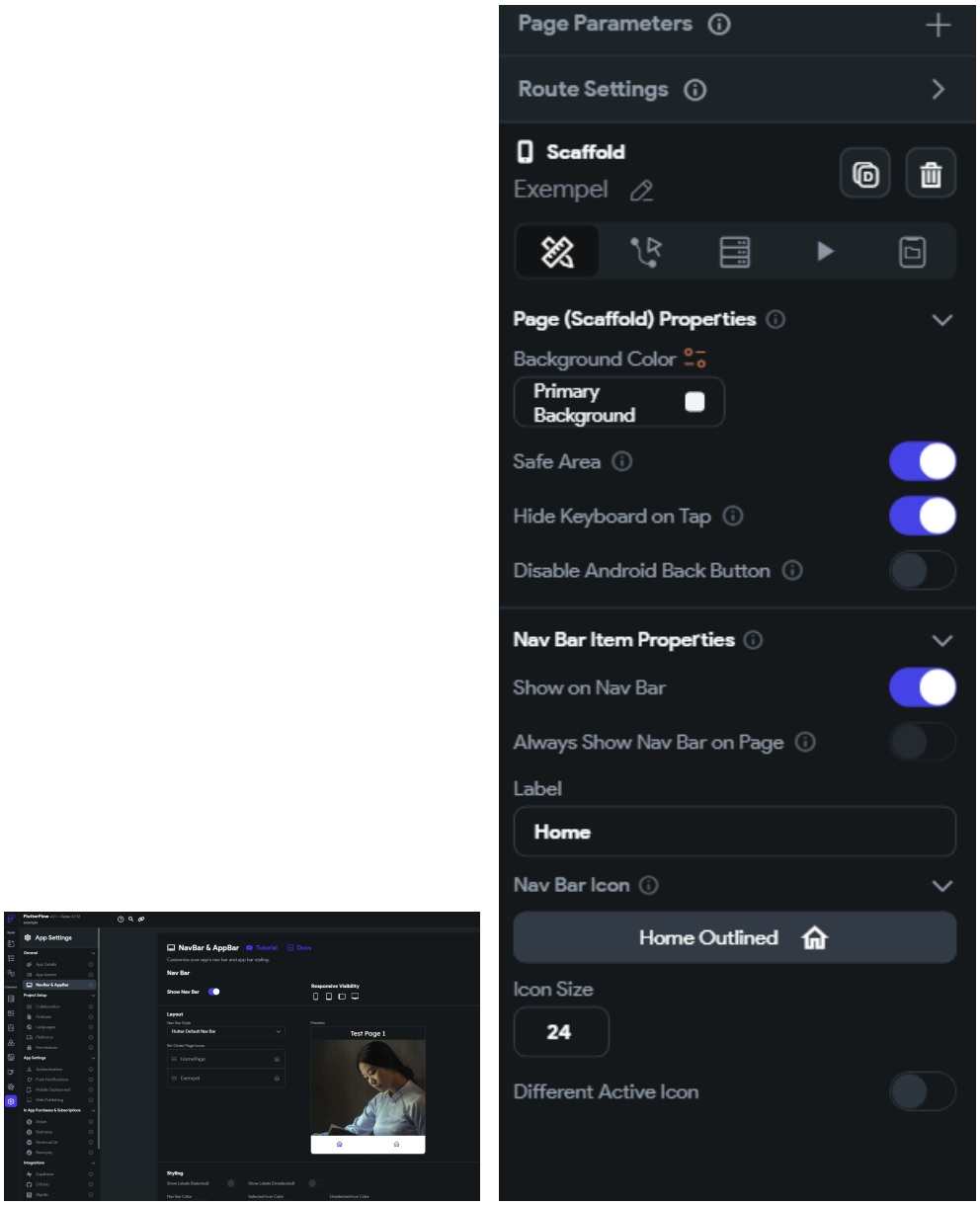

Figure 4.8: Implementation of the navbar.

and highlighting. There's also a floating navbar option, which overlays all pages and remains visible even on pages not listed in the navbar. Beyond these options, FlutterFlow allows the creation of a custom navbar through the creation of a custom component. However, for the purposes of this thesis, the default Flutter navbar was deemed sufficient.

4.2.3.3 Hexagonal Design Implementation

Implementing a hexagon with rounded corners and blur effects posed a significant challenge during the initial review, as there was no straightforward approach in FlutterFlow. While FlutterFlow does offer a similar feature to Jetpack Compose's Canvas Section 4.1.3.3, through the use of the Dart Path class [49], it does not provide a simple method to add rounded corners to a defined path. To accomplish this, a developer would have to craft a rather long and mathematically complex piece of code.

However, the built-in code editor in FlutterFlow noticeably slows down as the length of the code increases. Furthermore, each modification to the code must be recompiled through FlutterFlow's cloud service, a process that takes approximately three minutes. This makes any complex development within FlutterFlow practically unfeasible. While heavy-duty development could be carried out in Android Studio, this option will not be considered as this part of the thesis is focused solely on FlutterFlow.

Fortunately, there is a Flutter package available on pub.dev [50] specifically designed for working with hexagons. To use this package, it must be imported into the code for a custom widget. The package proved invaluable as it provided not only the hexagonal shape but also rounding of the corners as well as the ability to fill the shape with color or an image. The required blur was initially solved by stacking two hexagons then wrapping the bottom hexagon in the `Blur` widget.

No such luck was found for the hexagonal bars, however, as no public package was available that could facilitate this implementation. Faced with the same problem as before and without a clear solution, the goal of implementing rounded corners for the hexagonal bar had to be abandoned.

In summary, creating shapes other than squares or circles in FlutterFlow can be a difficult task. The developer is left to either grapple with manual development through the built-in code editor or to search for a public package that fulfills their requirements.

4.2.3.4 Incorporating Application Logic and State Management

State management in FlutterFlow is governed by a concept referred to as Local State [51]. Similar to Android's official documentation, FlutterFlow defines Local State as "a value that might change over time" [51].

The Local State is segmented into three distinct categories. The first is App State, which has a dedicated page in FlutterFlow. Here, developers can define any type of state variable and decide whether these variables should persist in

the application storage or reset each time the application launches. These state variables are accessible throughout the entire application at all times.

The second category is Page State. These state variables are not globally accessible but exist solely on the page where they are defined. They can, however, be accessed by any widgets or components that are defined as children to the page. Unlike App State, these state variables cannot be saved into persistent storage and will reset when the application closes.

The final category is Component State. The scope of this state variable is even more limited than the Page State, being accessible only within the component it pertains to. Like the Page State, it can also be used by any subsequent child widgets or components.

In the end, FlutterFlow provides the actual implementation of the Local State. The developer's role is to choose the type of state variable and its content, based on its intended use.

Application logic is implemented by combining the use of different state variables and assigning actions to widgets. Actions define the application's behavior [52], ranging from a simple on-click that updates a state variable to complex flows with conditional statements that can potentially trigger actions elsewhere in the application.

Lastly, simulated data is achieved by initializing a timer for each activity tracker when the app starts, which then updates the corresponding tracker variable in the App State at varying time intervals.

4.2.4 UI Design with FlutterFlow

At a glance, Jetpack Compose and FlutterFlow share some similarities in UI construction, such as the use of rows and columns as layout components. However, the way they handle UI creation distinctly sets them apart. In FlutterFlow, UI creation is handled entirely through a visual interface.

The developer starts with an empty screen and constructs the desired layout by dragging and dropping a variety of widgets onto the screen.These widgets serve as the building blocks of the application's UI. The developer can furthermore manipulate the attributes of said widgets such as width, height, alignment, color or padding. The widgets range from common layout widgets such as `Row`, `Column`, `Container` and `Stack` to interactive widgets like `Button`, `Slider` and `TextField` or content-based widgets like `Image` and `VideoPlayer`. This approach allows the developer to create complex UIs without having to delve into the underlying code, making the process both efficient and user-friendly.

One of the unique features of Flutter, often highlighted as a key advantage over other frameworks, is its Hot Reload functionality [53]. This feature allows the developer to see the effects of their code changes almost instantly, without the need to rebuild the entire application. FlutterFlow offers a similar feature known as Instant Reload, which is available when running the application in test mode.

