FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor’s thesis

Title:	Mobile Android App for Items Giveaway
Student:	Tímea Čitbajová
Supervisor:	Ing. Marek Suchánek, Ph.D. et Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

Numerous portals, services, and applications are dedicated to offering items for free pickup. However, they are often integrated into larger platforms, and users frequently encounter confusion, reliability concerns, and other shortcomings within these services. The goal of this thesis is to develop a mobile application that will aim for ease of use, i.e., giving and receiving items without charge in a reliable and secure way. The development shall follow software engineering methods:

- Analyze the field of giving or receiving items without charge, use conceptual modeling methods, and describe key processes.
- Research existing popular solutions and summarize their shortcomings.
- Set requirements for your own solution in the form of a mobile application for the Android platform and prepare use cases.
- Design an application that meets the specified requirements. Also consider sustainability and extensibility of the solution while designing it. Choose appropriate technologies for implementation with regard to requirements, properly justify the choice.
- Implement a prototype application according to the design, document it, and test it.
- Evaluate the resulting solution and describe further possible development.

Electronically approved by Ing. Michal Valenta, Ph.D. on 18 November 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Mobile Android App for Items Giveaway

Tímea Čitbajová

Department of Software Engineering
Supervisor: Ing. Marek Suchánek, Ph.D. et Ph.D.

May 16, 2024

Acknowledgements

I would like to thank my supervisor, Ing. Marek Suchánek, Ph.D. et Ph.D., for his expert guidance, help, patience, and valuable advice in the completion of this work. My deep gratitude also goes to my family and friends, who supported me both during my studies and throughout this project.

Declaration

In Prague on May 16, 2024

Citation of this thesis

Čitbajová, Tímea. *Mobile Android App for Items Giveaway.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstrakt

Táto práca sa zameriava na vývoj mobilnej aplikácie pre Android, ktorá je navrhnutá na uľahčenie darovania predmetov s cieľom obmedziť znečistenie planéty a prispieť k udržateľnosti. Aplikácia umožňuje používateľom ponúkať a prijímať predmety zadarmo, čím prispieva k zníženiu tvorby odpadu tým, že dáva jej užívateľom možnosť dať nový život veciam, ktoré už nepotrebujú. Projekt využíva metódy softwarového inžinierstva, vrátane analýzy existujúcich riešení, identifikácie požiadaviek a implementácie pomocou Kotlin a Jetpack Compose. Práca skúma dizajn a architektúru aplikácie, so zameraním na použiteľnosť a škálovateľnosť. Výsledkom je funkčná mobilná aplikácia, ktorá demonštruje potenciál podporovať kultúru štedrosti a environmentálnej zodpovednosti prostredníctvom mobilných technológií.

Klíčová slova znečistenie odpadom, darovanie vecí, mobilná aplikácia, Android, Kotlin, Jetpack Compose

Abstract

This thesis presents the development of a mobile Android application designed to facilitate the giveaway of items, aiming to address waste pollution and promote sustainability. The application allows users to offer and receive items for free, contributing to waste reduction by extending the lifecycle of possessions. The project follows comprehensive software engineering methodologies, including analyzing existing solutions, identifying requirements, and implementing Kotlin and Jetpack Compose. The thesis explores the design and architecture of the application, emphasizing usability and scalability. The result is a functional mobile app that demonstrates the potential to foster a culture of generosity and environmental responsibility by leveraging mobile technology.

Keywords waste pollution, Items giveaway, mobile application, Android, Kotlin, Jetpack Compose

Contents

List of Figures

Introduction

In today's consumer-driven world, the rapid turnover of goods has led to significant waste and environmental problems. Traditional methods like recycling are helpful, but they are not sufficient to address the growing waste issue alone. Therefore, innovative solutions that encourage the reuse of items are essential.

This thesis explores the development of a mobile Android application designed to make it easy for people to give away unwanted items. By providing a platform for users to offer and receive items for free, the app aims to reduce waste and promote sustainability. While physical donation centers and thrift stores provide avenues for repurposing goods, a mobile app can make this process more accessible and convenient to a broader audience.

The project employs comprehensive software engineering methodologies, starting with an analysis of existing solutions to identify their shortcomings and draw inspiration from their positive aspects. This is followed by defining the key requirements for the new application. The development process involves designing and implementing the application, ensuring the final product is user-friendly, secure, and scalable.

The primary benefit of the application is its ability to reduce waste while fostering a culture of generosity and environmental responsibility. By making it easy for users to donate and acquire items, the app encourages people to think critically about their consumption habits and the potential of their unused possessions. Additionally, the app aims to build a community of users who are committed to sustainable living practices.

Chapter 1

Goals

The primary goal of this thesis is to develop a mobile application that facilitates giving and receiving items without charge in a reliable and secure manner. To achieve this, the thesis will follow a structured approach using established software engineering methods.

First, the thesis will analyze the current landscape for giving and receiving items for free, using conceptual modeling methods to describe key processes. This will be followed by an investigation of existing popular solutions, identifying their shortcomings, such as user confusion and reliability issues.

Based on the insights gained from this research, the thesis will define the requirements for a new mobile application solution for the Android platform. This step will involve preparing detailed use cases and specifying the necessary features and functionalities that the application must include. The design phase will then focus on creating an application that meets these requirements, emphasizing user-friendliness, sustainability, and extensibility. The choice of technologies for implementation will be carefully justified to align with these goals.

Next, the thesis will proceed to implement a prototype of the mobile application according to the design specifications. The development process will be thoroughly documented, and tested.

Upon completing the prototype, the thesis will evaluate the developed solution, describing further possible development and improvements. This evaluation will provide insights into the application's effectiveness in meeting its goals and identify potential areas for future enhancement.

Chapter 2

Analysis

In this section, the discussion will focus on waste pollution and its impacts. The analysis will explore how the proposed application could serve as a solution to this global issue. Additionally, the concept of item giveaways and existing solutions will be analyzed, highlighting how the proposed application improves upon and fills gaps in current approaches.

2.1 Introduction to the problematics

All living beings produce waste, but due to a large population and industrialized ways of living, people generate an amount of waste that exceeds what nature can handle, sometimes leading to the total destruction of ecosystems. If solid, liquid, and gaseous wastes are not appropriately managed, treated, and disposed of, they present considerable dangers to human health and the environment. [1]

2.1.1 Waste pollution

The difference between waste and pollution is not very significant. However, waste does not necessarily imply harm, whereas pollution is inherently associated with destructiveness. For example, oxygen, essential for humans, is also a waste of photosynthesis but is not considered pollution. On the other hand, carbon dioxide is a byproduct of hydrocarbon combustion. It is connected to global warming, and we all know it harms our environment and planet. Although both are types of waste, we can conclude that not all waste is pollution. [2]

Humans produce a variety of wastes, including plastics, electronic devices, food, metals, and textiles [3]. *According to the World Bank, humans generate over two billion tons of waste every year [4].* This waste typically ends up in landfills, as shown in Figure 2.1, or, more desirably, it is recycled. Unfortunately, recycling is not always an available option. [3]

Figure 2.1: Illustration of landfill [5]

2.1.2 How to deal with the problem

The three pillars of sustainable living are reduce, reuse and recycle. These principles are visually represented in the Figure 2.2. *This Eco Trio are rules that guide us to take care of our planet and use things wisely.* [6]

Recycling involves collecting waste materials and their transformation into new products or materials that can be used again [7]. As previously mentioned, recycling may not always be possible, but doing so with recyclable materials is far more crucial than some might think. Engaging in recycling efforts does more than just lessen landfill dependency. It significantly reduces the demand for new raw materials, along with the energy-intensive extraction, refining, and manufacturing processes. [8]

Recycling undeniably plays a crucial role in minimizing waste, yet the volume of waste we generate remains excessive. We often choose convenience over sustainability. We use disposable cups for our coffee, buy new bags each time we shop, or purchase items we do not actually need. It is important to recognize that while recycling offers benefits, it demands energy for its processes, inevitably resulting in some waste production. This leads to the realization that reducing our purchases and consumption is a more direct and impactful way to support environmental health. [9] *The most effective way to reduce waste is to not create it in the first place [10].*

Another fundamental aspect of sustainable living is reuse. Reuse addresses issues recycling alone cannot solve. The concept of repurposing items is not as complex as it might seem. Often, people overlook the potential for items to serve alternate purposes or to be repaired rather than discarded. Laziness or a lack of awareness leads to items being thrown away when they could be given a second life. For example, clothes could be donated to charity or transformed into cleaning rags instead of being discarded. By exploring ways to reuse, we can significantly reduce the amount of waste we generate. [11]

Figure 2.2: Illustration of 3R [12]

2.1.3 Items Giveaway

Firstly, the project will focus on investigating the origins of the concept of item reuse, examining the role of thrift shops in this context, and discussing how this aligns with the development of the proposed item giveaway application.

2.1.3.1 History

At the beginning of the 19th century, religious groups and charities began collecting and selling used clothing to fund their activities. Later in 1930 this initiative laid the roots for what we now commonly call thrift shopping and started to gain popularity. The Great Depression, a period of economic difficulty, made thrift shopping popular for people looking for affordable ways to meet their needs. As thrift stores expanded their offerings beyond just clothing to include various useful items, their popularity continued to grow. [13]

2.1.3.2 Thrift shops today

Today, thrift stores are like hidden gems, offering various items, from casual and designer wear to one-of-a-kind fashion pieces, along with working electronics, art, furniture, and so much more. The popularity of thrift shopping in recent years is also tied to a growing awareness of consumerism's impact and the importance of sustainability. [13] Indeed, for those with lower income, thrift stores remain crucial, especially in times of increasing inflation. As technology progressed and online activity became the norm, several thrift stores adapted by establishing an online presence, which comes with its own set of pros and cons.

Individuals can donate items to thrift stores, which is essentially how these shops stock their inventory. This act is highly beneficial for donors, as it not only contributes to waste reduction but also helps clear out space in their homes. [14]

2.1.4 Mobile application

Donating to thrift stores can sometimes be inconvenient, mainly if the donation consists of a single item or an item that's too large, making transportation to another location challenging. Additionally, there are cases where individuals may not need the financial return from their donation but still wish to benefit someone else, especially those facing financial difficulties. This motivated the creation of an application aimed at simplifying these challenges, making the process of donating items more convenient and accessible for everyone. Given the earlier discussion on waste pollution, this idea gains even more relevance.

2.2 Market research

This section concentrates on analyzing existing solutions. Examining current solutions is crucial for designing an application that provides improved features and a more user-friendly interface. The outcomes of this analysis play a significant role in establishing the requirements for my application. The selection of applications was based on identifying the most used apps through searches on Google [15] and Google Play [16]. Only those available on Google Play, the official store for Android applications, were chosen. The choice was further refined based on ratings and download numbers.

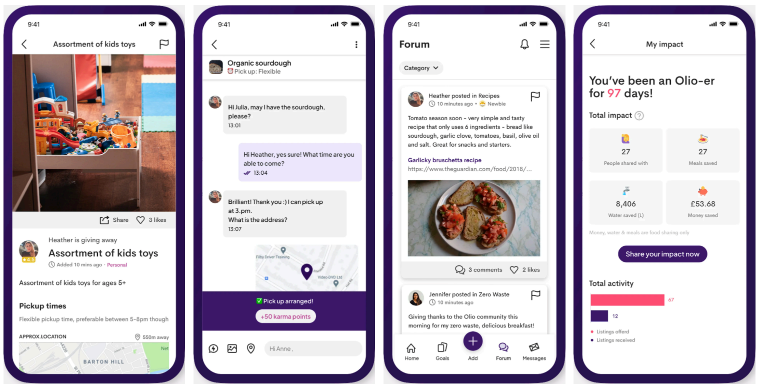
2.2.1 Olio

This app [17] stands out as a popular choice for offering items for free.

It displays vital information for each item: name, description, pickup time and location, along with the posting date to indicate recency. However, the posting time does not guarantee the item's availability. Users can share listings with friends and report issues to developers, though the app reveals excessive user information. Ultimately, the most crucial detail is whether a user is trustworthy. The option to "like" items seems unnecessary, and since it does not function as a way to bookmark them for later, it does not really add any value. The map view is an effective method to display the items' proximity to the user. However, not showing the exact pickup location can be somewhat misleading. The app limits displayed posts to those within a 25km radius, which can be limiting. A beneficial aspect is the ability to customize notifications, enabling users to choose the alerts they receive. While the user interface is aesthetically pleasing, navigation can be somewhat tricky, as shown in the Figure 2.3.

Olio also includes a section for non-free items, which shifts away from its purely free giveaway purpose. A positive feature is the karma points given to users for posting items, which motivates participation. The app includes badges for achievements like First Request, First Collection, etc., which seem redundant given the existing motivational feature. The community feature facilitates communication among users, allowing for exchanging information or inquiries about the app.

Overall, the app functions well and fulfils its intended purpose, but it can occasionally be confusing and cluttered with features. It lacks simplicity, which could greatly enhance user satisfaction.



Assortment of kids toys for ages 5+

Flexible pickup time, preferable between 5-8pm though

Hi Julia, may I have the sourdough, please?
13:01

Hi Heather, yes sure! What time are you able to come?
13:04

Brilliant! Thank you :) I can pick up at 3.pm.
What is the address?
13:07

Tomato season soon - very simple and tasty recipe that only uses 6 ingredients - bread like sourdough, garlic clove, tomatoes, basil, olive oil and salt. Great for snacks and starters.

https://www.theguardian.com/food/2018/...

Giving thanks to the Olio community this morning for my zero waste, delicious breakfast!

You've been an Olio-er for 97 days!

Money, water & meals are food sharing only

Figure 2.3: Illustration of Olio application screens [17]

2.2.2 Freegle

The application [18] allows users not only to make offers but also to submit requests for items they need. This feature enables individuals to ask for specific items, and if someone has the item and wishes to donate it, they can contact the requester directly. However, this may be redundant since those willing to give away items typically list them as offers.

The application displays basic information for each post, including images, title, description, and the number of people interested. It also shows a map with the approximate location for item pickup, which can be helpful, though it might confuse if the listed time is not indicated as either the pickup time or the time the post was made. Users interested in an item can contact the offerer via email or, if registered, through the app's messaging system. Users can also search for giveaway items on a map, which visually represents how far they need to travel to pick up items.

Registration is straightforward, but the application prompts users to provide personal details, which can feel redundant, as seen in the Olio app. In the settings, users can update their information and set up notifications to be received via SMS or email, which may seem outdated.

Additionally, the app includes community subscriptions, primarily based on location, which act as filters to customize viewing available items in specific areas.

The app also includes a "Chit Chat" screen where users can engage in casual conversations and send messages to each other. This feature offers a space for social interaction among users but is not directly related to the app's primary purpose of facilitating item giveaways.

At first glance, the application seems to have an old-school design, as depicted in the Figure 2.4. Overall, while the app functions adequately, it could benefit from improvements in simplicity and the user interface to enhance the overall user experience.