Like Hot Reload, Instant Reload enables developers to make changes to the application's design and functionality without having to rebuild the entire application. However, it differs from Hot Reload in a crucial aspect: while Flutter builds and runs applications locally on the developer's machine, FlutterFlow does this in the cloud. This approach might lead to unstable or unusually high reload times depending on the size of the project and the speed of the developer's internet connection, diminishing the benefit expected of the Instant Reload feature.

4.2.5 Animations in FlutterFlow

Although the initial plan involved an application rich in complex animations, time constraints necessitated a revision of this ambition. However, should time have permitted, implementation would have been straightforward given that FlutterFlow provides every widget the option of adding animations (See Figure Figure 4.9). These animations could be triggered either on page load or by some other trigger invoked within an action flow.

The built-in animations, though quite simple - encompassing basic animations such as fade, slide, and scale - allow for a considerable degree of customization in terms of animation duration, delays, coordinates, and animation curves. There doesn't appear to be a limit on the number of animations you can assign to a widget, and with a bit of patience, it's entirely possible to create intricate animations by combining these basic building blocks provided by FlutterFlow.

It's worth noting that, should a need arise for highly complex or graphical animations, FlutterFlow facilitates seamless integration of both Lottie [54] and Rive [55] animations as they already exists as widgets in FlutterFlow.

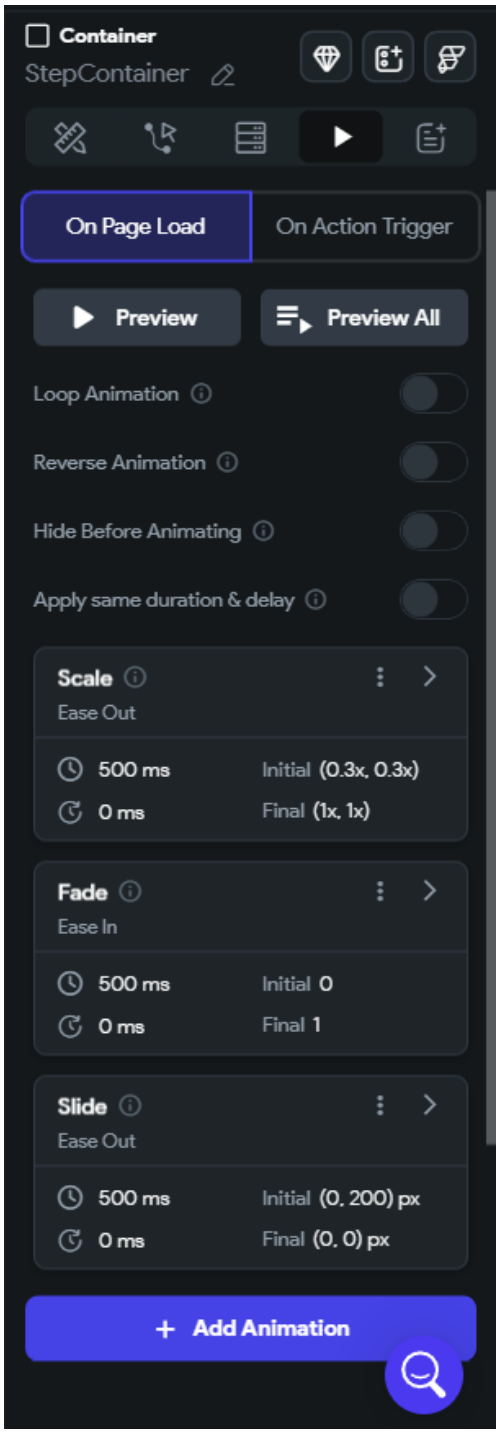

Figure 4.9: The animation options for a widget.

Chapter 5

Results

This chapter presents the results derived from the case study, encompassing both the developed applications and the user survey conducted.

Section 5.1 delves into the native application, highlighting its finalized design, the functionalities implemented, and the duration of the development process. Following this, section 5.2 shifts the focus to the FlutterFlow application, encapsulating the same aspects.

In the last Section 5.3, the findings from the user survey are shared.

5.1 Native Application

The coming subsections delve into the outcomes of the native application development process. The initial subsection reveals the end product, followed by an exploration of the functionalities that were realized, along with those that were eventually disregarded. The final subsection provides a breakdown of the time expenditures linked to the implementation of various functionalities.

5.1.1 Final User Interface Display

The completed UI for the native application is showcased in Figures 5.1 and 5.2, which illustrate the application's appearance in dark and light modes, respectively. In Figure 5.1, the top left image presents the activity screen where all the cards are oriented frontwards. Conversely, the top right image depicts the reversed orientation of the step count card, revealing the backside where the step goal can be customized.

The bottom two images focus on the user profile settings screen. This area is where adjustments to all user-related settings can be made. All buttons

displayed across the application are fully functional, and the hexagons are flippable, providing an interactive user experience.

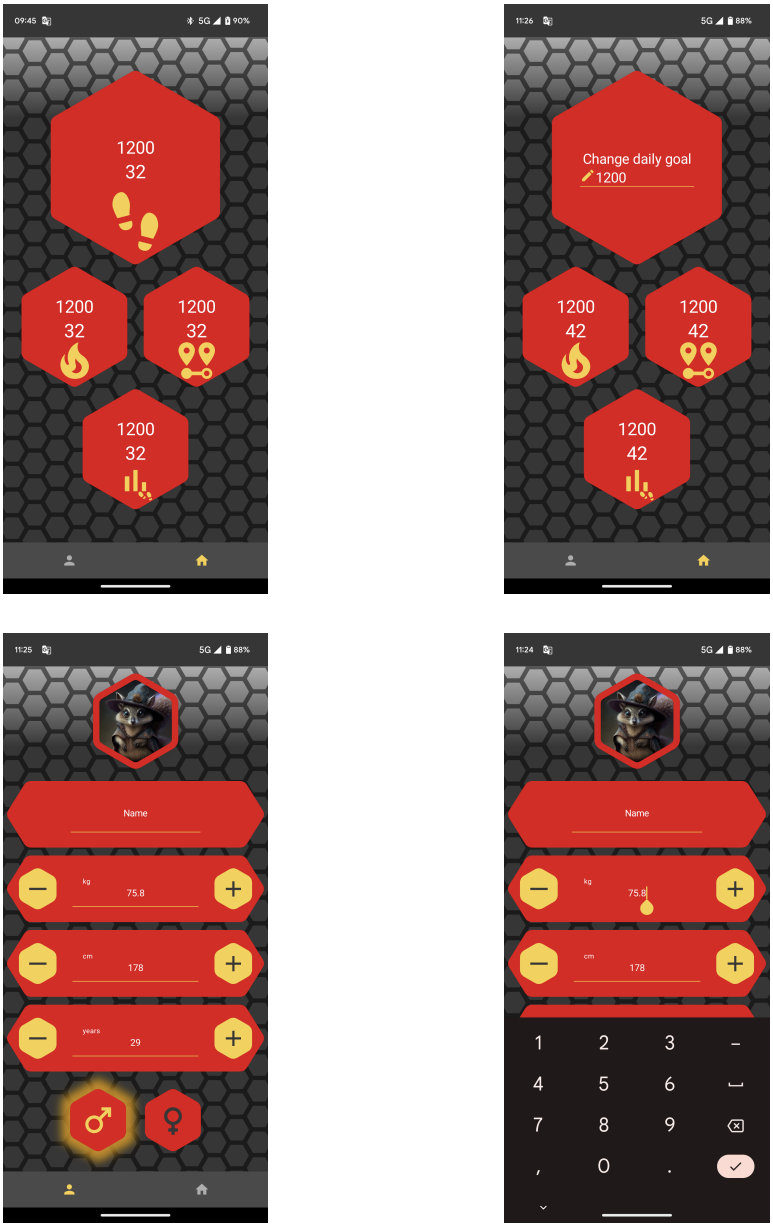

Figure 5.1: The finished native application in dark mode.