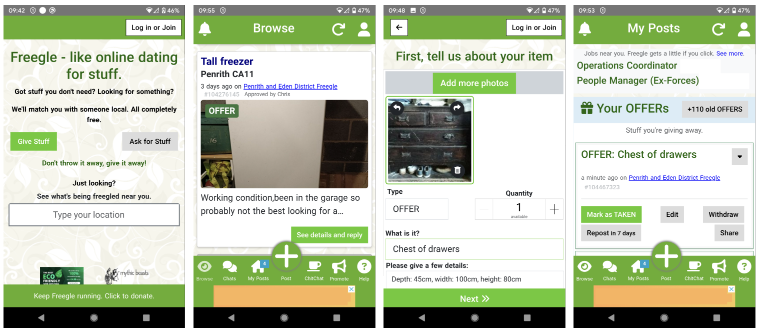

Log in or Join
Freegle - like online dating for stuff.
Got stuff you don't need? Looking for something?
We'll match you with someone local. All completely free.
Give Stuff
Ask for Stuff
Don't throw it away, give it away!
Just looking?
See what's being freegled near you.
Type your location
Keep Freegle running. Click to donate.
Browse
Tall freezer
Penrith CA11
OFFER
Working condition,been in the garage so probably not the best looking for a...
See details and reply
First, tell us about your item
Add more photos
Type
Quantity
OFFER
What is it?
Chest of drawers
Please give a few details:
Depth: 45cm, width: 100cm, height: 80cm
Next
My Posts
Operations Coordinator
People Manager (Ex-Forces)
Your OFFERs
+110 old OFFERS
Stuff you're giving away.
OFFER: Chest of drawers
Mark as TAKEN
Edit
Withdraw
Repost in 7 days
Share
Browse
Chats
My Posts
Post
ChitChat
Promote
Help

Figure 2.4: Illustration of Freegle application screens [18]

2.2.3 Thrash nothing

The design of this application [19], as shown in Figure 2.5, closely resembles that of the Freegle application. It also equally lacks a modern aesthetic. The app Thrash Nothing offers functionalities similar to those of Freegle, allowing users to add both offers and requests. Each post includes images, a title, a description, and collection times. Additionally, the app displays the offerer's membership duration and other postings, though these details may not sufficiently establish the user's trustworthiness. Users interested in a post can bookmark it, share it, or report any issues, but they must be logged in to send a message to the giver.

Like the community features in the Freegle app shown in Section 2.2.2, this application also supports groups where users can connect, functioning similarly.

Beyond item exchanges, Thrash Nothing enriches its community engagement by allowing users to share personal stories, as well as read and contribute to blogs. However, these features, much like those in the Freegle app, do not directly relate to the primary function of the app.

Nevertheless, this application could benefit from a more streamlined interface and improved visual design to enhance usability and aesthetic appeal, making it more approachable and enjoyable for users.

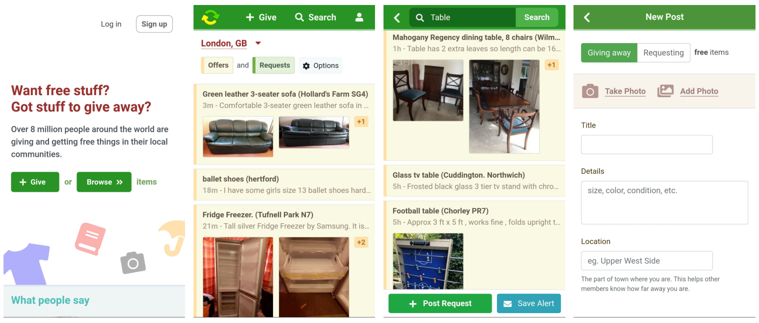

Log in
Sign up
Want free stuff?
Got stuff to give away?
Over 8 million people around the world are giving and getting free things in their local communities.
Give
or
Browse
items
What people say
Give
Search
London, GB
Offers
and
Requests
Options
Green leather 3-seater sofa (Hollard's Farm SG4)
ballet shoes (hertford)
Fridge Freezer. (Tufnell Park N7)
Table
Search
Mahogany Regency dining table, 8 chairs (Wilm...
Glass tv table (Cuddington. Northwich)
Football table (Chorley PR7)
Post Request
Save Alert
New Post
Giving away
Requesting
free items
Take Photo
Add Photo
Title
Details
size, color, condition, etc.
Location
eg. Upper West Side
The part of town where you are. This helps other members know how far away you are.

Figure 2.5: Illustration of Thrash Nothing application screens [19]

The analysis of existing solutions has highlighted areas for improvement. A design that is both visually appealing and easy to navigate is crucial in improving user satisfaction. Motivating users encourages regular engagement with the application, yet it is crucial to maintain a balance without being overly forceful. Considering the personal interaction between users, implementing a rating system is vital for ensuring the safety of participants.

2.3 Requirements

To begin the implementation of the application, it is important to establish precise requirements to outline the necessary tasks. Carefully considering these requirements is essential for the success of any software product, especially given that research indicates a 68 percent failure rate for software projects. Requirements are typically divided into two categories: Functional and Non-Functional. [20]

2.3.1 MoSCoW prioritization

The MoSCoW method is a popular project management and software development prioritization technique. This acronym stands for four categories of priority: "Must have", "Should have", "Could have", and "Won't have," which help distinguish the essential features or tasks from those that are less critical.

The simplicity and clarity of the MoSCoW method ensure that all involved parties easily understand project priorities. Additionally, its flexibility allows it to be effectively applied across various project types and industries, making it a universally adaptable tool. [21] Based on [22], we can define the categories as follows:

- **Must have** requirements are critical to the project's success, as they define its core functionality and purpose. Without meeting these, the project cannot be successfully executed.
- **Should have** requirements are significant yet not completely necessary for the project's development. While the project can succeed without them, they enhance the overall quality and should be included if feasible.
- **Could have** requirements are desirable but not essential. They represent nice-to-have elements that do not impact the project's core functionality. Although not critical to the project's success, it could be implemented if time and resources allow.
- **Won't have** requirements are intentionally omitted from the project scope. Some project features may be infeasible, too expensive or inconsistent with overall goals.

2.3.2 Functional requirements

Functional requirements detail the specific actions and capabilities the software system must have. These specify the necessary functions the software should perform to achieve its goals. They emphasize the features and processes with which users will engage. Functional requirements ought to be specific, measurable, and verifiable, ensuring they clearly demonstrate to developers that the software operates as intended. [23]

F1. User registration (Must have): The user can register a new account for the application.

F2. Login user (Must have): Users can log into the application using their email and password if registered.

F3. Log out user (Must have): Users can log out of the application when logged in.

F4. Display other users' posts (Must have): A list of other users' posts for giving items away is shown.

F5. Display current users' posts (Must have): A list displaying the current user's posts for giving items away is shown.

F6. Display post detail (Must have): The post detail for giving items away is shown.

F7. Delete post (Must have): Users can delete any of their own posts.

F8. Add a new post for giving away an item (Must have): The user can add a new post.

F9. Edit existing post (Must have): The user can edit existing post.

F10. Display user profile (Should have): The current user profile is displayed to the current user.

F11. User profile update (Should have): The user has the ability to modify their personal information.

F12. Delete an account (Should have): The user has the ability to delete their account.

F13. Send messages to other users (Should have): The user can send a message to another user to show interest in an item via a post.

F14. Display conversations between the current user and other users (Should have): The user can view a list of all their conversations with other users.

F15. Display messages between current user and other users (Should have): The user can display a list of all the messages with other users.

F16. Save other users posts (Could have): The user can save other users' posts for possible interest in the item.

F17. Display saved posts (Could have): The user can view a list of all the posts they have saved.

F18. Rate other users (Could have): When exchanging messages with another user, it is possible to rate that user.

F19. Display terms and conditions (Could have): The user can display the terms and conditions of the application.

F20. Mark post reserved (Could have): The user can mark their own post as reserved.

F21. Mark post picked up (Could have): The user can mark their own post as picked up.

F22. Display map with pick up location (Could have): The user can view a map showing the pickup location of an item.

F23. Messaging other than through application (Won't have): The user can contact another user through means other than the application's messaging system, such as email or phone number.

F24. Selling items for money (Won't have): The user can sell the items they offer through the application.

2.3.3 Non-functional requirements

Non-functional requirements describe the software system's qualities and attributes, focusing not on what the software does but how it performs. While functional requirements concentrate on the software's actions, non-functional requirements address its performance and behavior. These requirements ensure the software's overall quality and success, yet they often need more attention in software development projects. However, they are just as critical as functional requirements because they profoundly affect how effective the final product is and how satisfied users will be with it. [23]

N1. Android platform (Must have): The platform of the final product must be Android.

N2. Extendability (Must have): The application should be designed to be easily extendable for future development.

N3. Simplicity (Must have): The application should feature a simple design and be easily navigable.

N4. Modularization (Could have): The application is divided into modules for better coherence.

2.4 Use cases

A use case explains the interaction between an actor and a system, specifying the intended functions of an application. It is crucial at the start of a project as it helps outline the required tasks and identifies user needs. By thoroughly planning through use cases, time can be saved by preventing the need for significant revisions after development has begun, ensuring that the software meets its intended purposes from the outset. [24]

2.4.1 A list of use cases

This section is dedicated to discussing the use cases for this project. Below the list is a diagram Figure 2.6 that visualizes these use cases.

UC1. **Registration:** Upon opening the app, a new user needs to create a new account before logging in. To register, the user clicks on the "Register" button and must provide their first name, surname, username, email and password. After entering all the necessary information, the user clicks the "Create an account" button. If all details are correctly filled out, the app will confirm the successful creation of the account. If the registration is unsuccessful, the app will alert the user to any errors that need to be addressed.

UC2. **Log in:** Users can log into the application by entering their email and password for authentication. After filling in both text fields, the user clicks the "Log in" button. The application will proceed to the home screen if the email and password are correct. If either or both credentials are incorrect, the user will receive a notification, and corrections will be required to proceed.

UC3. **Log out:** Users can log out of the application if logged in by pressing a "Log out" button in the Settings section. If the logout proceeds correctly, the application will display the login screen. However, if a problem exists, such as a missing internet connection, the application will notify the user that the logging out was unsuccessful.

UC4. **Display other users' posts:** On the home page, a user can view posts from other users. Each post displays an image, title, a preview of the description, and a "reserved" badge if marked as such. If the application successfully loads the posts, they appear on the screen. If the loading fails, the system notifies the user and allows reloading the screen.

UC5. **Display user's posts:** A user can view a list of their own posts on a dedicated page within the application. Each post displays an image, title, a preview of the description, and a "reserved" or "picked up" badge if marked as such. If the application loads the posts successfully, they appear on the screen. If the loading process fails, the application will notify the user and offer an option to reload the screen.

UC6. **Display other user's post detail:** When a user clicks on any of the posts on the "Home" screen, the details of the selected post are displayed. The user can see images, the title, the description, the pickup time, the pickup location, and the rating of the post's owner. If the user is interested in the post, a button will be available to send a message to the owner of the post.

UC7. **Display the user's post detail:** In the "My Posts" screen, selecting one of their posts allows a user to view its details, including images, title, description, pickup time, pickup location, and the owner's rating. Additionally, there are options to mark the post as reserved or picked up, accessible through corresponding buttons.

UC8. **Edit a post:** Users can edit a post they have previously added by updating the title, description, pickup place, pickup time, and images. After making the necessary changes, the user must click the "Save changes" button to apply them. If any field is left empty, the user is notified and required to fill it out before the changes can be saved. Once the changes are successfully saved, the user is redirected to the "My Posts" screen and receives a notification confirming that the post has been updated.

UC9. **Delete a post:** A post can be deleted by its owner by navigating to the post's detail page and clicking on the delete button. If the deletion is successful, the user is redirected to the "My Posts" screen and receives a notification confirming the deletion. If the deletion is unsuccessful, the user is notified and remains on the post detail page.

UC10. **Add a new post:** A new post can be created by navigating to the "My Posts" screen and clicking on the plus button. It is mandatory to fill in the title, description, pickup place, pickup time, and images. After completing the form, the user clicks the button to add the new post. If the post is successfully added, the application redirects the user to their posts screen. If the post cannot be added, for instance, if not all fields are filled in or another error occurs, the user receives a notification and has the opportunity to correct the mistakes.

UC11. **Display all conversations:** When a user initiates a conversation with another user after expressing interest in a post, a new conversation appears in the "Messages" screen. This conversation item displays the other user's username, and a preview of the latest message exchanged.

UC12. **Display and send messages:** In the "Messages" screen, users can view their entire conversation history with another user by clicking on a specific conversation thread. The messages are organized chronologically. To send a new message, a user simply types into the provided text field and clicks the send button. If sent successfully, the new message will appear at the bottom of the list of messages. If the user attempts to send an empty message, a notification will appear, prompting them to enter text before sending.

UC13. **Rate user:** When initiating a conversation with another user, there is an option to rate the other party. Users can assess each other based on trustworthiness or responsiveness by clicking the "Rate User" button found within the conversation details.

UC14. **View profile:** When users navigate to the "Settings" screen, they can select the "Profile" option to view their personal information. The profile displays details such as first name, surname, and username, which users can update, although the email used for authentication cannot be changed. Users can also view their current rating and must click the "Save changes" button to save any updates made. If a user chooses to delete their account, they can do so by clicking the designated button for account deletion. Once the account is successfully deleted, the user will be automatically redirected to the login screen.

The use case coverage of functional requirements is shown in the Appendix A.