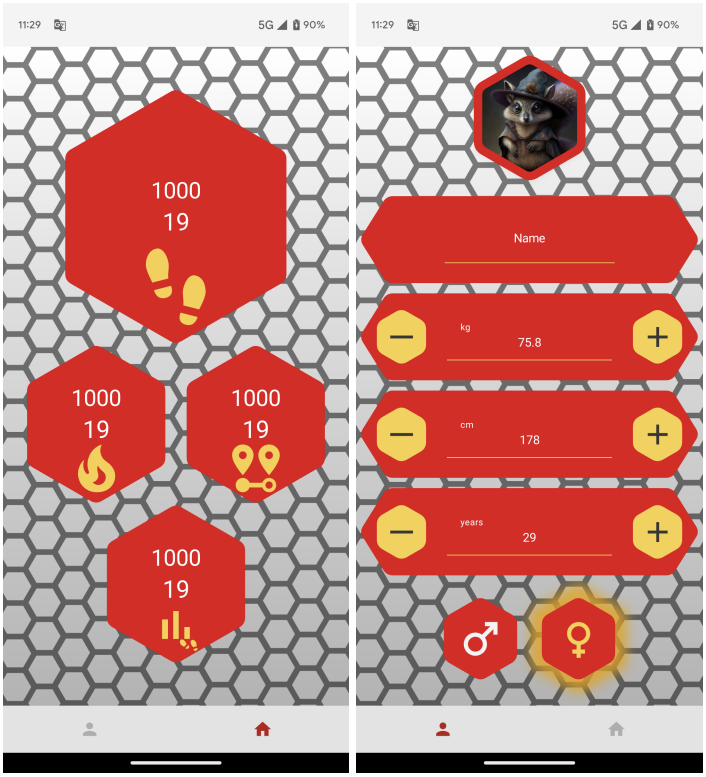

Figure 5.2: The finished native application in light mode.

5.1.2 Functionalities Implemented

As seen in Figure 5.1 a functional UI where implemented, fulfilling the following requirements from the MoSCoW models and high-fidelity design (refer to Figure 3.1, 3.2 and Figure 3.4):

Home screen — This screen acts as the start screen when the app is launched. Displays the activity tracking.

User profile screen — Contains the gender buttons and hexagon bar settings with a textfield for changing the data. Also contains functional buttons to easily increment the data.

Coherent design — A coherent design and color profile was implemented, centered around the host company's graphical profile. Furthermore it adjusts with the device's selected theme.

Work offline	The application does not require a connection to the internet.
Works when backgrounded	The application will not lose it state if backgrounded and then returned to.
Set personal goals	While the application in it current state uses fake data and all activities show the same data and goal, the goal can be changed, and the goal changed depends on the parameter passed to the component. It can thus easily be extended to work for all activities when the logic has been implemented.
Flippable activities	The activities on the home screen can be flipped, and the goal can be changed on the backside.
Background	The hexagonal background pattern with a gradient was implemented.
Glow	Glow can be turned on or off for any of the hexagon buttons.
Animation indication	All buttons have a ripple indication animation when pressed.

None of the **Should** or **Could** goals not listed above where implemented. Furthermore the following from the **Must** requirements and high-fidelity prototype where not implemented:

Store data locally	Data is not stored locally to the device. This means if the application is closed it will reset to the default setting when launched next time.
Notifications	No notifications was implemented because of time constraints.
Swipe based navigation	Implementing swipe based navigation was discarded. See Section 4.1.3.2 for the reasoning.
Count and display steps	Steps are not displayed live. Instead fake data is used to simulate the steps counting up. See end of Section 4.1.3.4 for the reasoning.

Flip activity front — The yellow outline seen in Figure 3.4 that should fill up towards the goal could not be implemented because of time constraints.

Flip activity back — When an activity card was flipped it should move to the center, change to a fix size and blur the background. This was not implemented because of time constraints and the complexity involved.

Flip activity back design — The design as seen in the bottom part of Figure 3.4 could not be implemented because of time constraints. However the button components where created, which should facilitate future implementation.

5.1.3 Development Time

A five-week duration was originally planned for the development of the native application. However, a couple of unforeseen events led to some adjustments in the timeline. Firstly, one week was lost due to the team contracting Covid-19. Secondly, a miscommunication about the project's final deadline cut half a week from the planned schedule, making it necessary to accelerate the development process.

The distribution of development time for executing the application requirements is detailed in Table 5.1. These timings include the necessary periods of research into implementation methodologies and the best practices relevant to each task.

The total lines of code written for the native application is $1,200$ lines, and can be determined by executing the following PowerShell command:

```
(Get-ChildItem -Recurse -Include *.kt | Get-Content |
Select-String -Pattern '^\s*(?!(//|import|\*|/\*\*|\*/))'
| Measure-Object -Line).Lines
```

This command will return the total number of lines of codes for all Kotlin files in the current directory and all its sub-directories, excluding comments and imports. Note however that this command considers only line beginnings and does not account for inline comments or trailing comments on lines of code.

Task description	Jetpack Compose	FlutterFlow
Implement theme and colors	1.5	1
Implementing navigation	3	0.5
Creating base hexagon with glow support and rounded corners	50	60
Creating hexagon bar with rounded corners	10	50*
Creating hexagon button with optional icon and onClick functionality	2	40
Creating setting bars	10	15
Implementing state management for user settings	20	2
Implementing flow for simulating steps	3	6
Implementing flippable cards for activities	16	10
Create layout for the two screens	20	25
Creating ripple effect for button press	2	N/A
Implementing application background with gradient that change based on device theme	3	0.5
Creating documentation comments	1	N/A
Total time	141.5	210

Table 5.1: Development time in hours for different tasks
* the hexagonal bar with support for rounded corners could never be fully realised due to the complexity and difficulty.

5.2 FlutterFlow Application

This section is dedicated to presenting the final outcome of the cross-platform application implementation. Mirroring the structure of the preceding Section 5.1, the initial subsection will unveil the final application. This introduction will be supplemented by a brief subsection highlighting the minor variances in functionality between the two implementations. The concluding subsection will account for the time expenditures required by this implementation.

5.2.1 Final User Interface Design

The FlutterFlow implementation of the UI is presented in figures 5.3 and 5.4 which illustrates the application in both light and dark mode setting. The application has the same functionality as the version implemented with Jetpack

Compose and comparing the figures to 5.1 and 5.2, one can see that they are visually mostly similar.

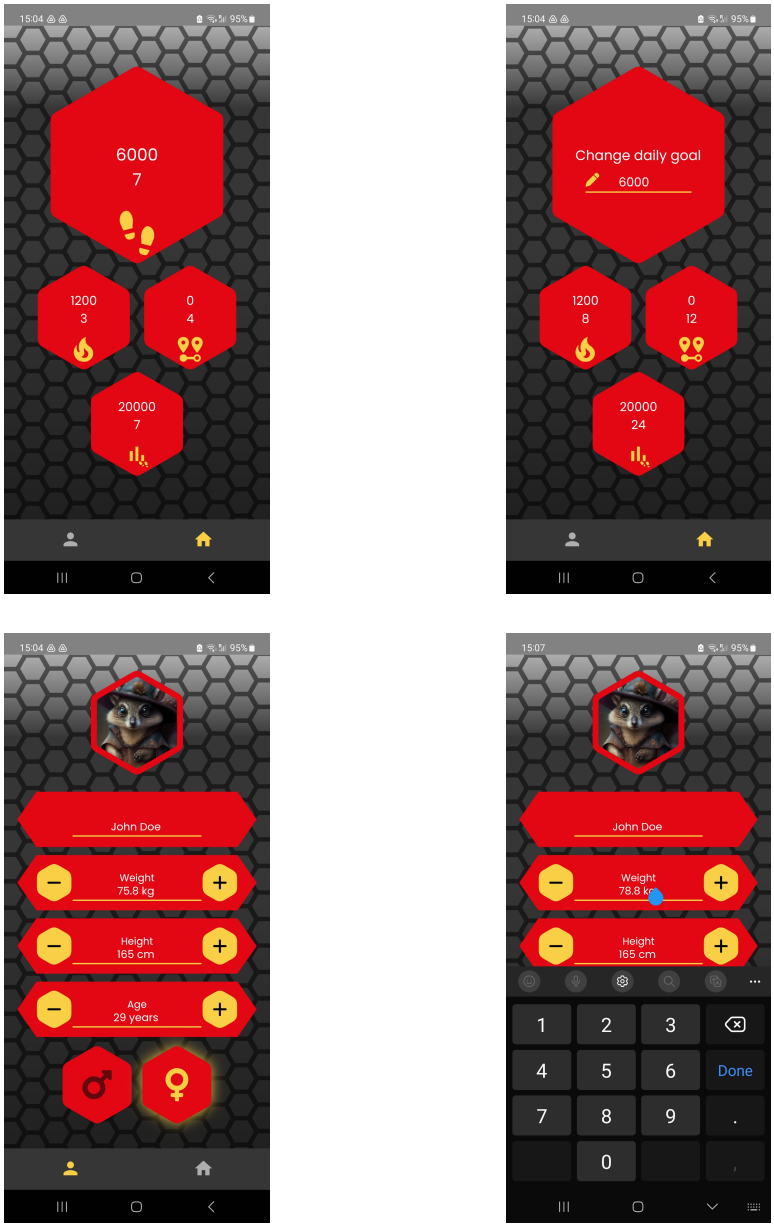

Figure 5.3: The finished cross-platform application in dark mode.

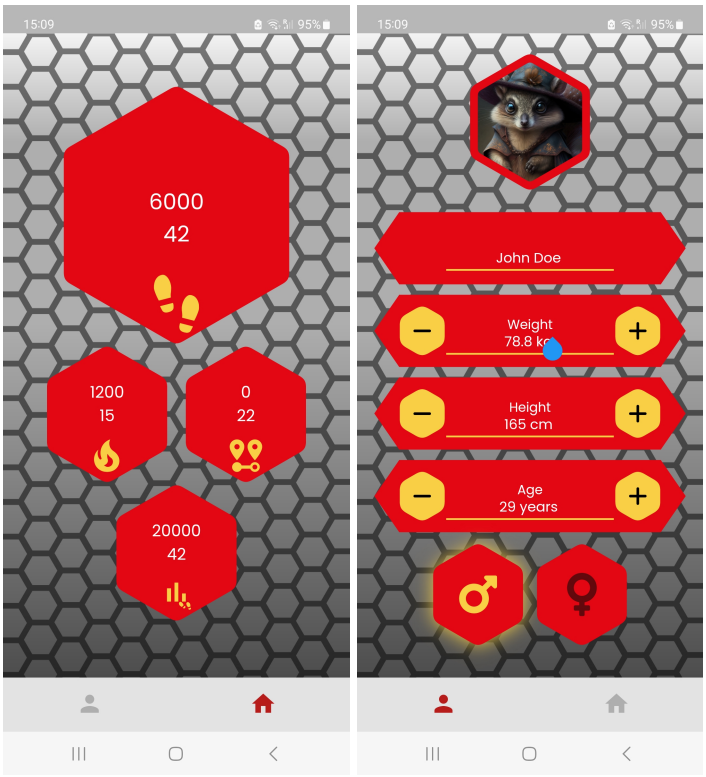

Figure 5.4: The finished cross-platform application in light mode.

5.2.2 Functionalities Implemented

Similarly to the native implementation 5.1.2, the version developed in FlutterFlow also meets the requirements of the MoSCoW model. Therefore it is unnecessary to list these implementations again. However, there is a small discrepancy between the different versions. State management in FlutterFlow differs Kotlin with Jetpack Compose and FlutterFlow more or less requires the App State to have any meaningful functionality and the implementation was rather simple, this version actually stores the data locally.

Store data locally — Data is stored locally to the device. When closing the application or even clearing the cache, the data still persists.

Moreover, this additional animation was not implemented as it was not a part of the MoSCoW model 3.1 or high-fidelity prototype Figure 3.4.

Animation indication — No buttons have a ripple indication animation when pressed.

5.2.3 Development Time

As stated in 5.1.3, the project had some set back regarding the the timeline which led to the final applications containing far less features than was anticipated.

The table 5.1 illustrates the time spent on developing the features of the application. Some of these tasks were almost instantly performed, whilst others took almost the whole development process.

5.3 User Survey Results

Six user surveys were conducted, of which the responses to four are presented in Table 5.2.

Table 5.2: User Survey answers

Survey	**1**	**2**	**3**	**4**
Age	25	28	32	46
Occupation	Computer Science student	Phd student in engineering	Software developer	Software developer
Proficiency with mobile apps	Yes. Both Android and iOS	Yes. Android	Yes. Android	Yes. Android
Comment	A:Compose B:FlutterFlow	A:FlutterFlow B:Compose	A:FlutterFlow B:Compose	A:FlutterFlow B:Compose
B.2.1	Both felt good and intuitive.	Liked it. B felt faster or more fluid	Pretty solid. Liked the icons and that there's not to many levels of options in both	A: It was good but not all icons meaning where clear. Liked the title for settings. B: Don't like proportions of the components on profile page.

Continued on next page

Table 5.2 – *Continued from previous page*

Survey	1	2	3	4
B.2.2	The visual design was appealing, intuitive, and effectively presented the desired information in both.	Liked it. B has better effects when navigating	Both where very clean and not overwhelming. It has a less is more feel to it which is good	Don't like app background. A and B color was nice but fonts where to small
B.2.3	No	No	No	A: The weight doesn't not allow for decimals. The settings don't move up so you can't see the last setting when using the textfield for age. B: No. Frustrating that there's no validation on what values can be entered
B.2.4	No	B felt smoother when using keyboard	Gender pre-selected in A. Prefer to select it myself	No
B.2.5	Some animations in A that is not implemented in B. For example when selecting gender	B. Button has ripple effect. The flip card also turns slower which feels better	No difference	No

Continued on next page

Table 5.2 – *Continued from previous page*

Survey	1	2	3	4
B.2.6	They where both clear and easy to use	Liked the slow card flip and button ripple	Liked the flip function. Really nice animations. Also liked the toggle switch for gender	Liked them both. It was very easy to use and it was fast
B.2.7	No	No	Textfield should not need to be deleted manually	No
B.2.8	They felt the same	B feels more responsive and smoother	A might be slightly faster when swapping pages. Felt snappy and fast	They felt the same
B.2.9	B, because of design	B, because of smoother operations and better visual effects. Mainly the button interaction	Generally A because it has nicer inputs default values that are closer to realistics values. But prefer the light mode in B	A because it had titles in the setting bars.
B.2.10	Flip card should flip back automatically. Title on front of flip card	No	Good app. Less is more feeling and reactive	Should be an option of not selecting a gender. The background could be set by the user. Do not like the current one. Profile pictures should be able to be changed

Chapter 6

Analysis and Discussion

This chapter begins with an analysis of the application development process, succeeded by an interpretation of the user survey results, and concludes with an examination of the scalability of the two distinct implementations. The chapter culminates in a discussion surrounding the omitted surveys.

6.1 Application Development Analysis

The case study conducted demonstrates that Jetpack Compose provides extensive customization options, with the primary restrictions being the developer's skills. Implementing the chosen hexagonal theme and its related components, despite presenting a certain level of challenge due to novice expertise, proved to be completely achievable. It becomes evident that the primary constraints are not the tools themselves, but the developer's knowledge, experience, and creativity, especially when devising custom graphical components. Equipped with the robust toolkit and the integrated features of Android Studio, developers have all they need to construct applications of graphical complexity.

FlutterFlow on the other hand provides a completely different experience, as detailed in Section 4.2.2. Similar to Jetpack Compose, the construction of custom shapes is achievable, although primarily facilitated by external libraries. However, considering the time investment required to create a basic shape such as a hexagon, it is apparent that FlutterFlow lags behind Jetpack Compose, exhibiting a significant disadvantage in this regard.