2.4.2 Diagram of use cases

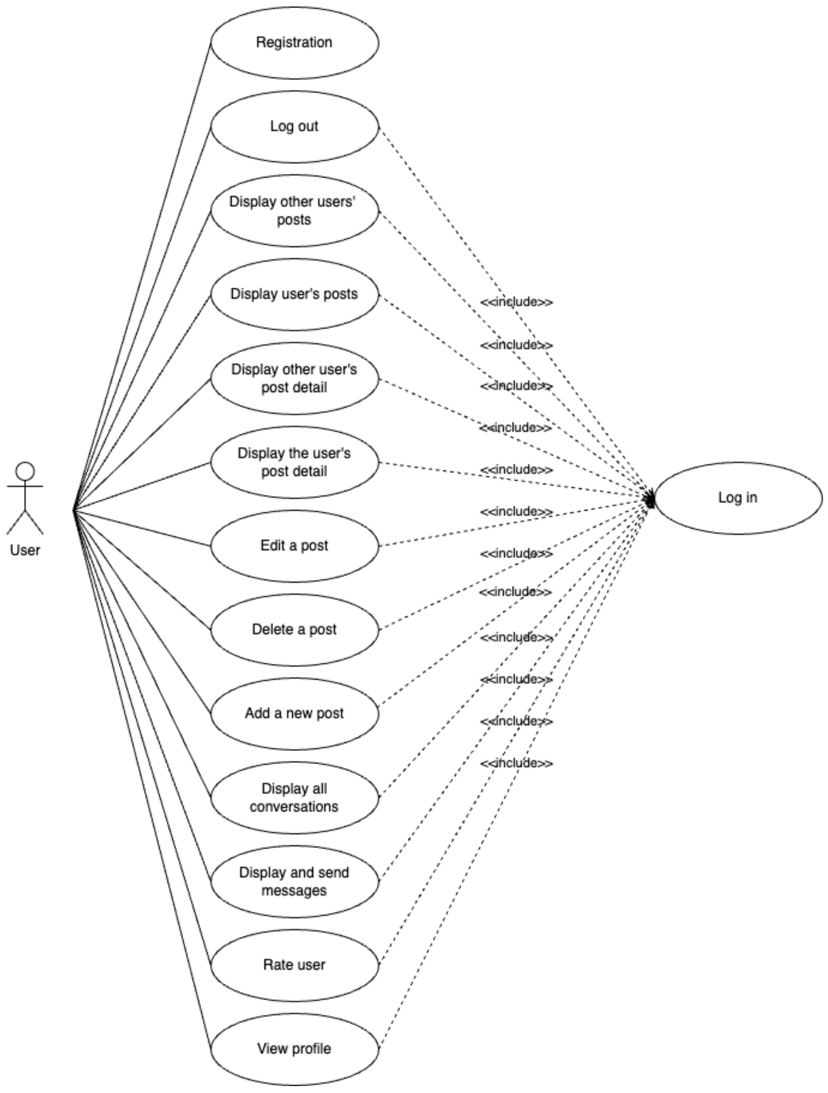

Registration
Log out
Display other users' posts
Display user's posts
Display other user's post detail
Display the user's post detail
Edit a post
Delete a post
Add a new post
Display all conversations
Display and send messages
Rate user
View profile
User
Log in
<<include>>

Figure 2.6: Use case diagram based on [25]

2.5 Domain conceptual model

Before beginning the implementation, it is crucial to develop a domain model, which is constructed based on the requirements. This model helps organise and structure the data, representing the key entities and their relationships within the system. Understanding the problem domain and designing a system that aligns with user needs and expectations is critical. The model for this project is shown below in Figure 2.7. [26]

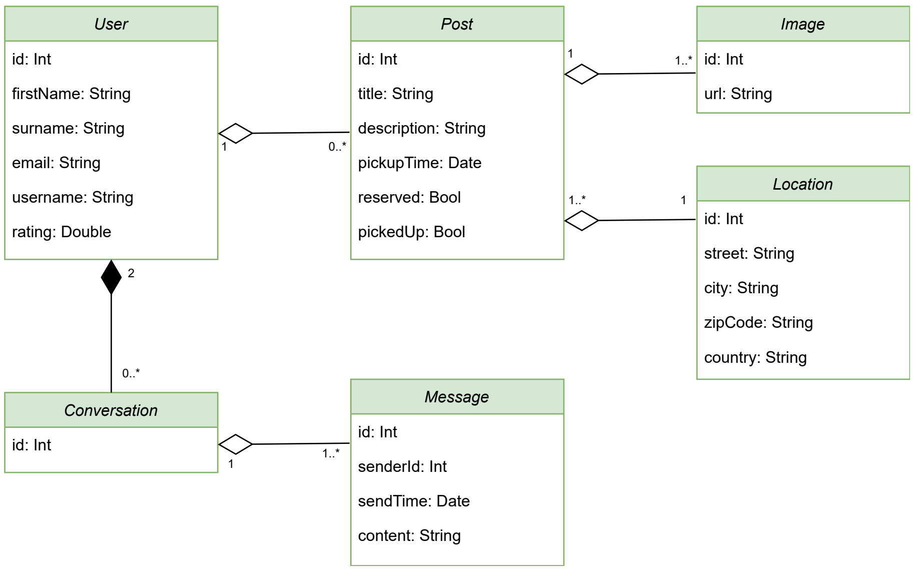


Figure 2.7: Domain model based on [27]

2.5.1 User

The user entity represents an individual user in the system. Each user has unique identifier and is characterized by several attributes: first name, surname, username, email, and rating representing an average of all the ratings received from other users. In the domain model, the relationship between the user and post entities is defined such that a user can have zero to many posts (0-to-N), and each post is associated exclusively with one user. Similarly, a user can engage in zero to many conversations (0-to-N).

2.5.2 Post

The post entity in the domain model represents an item or offering created by a user. Each post with an unique identifier includes several attributes: title, description and pickup time. Additionally, a post can be marked as reserved or picked up by its owner. The relationship between the user and post entities is crucial, as a post cannot exist without being associated with a user in the context of this application. This relationship is modelled as an aggregation, indicating that posts are a component of a user but do not define the user's entire identity. This modelling approach helps maintain the integrity of the user entity while allowing for the dynamic management of posts associated with that user.

2.5.3 Image

Each post must have at least one image, and each can be linked to only one post, forming a 1-to-N relationship. This relationship is modelled as an aggregation because each image relies on its associated post for context and relevance within the system. An image is characterized by a unique identifier and a Uniform Resource Locator (URL), which specifies its exact location where it is stored.

2.5.4 Location

In the context of an application for item giveaways, the location entity represents where the item pickup will occur. It is uniquely identified by an identifier and includes several attributes to specify its precise details: street, city, zip code, and country. This structured approach ensures that each pickup location is clearly defined and easily accessible for users arranging to collect an item. Each location is associated with at least one post and is tied to a specific location. This relationship is modelled as an aggregation because, within the application context, a location does not exist independently without being associated with a post.

2.5.5 Conversation

The conversation entity represents a dialogue between two users within the system. The relationship between the conversation and user entities is modelled as a composition, which underscores the dependency of a conversation on the users involved. This means that a conversation cannot exist independently without associated users. Each conversation must contain at least one message to be considered valid, establishing a 1-to-N relationship.

2.5.6 Message

The message entity represents a single message sent by a user within a conversation. It includes several attributes: sender ID, send time, content and a unique ID. These attributes define the characteristics and ownership of each message. In the context of this application, a message is inherently linked to a conversation, emphasizing that messages cannot exist independently outside of a conversation. The relationship between the message and conversation entities is modeled as an aggregation. This modeling choice signifies that while a message is a part of a conversation, it is distinct and belongs exclusively to that specific conversation.

CHAPTER 3

Design

After conducting a thorough analysis, it is time to move forward with the design of the final application. Armed with a clear understanding of the requirements, we can now discuss the selected platform, programming language, technologies, various architectural options for the project and User Interface (UI) design.

3.1 Android platform

The Android operating system was initially developed by Android Inc. for digital cameras. However, in 2005, Google Inc. acquired Android Inc. and redirected its focus towards developing it as an operating system for mobile phones. Android released the first phone in 2008. By 2012, it had become the most popular operating system globally, surpassing other platforms in usage and adoption. [28]

Android OS is currently developed for various devices, including smartphones, tablets, and smart televisions [29]. New major versions are released annually. The most recent version is Android 14, called "Upside Down Cake". [30]

New phones equipped with the Android operating system are continually being produced and are affordably priced for a wide range of consumers. The consistent updates to the system, along with its broad accessibility, suggest that Android's popularity will likely remain strong in the future. [31]

3.2 Programming language

The programming languages chosen for consideration for the project were Java, Kotlin, C++, C#, and Dart. These were selected because they are among the top five programming languages for Android mobile development, according to [32].

3.2.1 Java

Java is globally considered one of the most popular programming languages. A key benefit of Java is its cross-platform nature, which allows it to operate on any system equipped with a Java Virtual Machine. This capability ensures that Java applications can function on various operating systems without requiring modifications specific to each platform.

Despite improvements from the Android Runtime, Java still introduces some performance overhead. Known for its verbosity, Java often requires developers to write longer code filled with boilerplate, and its syntax lacks the expressiveness of newer programming languages. These factors can lead to longer development and testing periods, potentially slowing down the project delivery. [32]

3.2.2 Kotlin

As previously mentioned, newer programming languages like Kotlin provide enhanced expressiveness. According to [32], Kotlin, which builds on Java, simplifies coding by reducing the required code volume by approximately 40% compared to Java. This reduction is primarily due to Kotlin's more concise syntax and fewer boilerplate requirements, making it an increasingly preferred choice for Android developers. Additionally, according to [33], Kotlin is considered by Google to be the preferred language for Android development, emphasizing its growing importance in the mobile application development field and encouraging developers to adopt Kotlin due to its concise syntax, safety features, and interoperability with Java.

3.2.3 C++

C++ is known for its exceptional speed, making it well-suited for applications demanding high performance. It offers extensive capabilities, including precise control over memory management and manipulation. However, the complex syntax of C++ requires a deep understanding and can increase the risk of errors.

Additionally, C++ is not inherently supported as a development language for Android. Developers looking to utilize C++ in their Android applications must use the Android Native Development Kit (NDK), which enables the integration of C++ code with Java or Kotlin, allowing for performance optimizations in critical areas of the application. [32]

3.2.4 C#

Using C# for Android development is similar to using C++ because C# is also not a native language for Android, necessitating the use of the Android NDK. Developers often choose C# due to their familiarity with it. Moreover, they can utilize the extensive debugging and programming tools available in Visual Studio to accelerate their development process. However, it is important to acknowledge that C# and Visual Studio are not optimized for mobile app development. As a result, their features may not be as appropriate for this purpose as those available in Android Studio's toolset. Additionally, since Android development with C# is less prevalent, a smaller community of developers is available for support and guidance when challenges arise. [32]

3.2.5 Dart

Flutter is a robust cross-platform mobile app development framework that utilises Dart as its programming language [34]. Dart stands out for its speed among cross-platform languages, primarily because it is compiled into native Android code. This compilation enhances performance and enables smooth animations. A standout feature of Dart is the hot reload capability, which allows developers to instantly see the effects of code changes in real-time. [32]

However, one significant drawback of Flutter is the large size of the apps it creates, which could be problematic for users with limited device storage or in situations where apps are downloaded over mobile networks. While Flutter has made considerable strides in enhancing performance, it still may not fully replicate the experience of native development. [34]

3.2.6 The best choice for the project

In conclusion, among the programming languages discussed for Android development, Kotlin distinctly stands out due to its numerous advantages. Google's preference for Kotlin as the primary language for Android underscores its critical role in mobile app development. This endorsement, combined with Kotlin's enhanced safety features and seamless interoperability with Java, makes it an exceedingly attractive option for developers. Unlike C++ and C#, which require additional tools like the NDK for Android development, Kotlin works natively with Android Studio, providing a smoother and more integrated development experience. Given these compelling advantages, Kotlin not only meets the modern demands of Android development but also provides a more efficient and developer-friendly environment compared to its counterparts. This makes Kotlin the superior choice for developers aiming to build high-quality Android applications efficiently.

3.3 Integrated Development Environment

Since the Android development documentation [35] officially recommends Android Studio (AS) as the Integrated Development Environment (IDE), the author opted to use it for this project. *"Based on the powerful code editor and developer tools from IntelliJ IDEA, Android Studio offers even more features that enhance your productivity when building Android apps"* [35].

This IDE provides an environment that enables developers to create applications compatible with all Android devices. AS features a versatile Gradle-based build system, a fast and well-equipped emulator, and provides code templates that are very helpful for starting new projects. These tools streamline the development process, making it more efficient and accessible for developers. [35]

3.4 Architecture

This section will delve into the architecture of the project. According to Android documentation [36], the recommended architecture for mobile applications involves dividing the structure into three main layers as depicted in Figure 3.1: the UI layer, the Domain layer (which is optional), and the Data layer. These recommendations aim to enhance the scalability and testability of applications while improving overall quality.

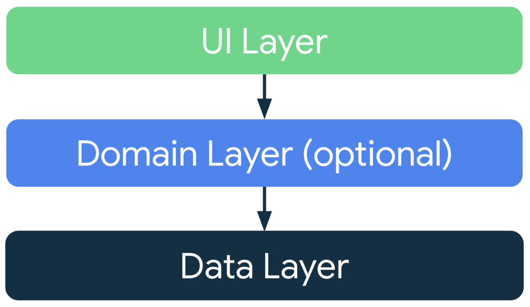


Figure 3.1: Android recommended architecture diagram by [36]

According to [37], the most commonly used architectures for the UI layer are Model-View-ViewModel (MVVM) and Model-View-Intent (MVI). Both architectures are favoured for their effectiveness in separating concerns and simplifying software applications' management and scalability.

3.4.1 MVVM vs MVI

MVVM and MVI architectures share common elements: the "M" for Model and the "V" for View. The Model acts as the data layer, managing business logic and data operations, while the View is dedicated to the UI layer, handling the presentation of data and user interactions. This separation ensures that the data management is independent of the user interface. [38]

3.4.1.1 MVVM

The "VM" in MVVM stands for ViewModel. It maintains the state, representing the current data condition displayed to the user. The ViewModel is responsible for managing changes and fetching data as needed. A single ViewModel can manage multiple states, thereby controlling the data-related logic to keep the UI refreshed with the latest information without directly interacting with the View. For instance, when a user clicks a button, the View communicates this action directly to the ViewModel via a function call. The ViewModel then processes this information, interacts with the Model to make necessary changes, and updates the state accordingly. This arrangement allows the ViewModel to act as a crucial intermediary between the Model and the View, streamlining data flow and enhancing the efficiency of UI updates. [38]

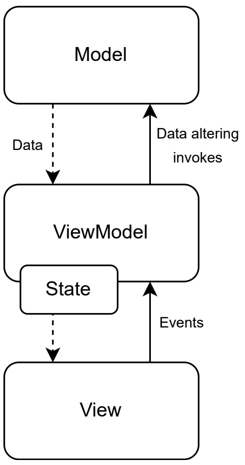


Figure 3.2: MVVM Diagram inspired by [38]

3.4.1.2 MVI

The “I” in MVI stands for Intent, which represents any intent from the view, typically triggered by user interactions. This concept is similar to that in MVVM, with a subtle distinction. In MVI, the view issues an Intent to the ViewModel instead of calling a function directly. For example, when a user taps on a button, the view issues an Intent to inform the ViewModel that it needs to handle this action. The ViewModel then processes this information and takes the necessary actions, similar to the MVVM approach. Essentially, the Intent acts as a mediator between the view and the ViewModel, facilitating communication and coordination. Another key difference is in how the state is managed. In MVVM, individual values each maintain their own separate state, whereas in MVI, all values are consolidated into a single immutable data object. One key advantage of this method is the ease with which the handling of Intents can be modified, enhancing the flexibility and maintainability of the application. [39]

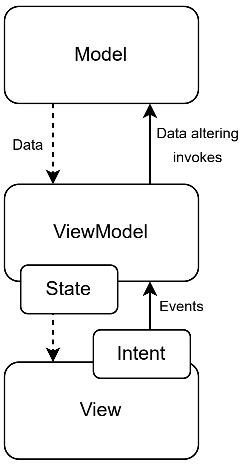


Figure 3.3: MVI Diagram inspired by [39]

3.4.1.3 The best UI layer architecture for the project

According to [40], the MVI adds a new layer of user interactions, resulting in a better separation of concerns recommended by official Android development documentation. In MVI, the application's entire state is encapsulated within a single, immutable data object. This means that at any given point, only one definitive version of the state represents all aspects of the app's UI and data. Because all changes to the application must go through a well-defined cycle of Intents, State updates, and rendering, it becomes much easier to understand how and why a state changes. Due to these advantages and the recommendations from the official Android documentation, this project will use the MVI architecture.

The project's UI layer architecture design is illustrated in the diagram shown in 3.4. Each feature includes its own Model, while each screen is associated with its own View, ViewModel, State, and Intent. This structure will be consistent across all features.

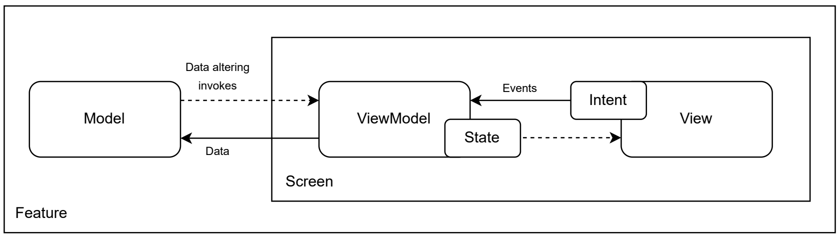


Figure 3.4: UI layer diagram

3.4.2 Clean architecture

While the chosen MVI architecture effectively manages the UI layer, Clean Architecture is designed to segment the entire application into distinct layers as depicted in Figure 3.5. [38]

Clean Architecture advocates for implementing design patterns like dependency injection and inversion of control to streamline and organize code more efficiently. These patterns help decouple the system's components and layers, making the architecture more modular and flexible. This approach facilitates more manageable maintenance and scalability by minimizing dependencies between application parts. [41]

The author chose this approach due to its popularity, extensibility, and adherence to the principles recommended by the official Android documentation. This ensures that the architecture aligns well with established best practices, making it a reliable and forward-compatible choice for Android development. [41]

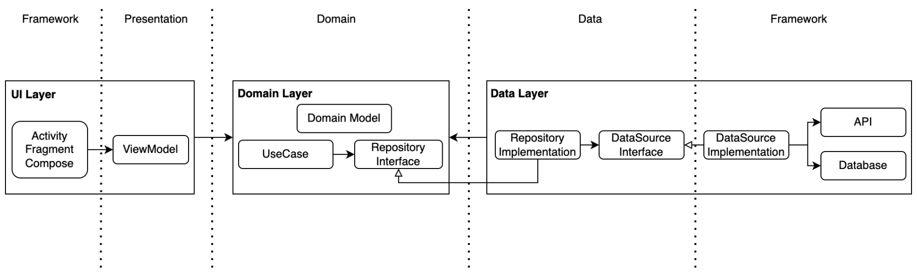


Figure 3.5: Clean architecture diagram by [42]

3.4.2.1 Domain layer

"The domain layer is an optional layer that sits between the UI layer and the data layer" [43]. It consists of domain model, use cases and repository interface.

Domain models are essential components, embodying the real-world concepts, entities, or business logic relevant to the application's domain. [44]

A use case is a component of the business logic layer, each designed to fulfil a single responsibility, representing a distinct action the application needs to perform. Use cases are typically invoked by view models, enhancing reusability when multiple ViewModels require the same functionality. This design not only aids in making the ViewModels cleaner and more modular but also improves the readability and clarity of the operations the ViewModel manages. [42]

The Repository Interface plays a crucial role in the architectural design of an application by defining what data operations are required by the Domain layer without dictating how these operations should be implemented. This abstraction is crucial for several reasons. By keeping the interface and implementation in separate layers, the system ensures that changes in the data management strategy or data source specifics do not impact the business logic in the Domain layer. This approach applies the loose coupling principle recommended by Android documentation. By applying the principle of dependency inversion, it separates the domain from the data layer. Testing becomes more straightforward with use cases depending on interfaces rather than concrete implementations. Since the domain logic depends only on the interface, changing the underlying implementation can be done with minimal impact on the rest of the application. [42]

3.4.2.2 Data layer

"While the UI layer contains UI-related state and UI logic, the data layer contains application data and business logic" [45]. This layer consists of repository implementation and data source interface.

The Repository Implementation component implements the Repository Interface from the Domain Layer. It serves as the main point for the rest of the application to access data [46]. It also provides interactions between different data sources, ensuring that the Domain Layer remains unaware of the origins of the data [47].

Data Source Interfaces provide abstract definitions for data retrieval methods, enabling repositories to function without detailed knowledge of the actual data sources, whether they are local databases, remote servers, or other storage mediums. This arrangement promotes loose coupling and upholds the dependency inversion principle by isolating data handling from the rest of the system components. [42]

Data Source Implementations are the concrete realizations of the Data Source Interfaces that handle direct interactions with the data sources, such as APIs or databases. These implementations execute the specific data operations defined by the interfaces, ensuring that data is fetched, stored, and managed according to the application's requirements. [42]

3.5 Backend

The project has specific needs, including user authentication, storage for post images, and storage for other data like user profiles, post details, and messages. This section will explore and decide on an appropriate backend solution to meet these requirements.

According to [48], a serverless architecture offers enhanced security compared to traditional server-based setups. This approach allows developers to concentrate more on application development rather than on managing infrastructure. It also significantly aids in managing application data, streamlining operations and reducing overhead. This is why a serverless architecture has been selected for this project, and Firebase and Amazon Web Services (AWS), two of the most popular serverless options, will be discussed [49].

3.5.1 Firebase

Firebase, developed by Google, is designed to accelerate and simplify app development. It provides various services, including authentication, file storage, analytics, Crashlytics, and many more. Regarding security, Firebase prioritises ease of use over complex configurations, which can be advantageous for projects requiring rapid development and deployment. This approach helps developers focus more on building features than managing security details, making Firebase an ideal choice for smaller projects prioritising rapid deployment and ease of management. [50]

3.5.2 AWS

AWS, developed by Amazon, is an excellent choice for developers who require extensive customization options. It offers many services, including file storage, DNS management, data security, and more. While AWS provides robust authentication services that support numerous identity providers and built-in tools for specialized needs like game development or media processing, it can be more complex to learn. This complexity may extend the development time for smaller, simpler applications. Moreover, AWS is renowned for its comprehensive security services, but mastering these can also present a learning curve. Given these characteristics, AWS is particularly well-suited for applications that demand a complex infrastructure and can benefit from its scalable and extensive service offerings. [50]

3.5.3 The best choice for the project

As previously discussed, this project requires functionalities such as user authentication, storage for images, and data management. Both AWS and Firebase offer these services but with distinct characteristics. With its capability to handle complex, highly customizable projects, AWS might be more than needed for this project's scope. Firebase, however, is ideal for more straightforward, smaller-scale projects and provides all the necessary features without the complexity of AWS. Thus, for its simplicity and direct support for the project's requirements, Firebase is the better choice for this project.

3.6 Frontend

Historically, Android applications have relied on Extensible Markup Language (XML) Views to shape their visual and functional aspects [51]. Jetpack Compose, a new approach to building Android applications, saw its first stable release in 2021 [52]. This section explores the differences and advantages of both XML and Jetpack Compose in the context of Android UI development.

3.6.1 XML and Fragments

XML facilitates the creation of UIs in a hierarchical structure and offers a wide range of components for reusable layouts. This separation of the UI from the business logic simplifies the process for designers, allowing them to focus on the aesthetic aspects without interfering with the application's core functionality. [53]

XML is widely utilized, simplifying the learning process and enabling easier access to community support. The example of XML code is shown in Listing 1. AS includes an editor for XML layouts, which streamlines development and assists in creating visually appealing interfaces. Additionally, a wealth of libraries and frameworks are available to aid development. [53]

The layout editor is a helpful tool for crafting visually appealing user interfaces. Instead of manually coding in XML, developers can simply drag and drop components into the desired positions, streamlining the UI design process. [54]

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

Listing 1: Illustration of XML code from [55]

To display the UI created in XML, it must be embedded in a fragment, which is then placed into an activity. For simpler UI designs, using only an activity may suffice. In both scenarios, the XML code becomes an integral part of them. [56]

You can think of an activity as a single screen with its visuals and fragments as modular sections of that activity. An activity is typically used for navigating between screens or managing different components of the UI, while fragments are designed to break down complex UIs into smaller, more manageable parts. [56]

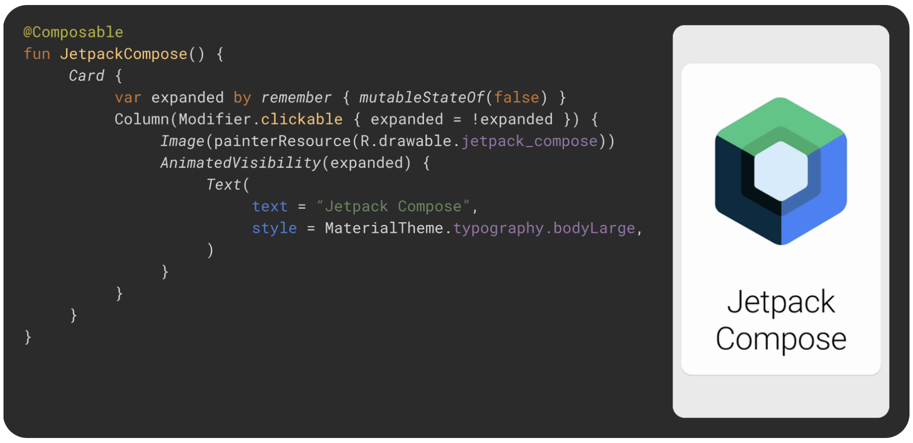
There is considerable criticism directed at fragments and activities due to their complexity and the challenges posed by managing two separate lifecycles. Many developers encounter issues such as memory leaks, handling configuration changes, and difficulties with components that are poorly documented. These challenges often complicate the development process and can lead to less stable applications. [57]

3.6.2 Jetpack Compose

Jetpack Compose brings a new era in Android development by eliminating the need for fragments in a composable application. Compose provides developers with all the necessary tools for development directly, streamlining the coding process. This approach reduces the amount of code and enhances stability by avoiding the complexities and issues traditionally associated with managing fragments. [57]

"Jetpack Compose is Android's recommended modern toolkit for building native UI. It simplifies and accelerates UI development on Android. Quickly bring your app to life with less code, powerful tools, and intuitive Kotlin APIs" [58].

All code in Jetpack Compose is written in Kotlin, simplifying the development process by significantly reducing the amount of code needed. This makes the coding more direct and intuitive. Components in Jetpack Compose are fully reusable and independent of any activity or fragment. AS enhances Jetpack Compose with capabilities such as live previews, enabling developers to view updates in real time as they modify the code. This functionality dramatically accelerates the development workflow by removing the necessity to rebuild the app for each minor adjustment constantly. Jetpack Compose incorporates aMaterial Design implementation that is fully open to developers. Material Design presents a thorough design framework that establishes key guidelines and best practices for developing visually cohesive and functional user interfaces for a variety of applications. This transparency allows developers to examine how standard components are built, making it easier to create custom components by understanding and modifying the existing implementations. Moreover, Jetpack Compose offers robust animation capabilities, which significantly enhance the visual appeal of the user interface, adding to the overall user experience. The illustration of Jetpack Compose code is shown in Figure 3.6. [59]



```
@Composable
fun JetpackCompose() {
    Card {
        var expanded by remember { mutableStateOf(false) }
        Column(Modifier.clickable { expanded = !expanded }) {
            Image(painterResource(R.drawable.jetpack_compose))
            AnimatedVisibility(expanded) {
                Text(
                    text = “Jetpack Compose”,
                    style = MaterialTheme.typography.bodyLarge,
                )
            }
        }
    }
}
```

Figure 3.6: Illustration of Jetpack Compose code from [58]

3.6.3 The best choice for the project

The benefits of Jetpack Compose clearly justify its selection for development projects. With its promise of less code, fewer complications, and simplified coding processes, it stands out as an advantageous choice in app development. In this project, Jetpack Compose will be used for creating an appealing user interface that aligns with the recommendations of the Android development documentation and modern development practices.

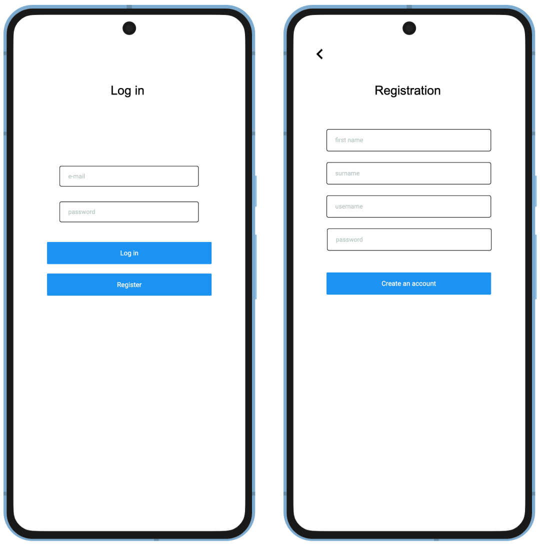
3.7 User Interface Design

As previously noted, it is crucial for the application to be visually appealing and easily navigable. Achieving this requires careful and thoughtful design, ensuring the interface is aesthetically pleasing and user-friendly.

While managing the various aspects of thinking and planning for an app can be challenging, mobile app wireframing plays a crucial role in turning the vision into reality. Essentially, a mobile app wireframe is demonstrating how the app will function. Wireframe is not a complete representation of the app's final design but instead focuses on the layout of key screens and interface elements. This helps in organizing the app's structure and user flow before moving into more detailed design phases. [60]

3.7.1 Authentication

When a user launches the application and is not logged in, the Login screen appears, as shown in Figure 3.7a. Users have two options: If they have an account, they can enter their email and password to log in, if they do not have an account, they can click on the register button, which redirects them to the registration screen, that is illustrated in Figure 3.7b.



(a) Log in (b) Registration

Figure 3.7: Illustration of authentication wireframes made in [61]

3.7.2 Bottom navigation

If the user is logged in, the application displays a bottom navigation bar divided into four sections: Home, My Posts, Messages, and Settings as displayed in Figure 3.8. This navigation bar always remains visible while the user is logged in. Since a user does not have access to any posts or other app features before logging in, it makes sense that they cannot navigate within the app during the authentication process.

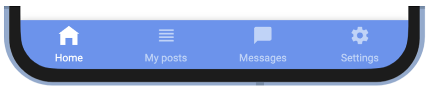
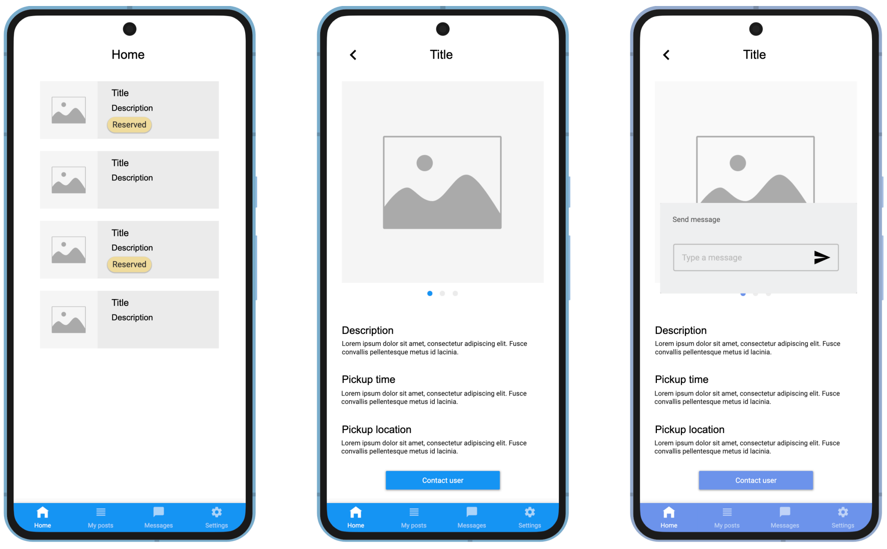


Figure 3.8: Bottom navigation

3.7.3 Other users posts

The Home screen appears after successfully logging into the application, as depicted in Figure 3.9a. On this screen, a user can view posts from other users. Each post has a preview that includes one image, a title, and a short description. If an item from a post is reserved, this status is indicated on the post. Clicking on a post preview opens the post detail screen, shown in Figure 3.9b.