The integration of navigation within the mobile application using Jetpack Compose, although not overly complex, demands a certain degree of effort, as observed in Table 5.1. By contrast, FlutterFlow merely requires the selection

of a navbar type, its color settings, and the sequence in which the pages should be displayed, consequently giving FlutterFlow an edge regarding usability and implementation time. Nevertheless, when considering flexibility, there is no clear winner, as both Jetpack Compose and FlutterFlow present the option to create custom navbars.

As for the implementation of state management within the Jetpack Compose application, given the authors lack of experience in Android development and the use of Jetpack Compose, the process was somewhat time-consuming, as documented in Table 5.1. Additionally, the current implementation does not save the state within the application's storage, meaning it will reset upon a device restart or similar events. A developer with greater experience would likely expedite the state management implementation process. In contrast, FlutterFlow adopts a more straightforward approach with the concept of Local State as described in Section 4.2.3.4 and though A proficient developer might have greater success in dealing with state management in Jetpack Compose, they can hardly compete against FlutterFlow's App State Page.

Jetpack Compose's Animation APIs provides developers with a comprehensive suite of tools to manifest any desired animation, featuring a variety of customization options to fine-tune animations to exact specifications. The Animation Preview tool integrated into Android Studio further empowers developers to generate precise animations effectively. However, to fully harness the capabilities of Jetpack Compose's Animation APIs, an experienced developer is necessary, given the extensive range of options available and its potential complexity for beginners.

On the other side, FlutterFlow promotes a more accessible approach to animation implementation as discussed in Section 4.2.5. While not as an advanced as what Jetpack Compose offers, it is quite possible to create complex animations by utilizing the basic building blocks. Whether it can rival Jetpack Compose or not in terms of what is possible to make, is undetermined as it would require to put both approaches to the limit.

6.2 User Survey Analysis

The findings depicted in Table 5.2 unequivocally indicate that the perceived differences between the two applications were predominantly design-oriented. The respondents deemed both applications to be responsive, fluid, and quick, demonstrating no appreciable performance disparity. All highlighted disparities were associated solely with distinct design elements featured in each application.

6.3 Scalability Analysis

The scalability of an application developed in Kotlin using Jetpack Compose does not impose any particular additional challenges. As long as the developer remains mindful of scalability during development and ensures the application code is designed to scale, the scalability (or lack thereof) is not influenced directly by Jetpack Compose. Developers retain the flexibility to structure and name code and files as desired, and they also have the capacity to write KDocs. All these factors equip developers with the necessary tools to create a scalable application.

When considering the scalability aspect of FlutterFlow, it is essential first to ascertain whether the application's continued development will remain within FlutterFlow or if the codebase will be exported for further development using other tools. Fortunately, the answer is rather straightforward. If the latter is chosen, then the application's scalability is arguably on par with the Kotlin version, as the exported code consists solely of Flutter, a formidable counterpart to Jetpack Compose. Conversely, if the former option is selected, then based on the discussion in Section 4.2.2, it can be concluded that relying solely on FlutterFlow for development may not be the most effective approach should the application require scalability.

6.4 Discussion

The research conducted for this thesis utilized both quantitative and qualitative data to explore the processes of mobile application UI development. The case study provided an anecdotal account of the development processes while also offering quantitative data regarding time expenditures. This can be used to approximate how the two approaches would measure in a larger scale project.

The user survey on the other hand, served as a source of qualitative data, capturing user perceptions and experiences with the two applications developed using Jetpack Compose and FlutterFlow. This data is essential since whether an application is performance or scalable on paper, it is still the end user's perception that holds most weight.

The first two user surveys were discarded during the data collection process due to incomplete responses or inconsistencies in the data provided. In the first instance, both applications still had quite a lot of bugs and the participant lost focus on the UI evaluation. In the second instance, one of the applications was of the wrong version and had animations implemented. This version was only

meant to be evaluated in the case study, not the user survey and this is suspected to have influenced the participants perceptions of that version. Despite this, the remaining data from the user survey still provided valuable insights into user perceptions of the two applications.

The reliability and validity aspects of the research method are fundamental in any research process, as they ensure that the results are accurate, reproducible and most importantly, provides credibility to the research. This thesis however is of a more qualitative nature and as Noble and Smith suggests [56], this imparts some inherent issues. Due to the time constraints and scope of the thesis, there was no possibility to allocate time and resources to adopt any of the research strategies listed in the paper, other than those already accounted for. Although some quantitative data was gathered during the case study, the method overall focuses on qualitative measurements. However, there can still be argued that the study conforms to a certain degree of validity.

The most glaring fact that speaks against the reliability of the research is that the tool FlutterFlow is constantly being developed, with new versions being released at least once a month since the start of the thesis, bringing new features and changing the development environment. Users have no option to choose which version of FlutterFlow to develop in and subsequently, it is very difficult to replicate the exact research conditions present during the performance of the thesis.

To conclude the discussion, it's important to acknowledge potential sources of error in the study. For instance, the case study was based on just two application development processes, and the findings may not be generalizable to all types of applications or development contexts. Likewise, the study only considers the operating system Android. Additionally, user perceptions can vary widely, and the user survey may not capture all nuances of the user experience as they were kept very short due to the final application being so limited. Finally, the lack of a proper analysis method, such as content analysis or thematic analysis, for the survey data could influence the outcome of the research.

Chapter 7

Conclusions and Future work

This chapter elucidates the conclusions derived from the thesis pertaining to the research questions posed in Section 1.2. It also discusses the limitations that may have influenced these conclusions and concludes with suggestions for potential avenues of future research.

7.1 Conclusions

The research question poised in Section 1.2 that this thesis aimed to answer where:

RQ 1 How do Jetpack Compose and FlutterFlow differ in terms of their flexibility for customization?

RQ 2 Is there a difference in user experience, particularly in the realms of UI responsiveness, animation smoothness, and overall fluidity between applications developed using Jetpack Compose and FlutterFlow?

RQ 3 What is the feasibility of extending the applications developed using Jetpack Compose and FlutterFlow to incorporate additional features?

In relation to RQ 1, the comparative analysis highlighted Jetpack Compose's proficiency in implementing and creating non-standard graphical UI elements such as hexagons, while FlutterFlow demonstrated superiority in creating standard components commonly seen in modern applications.

Moreover, Jetpack Compose outperformed in terms of animation customizability, affording the possibility to realize any animation conceivable by a developer. However, this came with an inherent complexity, making its implementation considerably more challenging compared to FlutterFlow.

As evidenced in Section 4.2.5, FlutterFlow streamlines the animation implementation process, yet does not compromise on the potential for creating more intricate animations.

In response to the second research question RQ 2, the analysis of the results from table 5.2 demonstrates that no substantial differences were discernible between Jetpack Compose and FlutterFlow.

In relation to the final research question RQ 3, the analysis of the case study, as detailed in Chapter 6, clearly indicates that Jetpack Compose provides superior scalability. This is largely due to its lack of the same restrictions that constrain developers when using FlutterFlow.

7.2 Limitations

A significant limitation encountered during the study was the authors' relative inexperience with Android development, Jetpack Compose, and FlutterFlow. Although the authors' background in the field of computer science established a foundation for rapidly gaining familiarity with these new technologies, the limited duration and scope of the study hindered the possibility of delving beyond a superficial understanding.

An additional constraint on the research stemmed from the initial design phase, where the UI design was established and a prototype constructed. While this stage was crucial for executing the case study and providing a UI that the host company could continue to build upon, it unavoidably consumed a significant portion of the allotted time. Consequently, this resulted in less time available for an exhaustive case study and comprehensive user survey.

7.3 Future work

An aspect warranting further exploration involves examining how FlutterFlow could be combined with traditional programming techniques. This examination would investigate whether FlutterFlow could expedite the initial set-up and architectural framework of an application. Once this foundation has been established, the source code could then be exported from FlutterFlow to facilitate the continued development and customization of the UI components.

References

[1] JetBrains, "The six most popular cross-platform app development frameworks," Accessed: 2023-05-23. [Online]. Available: https://kotlinlang.org/docs/cross-platform-frameworks.html [Page 2.]