On the post detail screen, a user can view the post's complete details, including all images, the full description, and information on the pickup time and location. If a user is interested in the item, they can click the dedicated button and send a message to the person offering it. After pressing this button a dialog appears where user can write the message as shown in Figure 3.9c.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce convallis pellentesque metus id lacinia.

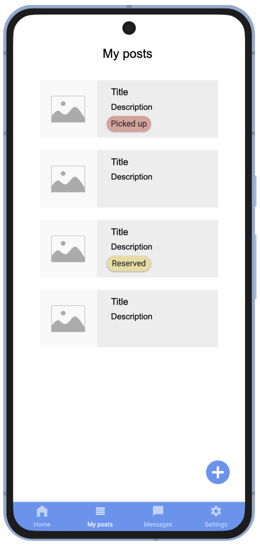
(a) Home (b) Post detail (c) Send message dialog

Figure 3.9: Illustration of wireframes related to other users' posts made in [61]

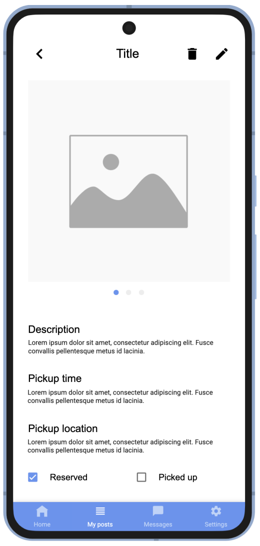
3.7.4 User's posts

The My Posts screen, displayed in Figure 3.10a, is specifically dedicated to displaying the posts made by the user. Unlike the Home screen, which showcases posts from other users, this screen exclusively presents the user's own posts. Additionally, posts on this screen can be marked not only as reserved but also as picked up. This screen also provides the user with the functionality to create a new post via a dedicated button located at the bottom of the screen.

Similar to the Home screen, clicking on any post on the My Posts screen will open the post detail screen, that is shown in Figure 3.10b. In addition to displaying the images, description, and pickup location and time, this screen also gives the user the option to delete or edit the post. Moreover, users can mark the post as reserved or picked up, providing full control over managing their listings.



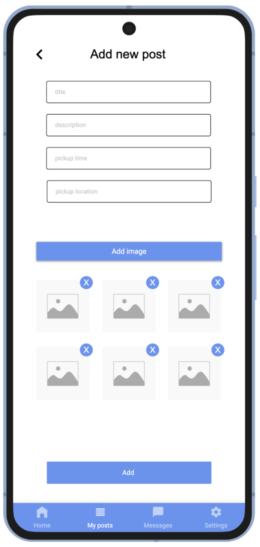
(a) My posts


Title
Description
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce convallis pellentesque metus id lacinia.
Pickup time
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce convallis pellentesque metus id lacinia.
Pickup location
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce convallis pellentesque metus id lacinia.
Reserved
Picked up
Home
My posts
Messages
Settings

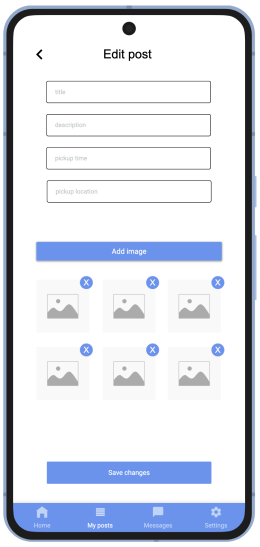
(b) My post detail

Figure 3.10: Illustration of wireframes related to the user's posts made in [61]

To access the screen for adding a new post, which is shown in Figure 3.11a, users can navigate through the button located in the post detail screen, as previously mentioned. Users are required to input all relevant information about the item they wish to give away. Completing the form also involves uploading images by tapping on the dedicated button for image uploads. This step is crucial to ensure that all necessary details are included, making the post informative and appealing to potential recipients.



(a) Add new post



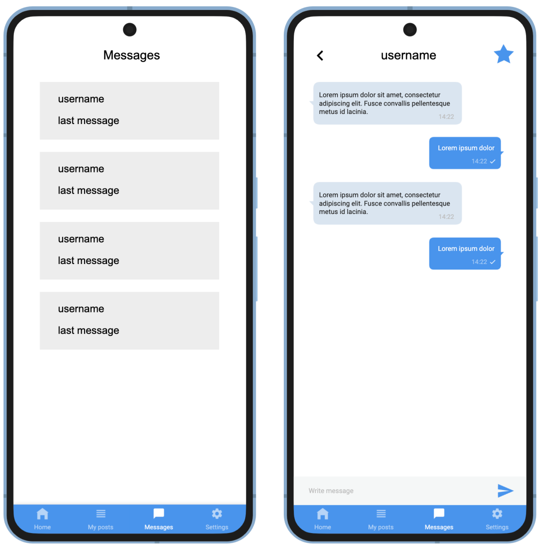
(b) Edit post

Figure 3.11: Illustration of post managment wireframes made in [61]

3.7.5 Messages

When a user accesses the Messages section from the bottom navigation, a screen displaying a list of all the user's conversations appears. Each conversation entry shows the username and the last message sent, as illustrated in Figure 3.12a.

Clicking on a conversation reveals its detailed view, depicted in Figure 3.12b. This screen displays all messages exchanged between the two users in chronological order. At the bottom of the screen, there is a text field where the user can type and send new messages. At the top of the screen, the other user's username is displayed alongside a rating button. This button allows the user to rate the other user based on responsiveness and safety.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce convallis pellentesque metus id lacinia. 14:22

Lorem ipsum dolor 14:22 ✓

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce convallis pellentesque metus id lacinia. 14:22

Lorem ipsum dolor 14:22 ✓

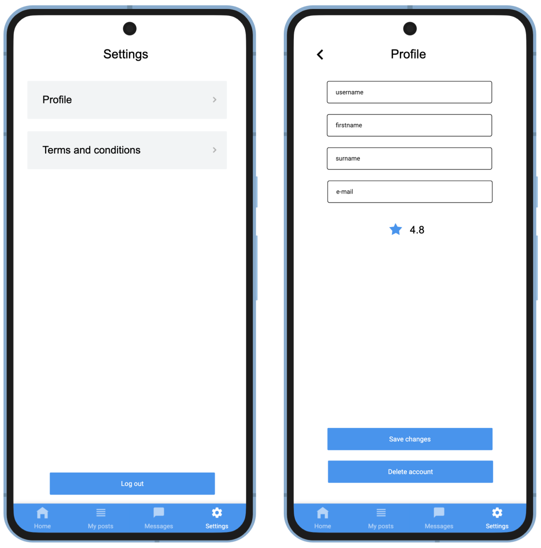
(a) All conversations (b) Conversation detail

Figure 3.12: Illustration of conversations wireframes made in [61]

3.7.6 Settings and profile

The Settings screen, which is shown in Figure 3.13a, includes a section dedicated to the user's account management. At the bottom of this screen, there is a button for logging out. The Settings also provide access to the user's profile and the terms and conditions.

The Profile screen, shown in Figure 3.13b, allows the user straightforward access to modify any personal information as necessary and save the changes using a button located at the bottom of the screen. Additionally, users can check their ratings on this screen. Besides updating and checking profile information, users also have the option to delete their account using a dedicated button.



(a) Settings (b) Profile

Figure 3.13: Illustration of settings and profile wireframes made in [61]

Chapter 4

Implementation

4.1 Kotlin Multiplatform

Kotlin Multiplatform (KMP) [62] is a technology developed by JetBrains that allows developers to use Kotlin to create multiplatform applications. With KMP, developers can write their business logic once in Kotlin and then share it across both iOS and Android platforms while still allowing for platform-specific implementations where necessary. This approach reduces the time and effort required to develop and maintain both apps by sharing common code across platforms. This helps to ensure consistent behavior and logic throughout the application.

In this project, the domain and data layer were implemented using KMP. This was accomplished by creating a shared module that contains those layers. This setup enhances the project's extensibility, meaning that adding an iOS version of the app would primarily involve developing the UI layer in another module. The core logic and data management are already in place in the shared module, streamlining the expansion to other platforms.

4.2 Modularization

Modularization [63] refers to structuring a codebase into distinct modules, each designed to perform a specific role or function. Breaking down a large application into smaller parts makes it easier to manage. Modules designed to perform specific functions can be reused across different parts of an application. New functionalities can be added as new modules without extensive modifications to existing code.

One of the key advantages of modularization is the reduction in build times. When changes are made to a particular module, only that module needs to be recompiled, which can significantly save time, especially in larger applications.

Additionally, modularization helps prevent circular dependencies between components. Making sure that modules interact through clear, one-way dependencies avoids the complexities and challenges of managing intertwined relationships between modules.

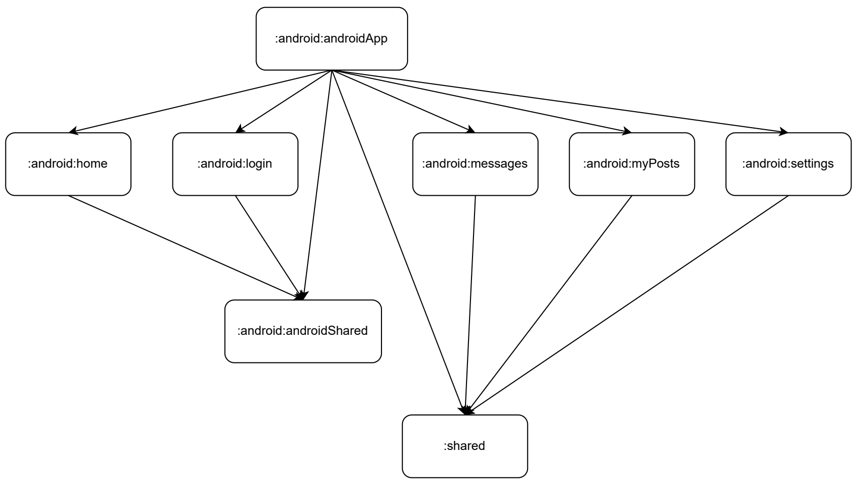


Figure 4.1: App division into modules

This modular approach, illustrated in Figure 4.1, effectively divides the application into distinct logical units, each handling a specific app function. Such segmentation improves the ease of maintenance, allows the app to scale efficiently, and may decrease build times since modules can be compiled independently.

Additionally, by isolating functionalities into distinct modules, dependencies are reduced, and application components are decoupled. This is crucial for avoiding circular dependencies and simplifying the app's testing and debugging.

4.3 Gradle

Gradle [64] is a powerful and versatile build tool used primarily for Java or Kotlin projects. It automates the process of compiling, packaging, testing, and deploying applications. AS uses Gradle, an official build tool for Android projects.

As discussed in Section 4.2, the project is organized into modules, each equipped with its own *build.gradle* file. The main module, referred to as the application module, is labeled as *:android:androidApp* in the Figure 4.1 and utilizes a different *build.gradle* file compared to the other modules categorized under *:android*. Given that each *build.gradle* file specifies dependencies unique to its module, it is efficient to have a system that manages common dependencies and versions across all modules. To address this, convention plugin has been implemented. This plugin, stored in the build-logic module, simplifies the setup by centralizing dependency management, reducing duplication and making it easier to update dependencies across all modules. The library convention plugin is demonstrated in the Listing 2.

```
class LibraryConventionPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        with(project) {
            project.pluginManager.apply("com.android.library")

            extensions.configure<LibraryExtension> {
                compileSdk = libs.findVersion(CompileSdk).toIntVersion()
                defaultConfig {
                    minSdk = libs.findVersion(MinSdk).toIntVersion()
                }
                compileOptions {
                    sourceCompatibility = JavaVersion.VERSION_11
                    targetCompatibility = JavaVersion.VERSION_11
                }
                tasks.withType<KotlinCompile> {
                    kotlinOptions {
                        jvmTarget = libs.findVersion(JvmTarget)
                    }
                }

                ...

                configureLibraryDependencies()
            }
        }
    }
}
```

Listing 2: Illustration of Library Convention Plugin

The application has a *settings.gradle* file, which is crucial during the initialization phase as it configures the application's properties. This file is essential for adding the required plugin and for including all the modules within the project, as detailed in Listing 3.

```
pluginManagement {
    includeBuild("build-logic")
    repositories {
        google()
        gradlePluginPortal()
        mavenCentral()
        maven("https://plugins.gradle.org/m2/")
    }
}
...
include(":android:androidApp")
include(":shared")
include(":android:home")
include(":android:messages")
...
```

Listing 3: Illustration of Gradle build file

4.4 Firebase

To integrate Firebase [65] into the application, as shown in the [66], the following steps are necessary:

- Create a new project in the Firebase console.
- Register the application using the application ID found in the *build.gradle* file of the application module.
- Download and add the configuration file to the project's application module.
- Install the Firebase SDK by adding it to the *build.gradle* of the application module.
- Grant internet permissions by adding them to the *AndroidManifest* file in the application module, as depicted in Listing 4.
- Initialize Firebase in the *MainActivity* of the application module as shown in Listing 5.
- Disable the Google Services plugin in the root *build.gradle* file as suggested in the [67].
- Add the serializable library to the KMP shared module's *build.gradle* file, along with the necessary Firebase libraries for KMM [68].

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:dist="http://schemas.android.com/apk/distribution">

    ...

    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>

    ...

</manifest>
```

Listing 4: Illustration of internet permissions in Manifest

```kotlin
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        // Initialization code
        Firebase.initialize(this)
        setContent { ... }
    }
}
```

Listing 5: Illustration of Firebase initialization

After this setup the Firebase services are fully available to use in the project and are shown more in Section 4.6.3.

4.5 Dependency Injection

Dependency injection (DI) is highly recommended by android documentation. *"Implementing dependency injection provides you with the following advantages: reusability of code, ease of refactoring, ease of testing"* [69]. Two well-known DI frameworks for Android are Koin [70] and Hilt [71]. For this project, Koin was selected due to its simplicity in setup and its non-reliance on annotations, which makes it exceptionally compact.

To setup Koin few steps are required:

- Add Koin dependencies into the KMP shared module's *build.gradle* file and into the library convention plugin.
- Configure Koin modules in each module, as depicted in Listing 6.
- Load Koin modules by initializing Koin in *MainActivity* with the function showed in Listing 7.

```
val homeModule = module {
    viewModelOf(::HomeViewModel)
    viewModelOf(::SendMessageViewModel)
}
```

Listing 6: Illustration of Koin module

```
fun ComponentActivity.initializeDependencyInjection() {
    initKoin {
        val contextModule = module {
            factory<ComponentActivity> {
                this@initializeDependencyInjection
            }
            factory<Context> { this@initializeDependencyInjection }
        }

        modules(
            contextModule,
            homeModule,
            ...
        )
    }
}
```

Listing 7: Illustration of Koin initialization function

After completing the setup, the DI is ready and can be used as demonstrated in Listing 8. In this example, the `HomeViewModel` class is initialized with a `GetPostsUseCase` dependency. This use case is injected into the `HomeViewModel` via Koin, illustrating how DI facilitates the management of dependencies.