[2] Google, "Jetpack Compose UI App Development Toolkit," Accessed: 2023-04-20. [Online]. Available: https://developer.android.com/jetpack/compose [Pages 2, 13, 14, and 22.]

[3] Abdalslam, "160+ no-code development statistics, trends and facts 2023," Apr 2023, Accessed: 2023-05-24. [Online]. Available: https://abdalslam.com/no-code-development-statistics?utm_content=expand_article [Page 2.]

[4] A. C. Bock and U. Frank, "In search of the essence of low-code: an exploratory study of seven development platforms," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2021, pp. 57–66. [Page 2.]

[5] ——, "Low-code platform," *Business & Information Systems Engineering*, vol. 63, pp. 733–740, 2021. [Page 2.]

[6] K. Talesra and G. Nagaraja, "Low-code platform for application development," *International Journal of Applied Engineering Research*, vol. 16, no. 5, pp. 346–351, 2021. [Page 2.]

[7] K. Brush, "MoSCoW method," *TechTarget*, Mar 2023, Accessed: 2023-05-25. [Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/MoSCoW-method [Pages 4 and 20.]

[8] D. Maiorca, "What is figma and what is it used for?" *MUO*, Jan 2023, Accessed: 2023-05-25. [Online]. Available: https://www.makeuseof.com/what-is-figma-used-for/ [Pages 5 and 24.]

[9] B. Brauer, C. Ebermann, B. Hildebrandt, G. Remané, and L. M. Kolbe, "Green by app: The contribution of mobile applications to environmental sustainability," 2016. [Page 6.]

[10] D. Eliuseev, "Android: 12 years of design history," *UX Collective*, Apr 2021, Accessed: 2023-05-27. [Online]. Available: https://uxdesign.cc/android-1-0-how-does-it-look-today-476cbe74616a [Page 9.]

[11] M. Tsarouva, "The revolution and evolution of mobile application design," Feb 2019, Accessed: 2023-05-27. [Online]. Available: https://ventionteams.com/blog/revolution-and-evolution-mobile-application-design [Page 9.]

[12] F. Churchville, "user interface (UI)," *TechTarget*, Sep 2021, Accessed: 2023-05-27. [Online]. Available: https://www.techtarget.com/searchapparchitecture/definition/user-interface-UI [Page 10.]

[13] D. Norman and J. Nielsen, "The definition of user experience (UX)," *Nielsen Norman Group*, Accessed: 2023-05-27. [Online]. Available: https://www.nngroup.com/articles/definition-user-experience/ [Page 10.]

[14] experience ux, "What is wireframing?" Accessed: 2023-05-28. [Online]. Available: https://www.experienceux.co.uk/faqs/what-is-wireframing/ [Page 11.]

[15] N. Babich, "Prototyping 101: The difference between low-fidelity and high-fidelity prototypes and when to use each," *Adobe Blog*, Accessed: 2023-05-28. [Online]. Available: https://blog.adobe.com/en/publish/2017/11/29/prototyping-difference-low-fidelity-high-fidelity-prototypes-use [Pages 11 and 12.]

[16] A. Marchuk, "Native vs cross-platform development: How to choose," *up tech*, Accessed: 2023-05-28. [Online]. Available: https://www.uptech.team/blog/native-vs-cross-platform-app-development [Pages 12 and 13.]

[17] S. Singh, "Native vs hybrid vs cross platform – what to choose in 2023?" *net solutions*, Jan 2023, Accessed: 2023-05-28. [Online]. Available: https://www.netsolutions.com/insights/native-vs-hybrid-vs-cross-platform/ [Pages 12 and 13.]

[18] FlutterFlow, "FlutterFlow. A visual development platform," Accessed: 2023-04-20. [Online]. Available: https://flutterflow.io/ [Pages 13 and 22.]

[19] JetBrains, "Faq," Accessed: 2023-04-20. [Online]. Available: https://kotlinlang.org/docs/faq.html [Pages 14 and 22.]

[20] F. Lardinois, "Kotlin is now Google's preferred language for Android app development," *TechCrunch*, May 2019. [Online]. Available: https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/ [Page 14.]

[21] L. Richardson, "Understanding jetpack compose-part 1 of 2," *Medium*, Aug 2020. [Online]. Available: https://medium.com/androiddevelopers/understanding-jetpack-compose-part-1-of-2-ca316fe39050 [Pages 14 and 15.]

[22] G. Thomas, "What is Flutter and Why You Should Learn it in 2020," Dec 2019, Accessed: 2023-04-21. [Online]. Available: https://www.freecodecamp.org/news/what-is-flutter-and-why-you-should-learn-it-in-2020/ [Pages 15 and 16.]

[23] Google, "Dart overview," Accessed: 2023-04-22. [Online]. Available: https://dart.dev/overview [Page 15.]

[24] FlutterFlow, "What is FlutterFlow?" Accessed: 2023-04-24. [Online]. Available: https://docs.flutterflow.io/#what-is-flutterflow [Pages 16 and 22.]

[25] B. G. Asefa, "Building android component library using jetpack compose," 2022. [Pages 16 and 17.]

[26] V. Soininen, "Jetpack compose vs react native–differences in ui development," 2021. [Pages 16 and 17.]

[27] M. Olsson, "A comparison of performance and looks between flutter and native applications: When to prefer flutter over native in mobile application development," 2020. [Pages 16 and 17.]

[28] T. F. E. Team, "Qualitative vs. quantitative data: what's the difference?" Oct 2021, Accessed: 2023-06-05. [Online]. Available: https://www.fullstory.com/blog/qualitative-vs-quantitative-data/ [Page 19.]

[29] D. Rawson, "Idiomatic code. what it is and why it matters." *Medium*, Jul 2020, Accessed: 2023-05-31. [Online]. Available: https://medium.com/swlh/idiomatic-code-a73f17f0f287 [Page 23.]

[30] "Lucidcharts. where seeing becomes doing." Accessed: 2023-05-31. [Online]. Available: https://www.lucidchart.com/pages/ [Page 24.]

[31] "Android studio," Accessed: 2023-05-31. [Online]. Available: https://developer.android.com/studio [Pages 25 and 30.]

[32] A. Gallo, "A refresher on a/b testing," June 2017, Accessed: 2023-06-02. [Online]. Available: https://hbr.org/2017/06/a-refresher-on-ab-testing [Page 26.]

[33] J. Nielsen, "Why you only need to test with 5 users," March 2000, Accessed: 2023-06-13. [Online]. Available: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/ [Page 26.]

[34] "Canvas," Accessed: 2023-06-02. [Online]. Available: https://developer.android.com/reference/android/graphics/Canvas [Page 33.]

[35] J. Shvarts, "Getting started with canvas in compose," Apr 2022, Accessed: 2023-06-02. [Online]. Available: https://www.valueof.io/blog/jetpack-compose-canvas-custom-component-ondraw-drawscope [Page 33.]

[36] F. Memon, "Jetpack compose - custom hexagon border curved edge," Accessed: 2023-06-02. [Online]. Available: https://stackoverflow.com/questions/75019468/jetpack-compose-custom-hexagon-border-curved-edge [Page 33.]

[37] P. Dukhov, "How to draw rounded corner polygons in jetpack compose canvas?" Accessed: 2023-06-02. [Online]. Available: https://stackoverflow.com/questions/69748987/how-to-draw-rounded-corner-polygons-in-jetpack-compose-canvas [Page 33.]

[38] "State and jetpack compose," Accessed: 2023-06-02. [Online]. Available: https://developer.android.com/jetpack/compose/state [Pages xv, 34, and 78.]

[39] "State holders and ui state," Accessed: 2023-06-02. [Online]. Available: https://developer.android.com/topic/architecture/ui-layer/stateholders [Page 34.]

[40] "Viewmodel overview," Accessed: 2023-06-02. [Online]. Available: https://developer.android.com/topic/libraries/architecture/viewmodel [Page 34.]

[41] "androidx.compose.ui overview," Accessed: 2023-06-03. [Online]. Available: https://developer.android.com/reference/kotlin/androidx/compose/ui/package-summary [Page 35.]

[42] "Flippable: A jetpack compose utility library to create flipping composable views with 2 sides," Accessed: 2023-06-13. [Online]. Available: https://github.com/wajahatkarim3/Flippable [Page 35.]

[43] J. Shvarts, "Animation in jetpack compose," July 2022, Accessed: 2023-06-02. [Online]. Available: https://www.valueof.io/blog/animation-compose-api-summary [Page 35.]

[44] "Animations in compose," Accessed: 2023-06-03. [Online]. Available: https://developer.android.com/jetpack/compose/animation/introduction [Pages xi, 35, and 37.]

[45] "Customize animations," Accessed: 2023-06-03. [Online]. Available: https://developer.android.com/jetpack/compose/animation/customize [Pages 35 and 36.]

[46] "Animation tooling support," Accessed: 2023-06-03. [Online]. Available: https://developer.android.com/jetpack/compose/animation/tooling [Pages xi, 36, and 38.]

[47] "Animation preview," Accessed: 2023-06-03. [Online]. Available: https://developer.android.com/jetpack/compose/tooling/animation-preview [Page 36.]

[48] "Effective dart: Style," Accessed: 2023-06-20. [Online]. Available: https://dart.dev/effective-dart/style [Page 40.]

[49] "Path class," Accessed: 2023-06-20. [Online]. Available: https://api.flutter.dev/flutter/dart-ui/Path-class.html [Page 44.]

[50] "Hexagon," Accessed: 2023-06-13. [Online]. Available: https://pub.dev/packages/hexagon [Page 44.]

[51] "Local state," Accessed: 2023-06-13. [Online]. Available: https://docs.flutterflow.io/data-and-backend/local-state [Page 44.]

[52] “Actions,” Accessed: 2023-06-13. [Online]. Available: https://docs.flutterflow.io/actions/actions [Page 45.]

[53] “Hot reload,” Accessed: 2023-06-20. [Online]. Available: https://docs.flutter.dev/tools/hot-reload [Page 46.]

[54] “Lottiefiles,” Accessed: 2023-06-20. [Online]. Available: https://lottiefiles.com/ [Page 46.]

[55] “Rive - build interactive animations that run anywhere,” Accessed: 2023-06-20. [Online]. Available: https://rive.app/ [Page 46.]

[56] H. Noble and J. Smith, “Issues of validity and reliability in qualitative research,” *Evidence-Based Nursing*, vol. 18, no. 2, pp. 34–35, 2015. doi: 10.1136/eb-2015-102054. [Online]. Available: https://ebn.bmj.com/content/18/2/34 [Page 64.]

Appendix A

Supporting materials

Listing A.1: The default Color.kt file created at the start of the development.

```
package com.codelab.thesisexample.ui.theme
import androidx.compose.ui.graphics.Color

val Purple80 = Color(0xFFD0BCFF)
val PurpleGrey80 = Color(0xFFCCC2DC)
val Pink80 = Color(0xFFEFB8C8)

val Purple40 = Color(0xFF6650a4)
val PurpleGrey40 = Color(0xFF625b71)
val Pink40 = Color(0xFF7D5260)
```

Listing A.2: The default Theme.kt file created at the start of the development.

```
package com.sigma.stepcounter.ui.theme
import ...
private val DarkColorPalette = darkColors(
    primary = Purple80,
    secondary = PurpleGrey80,
    tertiary = Pink80
)
private val LightColorPalette = lightColors(
    primary = Purple40,
    secondary = PurpleGrey40,
    tertiary = Pink40
```

```
    /* Other default colors to override
     background = Color.White,
     surface = Color.White,
     onPrimary = Color.White,
     onSecondary = Color.Black,
     onBackground = Color.Black,
     onSurface = Color.Black,
     */
)
@Composable
fun StepCounterTheme(
     darkTheme: Boolean = isSystemInDarkTheme(),
     content: @Composable () -> Unit
     ) {
     val colors = if (darkTheme) {
          DarkColorPalette
     } else {
          LightColorPalette
     }
     MaterialTheme(
          colors = colors,
          typography = Typography,
          shapes = Shapes,
          content = content
     )
}
```

Listing A.3: Jetpack Compose's Scaffold composable.

```
Scaffold(
     bottomBar = {
          BottomNavigationBar(
               items = listOf(
                    BottomNavItem(
                         name = "User",
                         route = "user",
                         icon = Icons.Default.Person
                    ),
                    BottomNavItem(
```

```
                        name = "Main",
                        route = "main",
                        icon = Icons.Default.Home
                    )
                ),
                navController = navController,
                onItemClick = {
                    navController.navigate(it.route)
                }
            )
        },
        content = { paddingValues ->
            Box(
                modifier = Modifier
                    .fillMaxSize()
                    .padding(paddingValues)
            ) {
                Navigation(
                navController = navController,
                userProfileViewModel = userProfileViewModel,
                activityTrackingViewModel = activityTrackingViewModel
                )
            }
        }
)
```

Listing A.4: The `BottomNavigationBar`. Note that some lines had to be split to fit the page.

```
@Composable
fun BottomNavigationBar(
    modifier: Modifier = Modifier,
    items: List<BottomNavItem>,
    navController: NavController,
    onItemClick: (BottomNavItem) -> Unit
) {
val backStackEntry =
navController.currentBackStackEntryAsState();
BottomNavigation(
```

```
    modifier = modifier,
    backgroundColor = MaterialTheme.colors.surface,
    elevation = 5.dp
) {
    items.forEach() { item ->
    val selected =
    item.route == backStackEntry.value?.destination?.route;
    BottomNavigationItem(
        selected = selected,
        onClick = { onItemClick(item) },
        selectedContentColor = MaterialTheme.colors.onPrimary,
        unselectedContentColor = NeutralGrey,
        icon = {
        Column(horizontalAlignment = CenterHorizontally) {
            Icon(
                imageVector = item.icon,
                contentDescription = item.name
                )
        }
        }
    )
    }
 }
}
```

Listing A.5: The Navigation composable.

```
@Composable
fun Navigation(
    navController: NavHostController,
    userProfileViewModel: UserProfileViewModel,
    activityTrackingViewModel: ActivityTrackingViewModel
) {
NavHost(
    navController = navController, startDestination = "main"
) {
    composable(route = "main") {
        ActivityTrackingScreen(
          activityTrackingViewModel = activityTrackingViewModel
```

```
            )

        }
        composable(route = "user") {
            UserProfileScreen(
                userProfileViewModel = userProfileViewModel
            )

        }
    }
}
```

Listing A.6: The Hexagon composable created with the use of Canvas.

```
@Composable
fun Hexagon(
modifier: Modifier = Modifier,
backgroundColor: Color = MaterialTheme.colors.secondary,
) {
BoxWithConstraints(
    modifier = modifier.fillMaxWidth(),
    contentAlignment = Alignment.BottomCenter
) {
    val maxWidth = this.maxWidth
    var canvasSize by remember {
        mutableStateOf(Size.Zero)
    }
    Canvas(
        modifier = Modifier.size(maxWidth)
    ) {
        val height = size.height
        val width = size.width
        canvasSize = Size(width, height)
        val path = drawCustomHexagonPath(size)
        val roundEffect = PathEffect.cornerPathEffect(20f)
        drawIntoCanvas { canvas ->
            canvas.drawOutline(
            outline = Outline.Generic(path),
            paint = Paint().apply {
```

```
                this.color = backgroundColor
                this.style = PaintingStyle.Fill
                this.pathEffect = roundEffect
            })
        }
    }
 }
}
```

Listing A.7: Example illustrating the use of the remember API [38].

```
@Composable
fun HelloContent() {
    Column(modifier = Modifier.padding(16.dp)) {
        var name by remember { mutableStateOf("") }
        if (name.isNotEmpty()) {
            Text(
                text = "Hello, $name!",
                modifier = Modifier.padding(bottom = 8.dp),
                style = MaterialTheme.typography.bodyMedium
            )
        }
        OutlinedTextField(
            value = name,
            onValueChange = { name = it },
            label = { Text("Name") }
        )
    }
}
```