```
internal class HomeViewModel(
    private val getPosts: GetPostsUseCase
) { ... }
```

Listing 8: Illustration of using DI

4.6 Android application

This section covers the implementation of the three main layers of the application. First, the UI layer is explored, including details about UI components, screen navigation, and how ViewModels manage data. Then, the domain layer is explored, focusing on use cases and repositories. Finally, the data layer is described, highlighting data sources and API calls.

4.6.1 UI layer

This section covers the Views, ViewModels, and navigation within the application. As decided in Section 3.4.1.3, the UI layer implements the MVI architecture.

4.6.1.1 Navigation

All the module-specific navigation graphs are called in the `Root` component, illustrated in Listing 9. These navigation graphs contain the navigation logic for each module, organizing how the app moves between different screens within a module. In the beginning, the application checks through a use case if the user is logged in to determine whether the starting destination should be the `LoginScreen` or the `HomeScreen`.

```
@Composable
fun Root() {
    ...
    val startDestination = remember(isUserLoggedIn) {
        if (isUserLoggedIn) Destination.Home.HomeGraph.route
        else Destination.Login.LoginGraph.route
    }
    Scaffold(
        bottomBar = { BottomBar(navController) },
    ) { padding ->
        Box(modifier = Modifier.padding(padding)) {
            NavHost(navController, startDestination = startDestination) {
                loginNavGraph(
                    navHostController = navController,
                    onNavigateToApp = {
                        navController.navigate(
                            Destination.Home.HomeGraph.route
                        )
                    }
                )
                homeNavGraph(navController)
                messagesNavGraph(navController)
                myPostsNavGraph(navController)
                settingsNavGraph(navController)
            }
        }
    }
}
```

Listing 9: Illustration of Root component

As previously discussed, each module is equipped with its own navigation graph, as illustrated in Listing 10. This graph outlines the routes connecting to the module's various screens.

```
fun NavGraphBuilder.homeNavGraph(navHostController: NavHostController) {
    ...
    navigation(
        startDestination = Destination.Home.HomeOverview.route,
        route = Destination.Home.HomeGraph.route,
    ) {
        homeRoute(
            onPostClick = { postId ->
                navHostController.navigate(
                    Destination.Home.Post.PostDetail(postId).route
                )
            }
        )
        homePostDetailRoute( ... )
        ...
    }
}
```

Listing 10: Illustration of navigation graph

Each route is defined by a specific destination, as illustrated in Listing 11. These destinations are implemented as a `sealed class` and include details about the route and any necessary arguments.

```
sealed class Destination(
    val route: String,
    val arguments: List<NamedNavArgument> = emptyList()
) {
    data object Home {
        data object HomeGraph : Destination("home")
        data object HomeOverview : Destination("home/overview")
        data object Post {
            data class PostDetail(val postId: String = "") : Destination(
                route = "home/post/postDetail/${postId}",
                arguments = listOf(
                    navArgument("postId") { type = NavType.StringType },
                )
            ) {
                val routeNoArgs = "home/post/postDetail/{postId}"
            }
            ...
        }
    }
    ...
}
```

Listing 11: Illustration of Destination class

The `homeRoute` function shown in Listing 12, along with the `HomeRoot` and `HomeScreen`, are all defined in a single file. The `homeRoute` function constructs a composable route for a particular screen and is declared as `internal` to limit its visibility to within the module. Each screen's `Root` composable is marked `private`, ensuring it is only accessible within its own file. This configuration also applies to `HomeScreen`. This setup prevents the navigation graph, which manages how screens connect, from modifying the individual screens' implementation.

```
internal fun NavGraphBuilder.homeRoute(onPostClick: (String) -> Unit) {
    return composable(route = Destination.Home.HomeOverview.route) {
        HomeRoot(onPostClick = onPostClick)
    }
}

@Composable
private fun HomeRoot(
    onPostClick: (String) -> Unit,
    ...
) {
    ...
    HomeScreen( ... )
}
```

Listing 12: Illustration of Route function

This navigation pattern is consistently implemented across all the modules, providing a well-organized structure that simplifies the process of adding new screens or modifying navigation details as needed.

4.6.1.2 View

As previously noted, the project utilizes Jetpack Compose to develop the application's UI. This subsection demonstrates the implementation of the screens using Jetpack Compose.

As mentioned in Section 4.6.1.1, in the application's architecture, each screen's root composable function is marked `private` to restrict its accessibility to within its own file. Within these root composables, necessary dependencies and resources, such as ViewModels and state handlers, are injected. This function handles the communication between the View and Intents or handles Actions coming from ViewModel, all explained in Section 4.6.1.3.

```
@Composable
private fun HomeRoot(
    onPostClick: (String) -> Unit,
    viewModel: HomeViewModel = koinViewModel(),
    errorSnackBarHostState: SnackbarHostState = remember {
        SnackbarHostState()
    }
) {

    ...

    val state by viewModel.state.collectAsState(
        HomeViewModel.ViewState()
    )
    HomeScreen(
        state = state,
        errorSnackBarHostState = errorSnackBarHostState,
        onIntent = viewModel::onIntent
    )
}
```

Listing 13: Illustration of root composable

The screen composable in the application is also marked as `private` to limit its access to just its own file, enhancing encapsulation. This clear division between `UI` and business logic means changes to the `UI` won't impact the underlying business processes.

```
@Composable
private fun HomeScreen(
    state: HomeViewModel.ViewState,
    errorSnackBarHostState: SnackbarHostState,
    onIntent: (ViewIntent) -> Unit,
) {
    BaseScreen(
        title = "Home",
        isLoading = state.loading,
        errorSnackBarHostState = errorSnackBarHostState,
    ) {
        PostsList(
            posts = state.posts,
            onPostClick = { id ->
                onIntent(ViewIntent.OnPostClicked(id))
            },
        )
    }
}
```

Listing 14: Illustration of screen composable

The `PostItem` component is shown in Figure 4.2. Constructing these components is straightforward and intuitive.

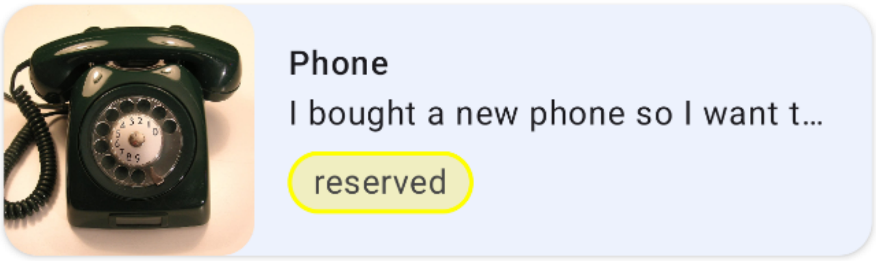


Figure 4.2: Illustration of PostItem component

Jetpack Compose makes it easy to create attractive components with its variety of built-in tools like `Rows`, `Columns`, `Cards`, `Texts`, `Images` and much more. This allows developers to design appealing user interfaces efficiently.

The `PostItem` from `HomeScreen` or `MyPostsScreen` is made out of built-in `ElevatedCard`, inside of it is a `Row` and inside of it is an `Image`, `Column` with `Text` and a custom component `ReservedCard` as shown in Listing 15.

```
@Composable
fun PostItem(
    post: Post,
    ...
) {
    ElevatedCard( ... ) {
        Row {
            ...
            Image( ... )
            Column( ... ) {
                Text( ... )
                ...
                Row( ... ) {
                    if (post.reserved && !post.pickedUp) {
                        ReservedCard()
                    }
                     ...
                }
            }
        }
    }
}
```

Listing 15: Illustration of PostItem component code

All the screens of the application are shown in Appendix B.

4.6.1.3 ViewModel

Every ViewModel in the application is structured to include a State, Intent, and Action, as illustrated by the `BaseViewModel` shown below in Listing 16. Each ViewModel in the application extends this `BaseViewModel` class, incorporating its State, Intent, and Action elements.

Each ViewModel includes a State, which encapsulates the current UI state. The update function is utilized to modify this state in response to UI events, such as user input in a text field. This ensures that the UI accurately reflects the latest user interactions and system changes.

The Intent serves as a directive from the UI, signaling the ViewModel to perform certain operations. The `launchOnIO` function is designed to handle tasks that involve network communication or other I/O operations. These tasks are executed on the `Dispatchers.IO` coroutine dispatcher, backed by a pool of threads optimized for such operations. This setup allows for efficient management of I/O tasks without blocking the UI, which runs on the main thread.

The Action component is responsible for instructing the UI to execute specific actions based on the decisions made by the ViewModel. For example, if a user presses a button to navigate back, the ViewModel captures the Intent, which then processes and determines the appropriate course of action. The `launchViewAction` function emits these actions, which the UI handles. This function operates on the main thread (using `Dispatchers.Main`), ensuring that interactions with the UI are smooth and responsive.

```kotlin
interface State
interface Intent
interface Action

abstract class BaseViewModel<S : State, A : Action, I : Intent>(
    initialState: S,
) : ViewModel() {

    private val stateFlow = MutableStateFlow(initialState)
    val state: Flow<S> = stateFlow

    private val _viewAction = MutableSharedFlow<A>()
    val viewAction = _viewAction.asSharedFlow()

    protected fun update(body: S.() -> S) {
        stateFlow.value = body(stateFlow.value)
    }

    protected fun launchViewAction(viewAction: A) {
        viewModelScope.launch(Dispatchers.Main) {
            _viewAction.emit(viewAction)
        }
    }

    protected fun launchOnIO(
        block: suspend CoroutineScope.() -> Unit,
    ) = viewModelScope.launch(Dispatchers.IO) { block() }

    abstract fun onIntent(intent: I)

    val currentState: S get() = stateFlow.value
}
```

Listing 16: Illustration of BaseViewModel

For example, consider the `HomeViewModel` shown in Listing 17. When the screen is initialized, it triggers the `OnViewInitialized` intent, which prompts the ViewModel to load the necessary data. Additionally, if a user taps on a post, the `OnPostClicked` intent is activated, leading to an action that navigates the user to the post's detailed view.

The user interface is designed to only display information and relay user interactions back to the ViewModel through Intents. It remains completely decoupled from the business logic, focusing solely on rendering the UI based on the state provided by the ViewModel. Intents, such as `OnViewInitialized` and `OnPostClicked`, serve as the primary means of communication between the View and the ViewModel. They describe the user's intended actions, enabling the ViewModel to handle necessary state changes and actions accordingly.

This clear division of responsibilities supports the principle of separation of concerns, ensuring that the UI layer focuses exclusively on user interactions and visual presentation, while the ViewModel efficiently handles logic and state management.

```kotlin
internal class HomeViewModel(
    private val getCurrentUserId: GetCurrentUserIdUseCase,
    private val getOtherUsersPosts: GetOtherUsersPostsUseCase
) : BaseViewModel<ViewState, ViewAction, ViewIntent>(ViewState()) {
    override fun onIntent(intent: ViewIntent) {
        when (intent) {
            is ViewIntent.OnViewInitialized -> onViewInitialized()
            is ViewIntent.OnPostClicked -> onPostClicked(intent.id)
        }
    }
    private fun onViewInitialized() { loadData() }
    private fun loadData() {
        launchOnIO {
            update { copy(loading = true) }
            ....
            update { copy(loading = false) }
        }
    }
    private fun onPostClicked(id: String) {
        launchViewAction(ViewAction.PostClicked(id))
    }
    sealed interface ViewIntent : Intent {
        data object OnViewInitialized : ViewIntent
        data class OnPostClicked(val id: String) : ViewIntent
    }
    sealed interface ViewAction : Action {
        data class PostClicked(val id: String) : ViewAction
        data class ShowError(val error: String) : ViewAction
    }

    data class ViewState(
        val loading: Boolean = false,
        ...
    ) : State
}
```

Listing 17: Illustration of HomeViewModel code

4.6.1.4 Model

In the context of MVI and many modern app architecture patterns, the term “Model” can sometimes be confusing because it does not strictly refer to the data model in the traditional sense. Instead, it often encompasses the overall domain logic, including state management and the business logic affecting this state. The ViewState is part of the Model, but the Model also includes the Use-Cases that encapsulate the business logic, but those are shown in the next section. While the ViewState is technically managed within the ViewModel, it represents the state part of the Model in MVI architecture terms. It holds the data needed by the UI at any given moment—such as whether data is being loaded or an error message if something goes wrong.

4.6.2 Domain layer

This layer contains Domain Models, UseCases and Repositories. Each of them will be shown in an example and is implemented for everything needed for this application to function.

4.6.2.1 Domain Model

As previously noted, domain models are crucial elements that encapsulate real-world concepts or entities. Each model is implemented as a `data class` containing the necessary values. For instance, the `Login` domain model is illustrated in Listing 18.

```
data class Login(
    val email: String,
    val password: String,
)
```

Listing 18: Illustration of Domain Model

4.6.2.2 UseCase

In the Section 3.4.2, UseCases were discussed as a part of clean architecture. They help ViewModels interact with relevant data. Each UseCase handles a specific action that the app needs to perform.

Each UseCase extends a base UseCase interface and is categorized into one of four types based on its function: `UseCaseResult`, `UseCaseResultNoParams`, `UseCaseFlowResult` and `UseCaseLocal`. All types include an `invoke` operator that executes the UseCase.

The `UseCaseResult` requires a parameter. For instance, as shown in Listing 23, this UseCase accepts all necessary registration details as a parameter. The outcome of executing a `UseCaseResult` is encapsulated in a `Result`, which indicates whether the operation was successful or provides an error with a message if it failed. The structure of this type of UseCase is detailed in Listing 19.

```
interface UseCaseResult<in Params, out T : Any> {
    suspend operator fun invoke(params: Params): Result<T>
}
```

Listing 19: Illustration of UseCaseResult

The key difference between `UseCaseResult` and `UseCaseResultNoParams` is that one requires parameters, while the other does not. The implementation of the `UseCaseResultNoParams` interface is detailed in Listing 20.

```kotlin
interface UseCaseResultNoParams<out T : Any> {
    suspend operator fun invoke(): Result<T>
}
```

Listing 20: Illustration of UseCaseResultNoParams

The third type, `UseCaseLocal`, is designed for local operations, such as communicating events between screens. The code of this type is shown in Listing 21. This UseCase might be used to notify a screen about an event on another screen, like informing a user that a post has been successfully deleted after they are redirected back to the list of their posts. `UseCaseLocal` does not return a `Result` since its main function is to relay information to the user, not to handle data operations that require success or error states.

```kotlin
interface UseCaseLocal<out T : Any> {
    suspend operator fun invoke(): T
}
```

Listing 21: Illustration of UseCaseLocal

The `UseCaseFlowResult`, whose structure is shown in Listing 22, is used for retrieving data with Flow - an asynchronous data stream. It is particularly useful for continuously receiving data, such as messages from other users. This use case needs to be collected in the ViewModel since it operates as a Flow.

```kotlin
interface UseCaseFlowResult<in Params, out T : Any> {
    suspend operator fun invoke(params: Params): Flow<Result<T>>
}
```

Listing 22: Illustration of UseCaseFlowResult

Implementing a UseCase involves extending the appropriate UseCase interface based on its functionality. For example, `RegisterUserUseCase` and its implementation shown in Listing 23 extends a specific interface tailored for receiving parameters. It takes a `Params` object that encapsulates all the necessary registration details. The primary operation within this UseCase is to call the `registerUser` method from the `AuthRepository`, which handles the registration process.