Listing A.8: The UI state class and corresponding viewModel for the user profile related settings.

```
data class ProfileUiState(
    val userWeight: Double = 75.8,
    val userHeight: Int = 178,
    val userAge: Int = 29,
    val userGender: Gender? = null,
    val userName: String = "Name"
```

```
)
/**
 * An enum class representing the user's gender options
 * within the app.
 */
enum class Gender {
    MALE, FEMALE
}
class UserProfileViewModel : ViewModel() {
 private val _profileUiState = MutableStateFlow(ProfileUiState())

 val profileUiState: StateFlow<ProfileUiState> = _profileUiState
                                                    .asStateFlow()

 fun updateUserWeight(newWeight: Double) {
    _profileUiState.value = _profileUiState.value.copy(
                                            userWeight = newWeight
                                            )
 }
 fun updateUserHeight(newHeight: Double) {
    val height = newHeight.toInt()
    _profileUiState.value = _profileUiState.value.copy(
                                            userHeight = height
                                            )
 }
 fun updateUserAge(newAge: Double) {
    val age = newAge.toInt()
    _profileUiState.value = _profileUiState.value.copy(
                                            userAge = age
                                            )
 }
 fun updateUserGender(newGender: Gender) {
    _profileUiState.value = _profileUiState.value.copy(
                                            userGender = newGender
                                            )
 }
 fun updateUserName(newUserName: String) {
    _profileUiState.value = _profileUiState.value.copy(
                                            userName = newUserName
```

```
                                                    )
  }
}
```

Listing A.9: The `FlipActivity` composable making use of the Flippable library to create the hexagon flip activities.

```
@Composable
fun FlipActivity(
activityTrackingViewModel: ActivityTrackingViewModel,
icon: Int
) {
val flipDuration = 400
val flipAnimationType = FlipAnimationType.HORIZONTAL_ANTI_CLOCKWISE
val flipController = remember(key1 = "2") {
    FlippableController()
}
var side by remember { mutableStateOf(FlippableState.INITIALIZED) }
 Flippable(
    flipDurationMs = flipDuration,
    flipOnTouch = true,
    flipController = flipController,
    flipEnabled = true,
    frontSide = {
     HexagonActivityFront(
        value = activityTrackingViewModel
                                        .countUpFlow
                                        .collectAsState(initial = 0)
                                        .value
                                        .toString(),
        goal = activityTrackingViewModel
                                        .getGoalValue()
                                        .toString(),
        icon = icon
     )
    },
    backSide = {
     HexagonActivityBack(
        goal = activityTrackingViewModel
```

```
                                        .getGoalValue()
                                        .toString(),
            setGoalValue = activityTrackingViewModel::setGoalValue,
            currentSide = side
        )
    },
    modifier = Modifier
        .wrapContentHeight(),
    contentAlignment = Alignment.TopCenter,
    onFlippedListener = { currentSide ->
        side = currentSide
        println(currentSide)
    },
    flipAnimationType = flipAnimationType
 )
}
```

Listing A.10: The custom code used to render hexagon shapes in FlutterFlow

```
// Automatic FlutterFlow imports
import '/backend/schema/structs/index.dart';
import '/flutter_flow/flutter_flow_theme.dart';
import '/flutter_flow/flutter_flow_util.dart';
import '/custom_code/widgets/index.dart';
import '/flutter_flow/custom_functions.dart';
import 'package:flutter/material.dart';
// Begin custom widget code
// DO NOT REMOVE OR MODIFY THE CODE ABOVE!

import 'package:hexagon/hexagon.dart';

class BasicHexagon extends StatefulWidget {
  const BasicHexagon({
    Key? key,
    this.width,
    this.height,
    this.color,
    this.imagePath = "",
    this.cornerRadius = 0.0,
```

```
  }) : super(key: key);

  final double? width;
  final double? height;
  final Color? color;
  final String? imagePath;
  final double cornerRadius;

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

class _BasicHexagonState extends State<BasicHexagon> {
  @override
  Widget build(BuildContext context) {
    return HexagonWidget(
      width: widget.width!,
      height: widget.height!,
      color: widget.color,
      cornerRadius: widget.cornerRadius,
      type: HexagonType.POINTY,
      child: widget.imagePath != ""
          ? Container(
              width: widget.width!,
              height: widget.width!,
              child: Image.network(
                                  widget.imagePath!,
                                  fit: BoxFit.scaleDown
                                  ),
            )
          : Container(),
    );
  }
}
```

Listing A.11: The custom code used to render hexagon bars in FlutterFlow.

```
// Automatic FlutterFlow imports
import '/backend/schema/structs/index.dart';
```

```dart
import '/flutter_flow/flutter_flow_theme.dart';
import '/flutter_flow/flutter_flow_util.dart';
import '/custom_code/widgets/index.dart';
import '/flutter_flow/custom_functions.dart';
import 'package:flutter/material.dart';
// Begin custom widget code
// DO NOT REMOVE OR MODIFY THE CODE ABOVE!

class HexagonBar extends StatelessWidget {
  const HexagonBar({
    Key? key,
    this.width,
    this.height,
    required this.backgroundColor,
    this.cornerRadius = 0,
  }) : super(key: key);

  final double? width;
  final double? height;
  final Color backgroundColor;
  final double cornerRadius;

  @override
  Widget build(BuildContext context) {
    return LayoutBuilder(
      builder: (context, constraints) {
        final size = Size(
                          constraints.maxWidth,
                          constraints.maxHeight
                          );
        return CustomPaint(
          size: size,
          painter: HexagonBarPainter(
            size: size,
            color: backgroundColor,
            cornerRadius: cornerRadius,
          ),
        );
      },
```

```
    );
  }
}

class HexagonBarPainter extends CustomPainter {
  final Size size;
  final Color color;
  final double cornerRadius;

  HexagonBarPainter({
    required this.size,
    required this.color,
    this.cornerRadius = 0,
  });

  @override
  void paint(Canvas canvas, Size size) {
    var paint = Paint()
      ..color = color
      ..style = PaintingStyle.fill;

    var path = Path();

    path.moveTo(size.width * (2 / 25), 0);
    path.lineTo(size.width * (23 / 25), 0);
    path.lineTo(size.width, size.height / 2);
    path.lineTo(size.width * (23 / 25), size.height);
    path.lineTo(size.width * (2 / 25), size.height);
    path.lineTo(0, size.height / 2);
    path.close();

    canvas.drawPath(path, paint);
  }

  @override
  bool shouldRepaint(covariant CustomPainter oldDelegate) => true;
}
```

The BibTeX references used in this thesis are attached. 

Appendix B

User Survey Questionnaire

B.1 Task list

1. Start the application by clicking the application icon.
2. When arriving to the home page, set a daily goal for steps.
3. Set another daily goal for calories burned.
4. Navigate to the user profile page.
5. Set a new user name.
6. Update the weight by using the text field.
7. Update the height by using the text field.
8. Update the age by using the increment or decrement buttons.
9. Set the gender by clicking the buttons on the bottom.
10. Close down the application.
11. Set the phone setting to light mode.
12. Start the application again. Navigate to the profile screen.
13. Update the age either with the buttons or the text field.
14. Lastly, change the gender to opposite.

B.2 Questions

B.2.1 Question 1

How would you describe your initial experience with each app?

B.2.2 Question 2

How did you feel about the visual design of each app?

B.2.3 Question 3

Were there any tasks that were particularly easy or difficult in each app? Did you encounter any surprises or frustrations?

B.2.4 Question 4

If you noticed any difference in the interaction with the visual elements, can you describe how you experienced them?

B.2.5 Question 5

Did the animations of the graphical elements differ in any way? If there was any noticeable difference, which version did you prefer?

B.2.6 Question 6

Were there any specific features or interactions that you particularly liked in each version?

B.2.7 Question 7

Were there any specific features or interactions that you particularly disliked in each version?

B.2.8 Question 8

Was there any difference in the overall responsiveness between the applications?

B.2.9 Question 9

In the end, which version did you prefer and why?

B.2.10 Question 10

Do you have any feedback for the application as a whole or a specific version? Is there anything you would like to add, change or remove from the applications? Suggestions for improvements?

TRITA-EECS-EX- 2023:515