```
interface RegisterUserUseCase
    : UseCaseResult<RegisterUserUseCase.Params, Unit> {
    data class Params(
        val email: String,
        val password: String,
        val firstName: String,
        val surname: String,
        val username: String
    )
}

internal class RegisterUserUseCaseImpl(
    private val repository: AuthRepository,
) : RegisterUserUseCase {
    override suspend fun invoke(
    params: RegisterUserUseCase.Params
): Result<Unit> =
        repository.registerUser(
            email = params.email,
            ...
        )
}
```

Listing 23: Illustration of Registration Use Case

4.6.2.3 Repository

A repository is responsible for handling all necessary functions related to a specific functionality. For instance, an `AuthRepository` manages all authentication functions such as `loginUser` and `registerUser`, as illustrated in Listing 24. As discussed in the design chapter, adhering to the principle of dependency inversion helps keep the domain and data layers separate. UseCases in the domain layer only interact with the repository interface without being aware of its specific implementation, which is detailed in the data layer. This approach ensures a clean separation of concerns and enhances the system's modularity.

```
interface AuthRepository {
    suspend fun loginUser(
        email: String,
        password: String
    ): Result<Unit>

    suspend fun registerUser( ... ): Result<Unit>
}
```

Listing 24: Illustration of AuthRepository

4.6.3 Data layer

This subsection describes the Data layer, building on the details provided in the design chapter. The Data layer is responsible for storing application data and managing business logic. It includes the implementation of Repositories, as well as data Sources, APIs, and their implementation.

4.6.3.1 Repository implementation

The `RepositoryImpl` serves as an implementation of the Repository interface, designed to keep the Domain Layer isolated from any knowledge of where its data originates. This architectural separation enables the repository to gather data from various sources, including local caches and remote sources. In the case of the `AuthRepositoryImpl` shown in Listing 25, a remote source is injected, providing the necessary data.

```
internal class AuthRepositoryImpl(
    private val source: AuthRemoteSource,
) : AuthRepository {
    override suspend fun loginUser(
        email: String,
        password: String
    ): Result<Unit> =
        source.loginUser(
            email = email,
            password = password
        )

    override suspend fun registerUser(
        email: String,
        password: String,
        firstName: String,
        surname: String,
        username: String
    ): Result<Unit> =
        source.registerUser(
            email = email,
            password = password,
            firstName = firstName,
            surname = surname,
            username = username
        )
}
```

Listing 25: Illustration of AuthRepositoryImpl

4.6.3.2 Source

The Source interface and its implementation are crucial for managing data interactions between the application and external APIs. This component is responsible for retrieving data from specific API implementations and transforming it for use within the application. For instance, it converts data transfer objects (DTOs) into domain objects that are suitable for the application's internal processes. Conversely, the Source also prepares and formats data from the application into payloads suitable for transmission back to the APIs. `PostRemoteSource` and its implementation are shown as an example in Listing 26.

```kotlin
internal interface PostRemoteSource {
    suspend fun getOtherUsersPosts(userId: String): Result<List<Post>>
    suspend fun getUsersPosts(userId: String): Result<List<Post>>
    suspend fun getPost(id: String): Result<Post>
    suspend fun deletePost(id: String): Result<Unit>
    suspend fun addPost( ... ): Result<Unit>

    suspend fun updatePost( ... ): Result<Unit>
}

internal class PostRemoteSourceImpl(
    private val api: PostApi,
) : PostRemoteSource {
    override suspend fun getOtherUsersPosts(userId: String)
    : Result<List<Post>> =
        api.getOtherUsersPosts(userId).map { dto ->
            dto.map { post -> post.toDomain() }
        }

    ...

    override suspend fun updatePost( ... ): Result<Unit> =
        api.updatePost( ... )
}
```

Listing 26: Illustration of PostRemoteSource

4.6.3.3 Api

APIs serve as the primary point of communication between the application and external systems, in this case, Firebase. They establish a controlled gateway for data exchange, ensuring that the application can securely and efficiently fetch and send data. Specifically, they facilitate communication with Firebase Authentication, Firebase Firestore, and Firebase Storage services. An example of this setup is demonstrated in the `PostApi` shown in Listing 27, with its implementation illustrated in Listing 28.

```
internal interface PostApi {
    suspend fun getPosts(): Result<List<PostDto>>
    suspend fun getOtherUsersPosts(userId: String): Result<List<PostDto>>
    suspend fun getUsersPosts(userId: String): Result<List<PostDto>>
    suspend fun getPost(id: String): Result<PostDto>
    suspend fun deletePost(id: String): Result<Unit>
    suspend fun addPost(images: List<File>, post: PostPayload): Result<Unit>
    suspend fun updatePost(
        postId: String,
        post: UpdatePostPayload,
        restOfImages: List<String>?,
        deletedImages: List<String>?,
        newImages: List<File>?,
    ): Result<Unit>
}
```

Listing 27: Illustration of PostApi

```
internal class PostApiImpl(
    private val db: FirebaseFirestore,
    private val storage: FirebaseStorage,
) : PostApi {
    override suspend fun getPosts(): Result<List<PostDto>> { ... }

    ...

    private suspend fun uploadFile(file: File): String { ... }

    private suspend fun deleteFile(fileName: String) { ... }
}
```

Listing 28: Illustration of PostApiImpl

Firebase services are comprehensively documented and offer robust functionality. Specifically, Firestore provides powerful filtering functions that allow for sophisticated collection querying. However, Firestore also presents certain limitations. For example, when a document within a collection contains a subcollection, it is impossible to fetch both the document and its subcollection simultaneously because it must be retrieved in a separate operation. This can be restrictive in scenarios where nested data structures are involved.

For example, when fetching users' posts, as demonstrated in Listing 29, the `.get()` method is utilized to retrieve data, and the `equalTo` function is applied for filtering users'posts. Upon successful data retrieval, a `Result` marked `Success` is returned, containing all the fetched data. Conversely, in the event of an error, a `Result` marked `Error` is returned along with an error message.

```
override suspend fun getUsersPosts(userId: String): Result<List<PostDto>> {
    try {
        val userResponse = db.collection("posts").where {
            "user_id" equalTo userId
        }.get()
        return Result.Success(
            userResponse.documents.map {
                (it.data() as PostDto).copy(id = it.id)
            }
        )
    } catch (e: Exception) {
        return Result.Error(e.message ?: "Unknown Error")
    }
}
```

Listing 29: Illustration of getUsersPosts API call

Firebase Storage provides reliable and straightforward methods for storing and retrieving data. In this application, Firebase Storage is explicitly utilized to store images associated with posts. After a successful image upload, the URLs of these images are stored within the corresponding `Post` object in the Firestore. An example of an image upload is detailed in Listing 30.

```
private suspend fun uploadFile(file: File): String {
    try {
        val storageRef = storage.reference.child(
            "images/img_${Timestamp.now().nanoseconds}${
                Timestamp.now().seconds
            }.png"
        )
        storageRef.putFileResumable(file).collect { progress ->
            when (progress) {
                is Progress.Running -> println(
                    "Upload is running: ${
                        progress.bytesTransferred
                    }/${
                        progress.totalByteCount
                    }"
                )
                is Progress.Paused -> println("Upload is paused")
            }
        }
        val downloadUrl = storageRef.getDownloadUrl()
        return downloadUrl
    } catch (e: Exception) {
        println("Upload failed: ${e.message}")
        throw e
    }
}
```

Listing 30: Illustration of uploadFile to Firebase Storage

Firebase Authentication provides a comprehensive suite of functions. This project utilizes the functionalities for registration, login, and logout. These functions are easy to set up. For instance, as demonstrated in Listing 31, the login process only requires calling `.signInWithEmailAndPassword()`. This function checks if the user is registered and, if so, logs the user in.

```kotlin
override suspend fun loginUser(data: LoginPayload): Result<Unit> {
    try {
        auth.signInWithEmailAndPassword(
            email = data.email,
            password = data.password,
        )
        return Result.Success(Unit)
    } catch (e: Exception) {
        return Result.Error(e.message ?: "Unknown Error")
    }
}
```

Listing 31: Illustration of loginUser to Firebase Authentication

4.7 Documentation

Documentation is crucial in software development for several reasons. It enhances onboarding and collaboration by helping new team members understand the system quickly. Documentation improves maintainability, supports debugging, and ensures consistency across the codebase. Overall, well-documented code is easier to manage, update, and use, reducing maintenance costs and improving software quality. [72]

4.7.1 KDoc

KDoc [73], which extends Javadoc, is the official documentation format for Kotlin code. It uses Markdown syntax alongside special tags and annotations to describe the structure and behavior of Kotlin programs. Tools like Dokka, shown in Section 4.7.2, can parse KDoc to generate documentation in formats like HyperText Markup Language (HTML). Below in Listing 32 is shown how a class might be documented using KDoc.

```kotlin
/**
 * A group of *members*.
 *
 * This class has no useful logic; it's just a documentation example.
 *
 * @param T the type of a member in this group.
 * @property name the name of this group.
 * @constructor Creates an empty group.
 */
class Group<T>(val name: String) {
    /**
     * Adds a [member] to this group.
     * @return the new size of the group.
     */
    fun add(member: T): Int { ... }
}
```

Listing 32: Example of commentary using KDoc format from [73]

4.7.2 Dokka

Dokka [74] is a documentation generation tool developed by JetBrains, the creators of Kotlin. It parses KDoc annotations to produce documentation in various formats, including HTML and Markdown. Designed to integrate seamlessly with Kotlin's build tools and IDEs, Dokka supports multi-module projects, making it suitable for this project.

Integrating Dokka with Kotlin's build system, particularly Gradle, is straightforward. The plugin must be added to the *build.gradle* file of each module that requires documentation. Additional configurations can be customized to meet specific needs, such as modifying visibility levels or adding custom pages. Below in Listing 33 is shown how was Dokka configured in the project's *build.gradle* files.

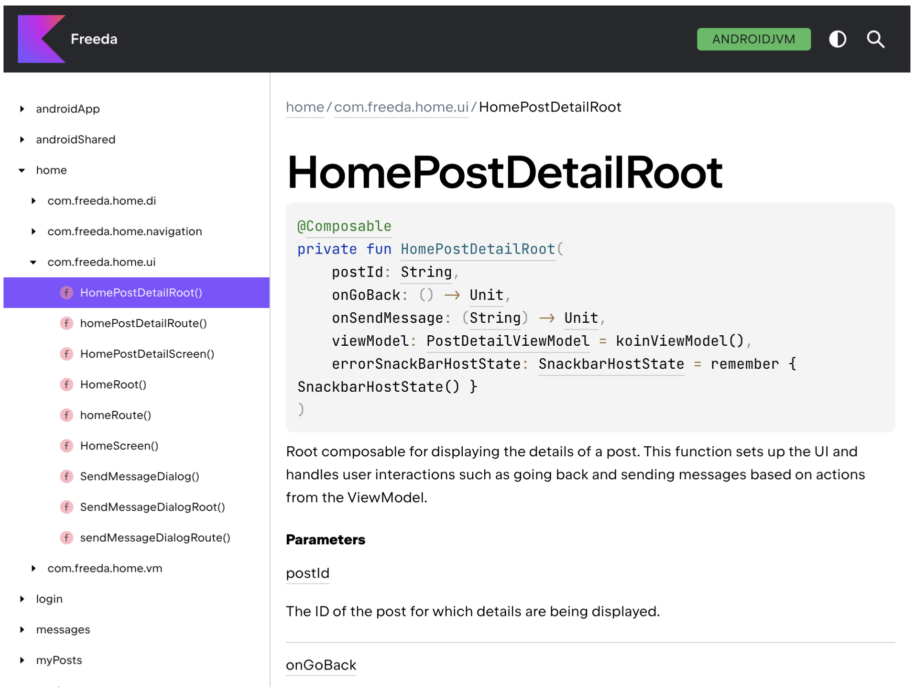
```kotlin
tasks.withType<DokkaTaskPartial>().configureEach {
    dokkaSourceSets {
        configureEach {
            documentedVisibilities.set(
                setOf(
                    DokkaConfiguration.Visibility.PUBLIC,
                    DokkaConfiguration.Visibility.INTERNAL,
                    DokkaConfiguration.Visibility.PRIVATE,
                )
            )
            suppressInheritedMembers.set(true)
        }
    }
}
```

Listing 33: Illustration of Dokka configuration

Team members gain access to comprehensive documentation of all components, including private and internal APIs that are crucial for understanding the system's inner workings but are not exposed externally. This inclusivity ensures that developers can fully grasp the functionality and interactions within the application.

Setting `suppressInheritedMembers` to `true` ensures that inherited members are not documented repeatedly across subclasses unless they are overridden. This approach significantly reduces clutter in the documentation, making it easier to identify unique features and behaviors of each class. It helps to highlight what is specifically implemented in a subclass, as opposed to what it inherits, enhancing clarity for developers navigating the documentation.

Running `./gradlew dokkaHtmlMultiModule`, after configuring Dokka appropriately, generates a comprehensive HTML documentation. This documentation is stored in the root build directory and is structured to provide a clear and accessible overview of the project's codebase. The structure of this documentation, as illustrated in Figure 4.3, allows developers to find relevant information quickly, facilitating better understanding and more efficient use of the codebase.



HomePostDetailRoot

```
@Composable
private fun HomePostDetailRoot(
    postId: String,
    onGoBack: () -> Unit,
    onSendMessage: (String) -> Unit,
    viewModel: PostDetailViewModel = koinViewModel(),
    errorSnackBarHostState: SnackbarHostState = remember {
SnackbarHostState() }
)
```

Root composable for displaying the details of a post. This function sets up the UI and handles user interactions such as going back and sending messages based on actions from the ViewModel.

Parameters

postId

The ID of the post for which details are being displayed.

onGoBack

Figure 4.3: Illustration of Dokka documentation

4.8 Testing

Testing is vital in software development as it ensures that applications are reliable, secure, and user-friendly. It helps catch bugs early, reduces development costs, and supports continuous improvement. Ultimately, thorough testing is critical for delivering high-quality software that meets both user expectations and business goals. [75]

4.8.1 Unit testing

Unit tests are a form of software testing that focuses on individual components or sections of code within an application to ensure they function correctly. Developers typically write and run these tests during the coding process to verify that each specific function or module performs as intended. By testing these smaller parts independently before integrating them into the broader system, unit tests help catch issues early, simplifying debugging and enhancing overall software quality. [76]

4.8.1.1 JUnit and Mockito

JUnit [77] and Mockito [78] are essential tools for unit testing in Java and Kotlin projects. JUnit is used for setting up and executing tests, providing annotations to define and manage test methods, and assertions to verify the code's correctness. Mockito, on the other hand, is used to mock class dependencies, allowing the creation of mock objects and the specification of their behaviors, which is crucial for testing classes in isolation from their actual dependencies. An example of a unit test using JUnit and Mockito is shown in the `LoginViewModelTest` in Listing 34.

```
class LoginViewModelTest {
    private lateinit var loginUseCase: LoginUseCase
    private lateinit var viewModel: LoginViewModel

    @Before
    fun setUp() {
        val testDispatcher = StandardTestDispatcher()
        Dispatchers.setMain(testDispatcher)
        loginUseCase = mock()
        viewModel = LoginViewModel(loginUseCase)
    }

    @After
    fun tearDown() { Dispatchers.resetMain() }

    @Test
    fun loginSuccess() = runTest {
        val loginData = Login("user@mail.com", "password")
        whenever(loginUseCase(loginData)).thenReturn(
            Result.Success(Unit)
        )
        ...
        val action = viewModel.viewAction.first()
        assertTrue(action is LoginViewModel.ViewAction.LogIn)
    }
    ...
}
```

Listing 34: Illustration of a unit test

To ensure that tests run consistently and predictably the `setUp` method prepares the testing environment by using a `StandardTestDispatcher` to manage coroutine execution. It also mocks the `loginUseCase` to simulate functionality without actual implementation and initializes the `LoginViewModel` with this mock. The `tearDown` method resets `Dispatchers.Main` to prevent cross-test interference.

The `@Test` annotation is essential in JUnit, indicating which methods are test cases. It ensures that each test method is executed independently, maintaining test isolation and straightforward error tracking.

In test methods, assertions are critical for verifying that the code behaves as expected. They check conditions to ensure the actual outcomes match the anticipated results, validating the code's performance during testing.

4.8.2 User testing

User testing can be performed after completing the application's implementation. This method evaluates a product by having real users test it, allowing designers and developers to observe how it is used in practice. This helps find problems and gather ideas for solutions, improving how we understand the way users interact with the product and what changes are needed. [79]

4.8.2.1 Testers

Five testers were selected for user testing: one is a mobile app developer, two are software engineering students, and the remaining two are individuals from non-IT backgrounds. This diverse mix provides the author with a broad spectrum of user perspectives.

4.8.2.2 Testing scenarios

The scenarios outline specific situations or tasks users might perform when interacting with the application, ensuring that the application's features work as intended. There are nine scenarios, each testing a different application feature, detailed in Appendix C. These scenarios cover the most important and critical parts of the application. By following these testing scenarios, testers can systematically evaluate the application's behavior, identify any issues, and ensure a smooth and reliable user experience.

Before providing the testers with step-by-step instructions, certain conditions needed to be met for each scenario, such as being logged into the application or having registration information. These prerequisites were managed in advance and given to the testers at the beginning of the testing session. Once everything was prepared, the author gave the testers step-by-step instructions and guided them throughout the testing process.

4.8.2.3 Testers feedback

The testing process went smoothly, with no issues encountered. Testers appreciated the simplicity and intuitiveness of the application, noting that the color scheme is calming and well-suited to the app's purpose. They particularly liked that the app allows for seamless communication between users without needing any external tools.

However, two testers—one a mobile app developer and the other with a non-IT background—identified a couple of potential issues. They pointed out that once a post is marked as picked up, it cannot be deleted, which could lead to a cluttered and chaotic interface if the user has a lot of posts. This restriction was initially intended as a way to archive posts, but it may need reconsideration. Additionally, all testers suggested that an alert should be added when marking a post as picked up since this action cannot be undone and the post will be hidden from other users. The mobile app developer also highlighted a potential flaw in the user rating system. They observed that users can rate others multiple times consecutively. This could result in a disproportionate number of ratings from a single user, potentially skewing the overall ratings and diminishing their reliability, especially if other users rate infrequently.

4.8.2.4 Changes according to the testers feedback

Based on the feedback from the testers, the author made appropriate changes to the application.

The ability to delete archived posts was added to prevent the interface from becoming cluttered and chaotic. When marking a post as picked up, a confirmation dialog was added to ensure users do not perform this action accidentally and are aware that it cannot be undone, causing the post to be hidden from other users.

The issue with the user rating system was addressed by allowing users to rate each other only once after initiating contact through a post. This change helps ensure that ratings remain reliable and are not disproportionately influenced by a single user. The author considered the scenario where a user might send a message through the same post again instead of using the direct messaging feature to be an edge case that would only occur occasionally. Therefore, the new rating system provides more controlled and accurate user ratings.

4.9 Continuous Integration/Continuous Delivery

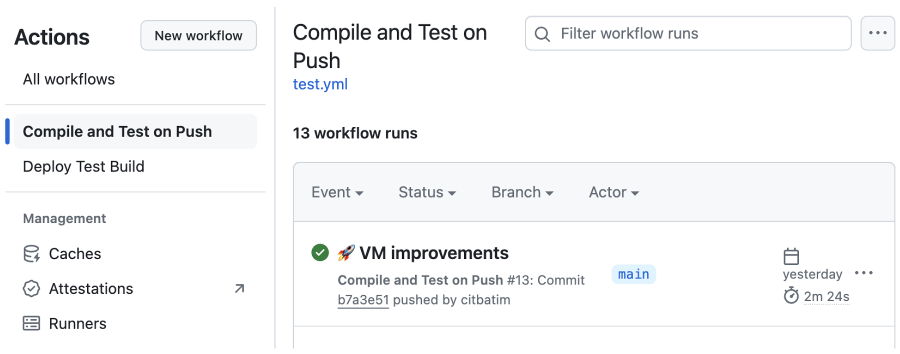
This project uses GitHub for version control and tracking and managing changes within the project. It incorporates both Continuous Integration (CI) and Continuous Deployment (CD) through GitHub Actions.

The ”Compile and Test on Push” workflow, shown in Listing 35, activates upon any new push to the repository. This workflow automatically runs unit tests to validate changes, helping to identify errors. It also compiles the project, ensuring that new changes integrate well without causing disruptions in the build process.

```yaml
name: Compile and Test on Push

on: push

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-java@v3
        with:
          distribution: 'temurin'
          java-version: '17'

      - name: Setup Gradle
        uses: gradle/gradle-build-action@v2

      - name: Make gradlew executable
        run: chmod +x ./gradlew

      - name: Execute Gradle command - Run unit tests
        run: ./gradlew test

      - name: Compile Sources
        run: ./gradlew compileDebugSources
```

Listing 35: Illustration of Compile and Test

After every code push to GitHub, the results of the compilation and tests are automatically displayed in the GitHub Actions workflows, as shown in Figure 4.4.



13 workflow runs

🚀 VM improvements
Compile and Test on Push #13: Commit b7a3e51 pushed by citbatim
2m 24s

Figure 4.4: Illustration of workflow after every push

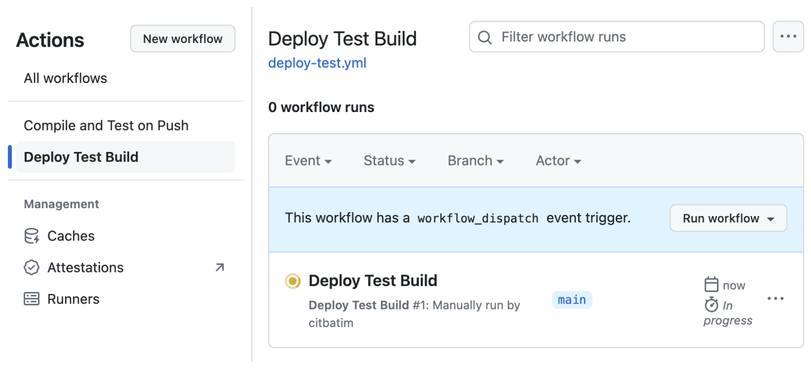
The “Deploy Test Build” workflow shown in Listing 36, facilitates the CD of new application test versions to testers. Initiated manually through GitHub Actions, this workflow allows developers to launch deployments whenever necessary.

```yaml
name: Deploy Test Build

on:
  workflow_dispatch:
    inputs:
      release_notes:
        type: string
        required: false
        description: 'Notes'

jobs:
  build:
    name: Building and distributing app
    runs-on: ubuntu-latest
    steps:
      ...
      - name: Execute Gradle command - assembleDebug
        run: ./gradlew assembleDebug

      - name: Upload Artifact to Firebase App Distribution
        uses: wzieba/Firebase-Distribution-Github-Action@v1
        with:
          appId: ${{ secrets.FIREBASE_APP_ID }}
          serviceCredentialsFileContent:
            ${{ secrets.CREDENTIAL_FILE_CONTENT }}
          groups: testers
          file:
            android/androidApp/build/outputs/apk/debug/androidApp-debug.apk
          releaseNotes: ${{ inputs.release_notes }}
```

Listing 36: Illustration of Test Deploy

During the initiation process, developers can input release notes detailing the updates included in the test build.

When deploying a new test version for testers, after initiating the workflow, the running job and its results are displayed in the GitHub Actions, as depicted in Figure 4.5. After the app is built and tested successfully, the APK is uploaded to Firebase App Distribution. Testers then receive an email to download the app for testing.



This workflow has a `workflow_dispatch` event trigger.

Deploy Test Build
Deploy Test Build #1: Manually run by citbatim

Figure 4.5: Illustration of workflow for deployment

Chapter 5

Evaluation

In this final chapter, the author evaluates the results of the application and explores possibilities for its future development.

5.1 Results

In the analysis, waste pollution and its solutions were introduced. The exploration of a mobile app as a means to improve our planet's health and encourage humane behavior was discussed. This led to a market research phase that highlighted the positives and potential inspirations for the app. Based on this, both functional and non-functional requirements were established, followed by use cases to demonstrate the app's functionality and a domain model to define the entities and their relationships.

The choice of Kotlin, along with Jetpack Compose, proved to be an excellent approach for this project. Kotlin's modern language features, strong community support, and seamless integration with Android Studio and official Android documentation made development efficient and enjoyable. Jetpack Compose, Google's modern toolkit for building native UI, pairs perfectly with Kotlin, allowing for clear and readable code. This combination simplifies and accelerates UI development by reducing boilerplate code and providing a more intuitive syntax.

Using clean architecture with MVI made the application highly extendable, with each part easily replaceable. This approach ensures maintainability and scalability, aligning well with established best practices for Android development. Clean architecture promotes separation of concerns, making the codebase easier to manage and evolve.

Using Firebase was straightforward and beneficial. It offered a wide range of technologies, such as real-time databases, authentication, and analytics, enhancing the application's quality and security. Firebase also facilitated the deployment of test versions, making future development and testing more efficient. Firebase's services are well-documented and widely used in the industry, supporting rapid development and iteration.

The final application is both functional and tested. All issues identified during testing were addressed. The application meets all the *Must have* and *Should have* requirements, and most of the *Could have* requirements were implemented. The display of terms and conditions was not implemented due to the need for legal consultation, and features like saving posts and displaying a map with the pickup location were not implemented due to time constraints.

The result of this thesis is a mobile application for the Android platform that simplifies the process of giving away items. According to testers, it is well-colored and easy to navigate, which are important attributes for a successful mobile application.

5.2 Future development

The application works well, but there are several areas where improvements could enhance the user experience.

As mentioned in the results, the feature to save posts still needs to be implemented. This could be a valuable addition, encouraging users to engage with the application more frequently. Users could save posts they are interested in but not ready to decide on, allowing them to revisit these posts later.

Integrating a map to show pickup locations could significantly improve user experience. Not only would it provide a visually appealing element, but it would also make it clearer for users to see where they need to go to pick up an item. This feature could be particularly helpful for users unfamiliar with the location, as it would provide a visual reference.

Adding notifications for messages could also enhance the application's functionality. Allowing users to receive notifications when they have new messages, with settings to control these notifications, could improve the application's overall quality.

Additionally, implementing notifications for new posts near the user could be another useful feature. This would alert users to new opportunities in their vicinity, making the application more dynamic and engaging. However, it is crucial to include settings for this notification feature to allow users to customize their preferences, as not everyone may want to receive these alerts.

Furthermore, the application is easily extendible to iOS because the domain and data layers are implemented using Kotlin Multiplatform, allowing for a shared codebase across both platforms.

Conclusion

The main goal of this thesis was to develop an application that enables users to donate items they no longer need. This initiative benefits the donors by decluttering their space, aids those in need by providing them with valuable items, and, on a broader scale, contributes positively to environmental conservation and the health of our planet.

The thesis included analysis, research, setting requirements, designing the application, implementing, and evaluating the final product.

The analytical phase introduces the issue of waste pollution, establishing its importance. It explains how people dispose of items and why a mobile app for donating items is a viable solution. Research on existing apps with similar focuses identifies their pros and cons, providing a detailed overview of possible improvements and features that the app could adopt and adapt.

Insights from analyzing existing applications define the functional and non-functional requirements for the project. These requirements transform into concrete use cases, providing clear scenarios of how the application should function from the user's perspective. Based on these use cases, a domain model was developed to structure the main entities and their relationships, which are essential for the design and implementation of the application.

The design phase focuses on choosing the language and architecture, considering sustainability and extensibility. It discusses the Android platform and identifies the most appropriate data storage solutions for this project. Technologies are selected to develop an attractive user interface, and a user interface design is created using wireframes.

In the implementation phase, the project structure and specific technologies are detailed. The description covers the individual components of the three-layer architecture and the division of the overall application into modules. It explains how dependency injection is handled, how Gradle is set up, and how Firebase technology is utilized. The project documentation and unit tests are also detailed. In addition to unit tests, user tests are performed.

The results of the project are summarized, highlighting both successes and areas for improvement and discussing potential future developments for the application. The project successfully accomplished all of its goals.

Bibliography

[1] Econation. Waste and pollution — Hazardous Waste — Econation. https://econation.one/waste-and-pollution/, (Accessed on 04/19/2024).

[2] University of Calgary. Pollution vs waste - Energy Education. https://energyeducation.ca/encyclopedia/Pollution_vs_waste, (Accessed on 04/19/2024).

[3] Meuresiduo Solucoes Tecnologicas LTDA. How Much Do We Waste? A Data-Driven Guide to Waste and Landfills. https://www.meuresiduo.com/blog-en/how-much-do-we-waste-a-data-driven-guide-to-waste-and-landfills/, (Accessed on 04/19/2024).

[4] Atlas Disposal. The Pollution Problem and What We Can do About it. https://atlasdisposal.com/blog/the-pollution-problem-and-what-we-can-do-about-it/, (Accessed on 04/19/2024).

[5] HWH Environmental. All About Landfills: Uses, Types, and More. https://www.hwhenvironmental.com/landfills-101/, (Accessed on 04/21/2024).

[6] Earth How. The 3 R's - Reduce, Reuse, and Recycle. https://earthhow.com/reduce-reuse-recycle/, (Accessed on 04/19/2024).

[7] Citizens Information Board. Reducing waste. https://www.citizensinformation.ie/en/environment/waste-and-recycling/reducing-waste/, (Accessed on 04/20/2024).

[8] The Waste and Resources Action Programme. Why is recycling important? https://www.recyclenow.com/how-to-recycle/why-is-recycling-important, (Accessed on 04/20/2024).

[9] Business Waste Ltd. How to Reduce Waste — Ways of Reducing Waste. https://www.businesswaste.co.uk/reduce-waste/, (Accessed on 04/21/2024).

[10] United States Environmental Protection Agency. Reducing and Reusing Basics. https://www.epa.gov/recycle/reducing-and-reusing-basics, (Accessed on 04/21/2024).

[11] Sherman, R. Before You Recycle, Choose to Reuse. https://content.ces.ncsu.edu/before-you-recycle-choose-to-reuse, (Accessed on 04/21/2024).

[12] GREEN LIFE. Reduce, Reuse, Recycle - GREEN LIFE - Be Ecofriendly. https://gulsheenkbhatia.altervista.org/reduce-reuse-recycle/, (Accessed on 04/21/2024).

[13] Kazmi, F. The history of thrift shopping: Exploring the origins and evolution of the thrift industry. https://goodfair.com/blogs/nonewthings/he-history-of-thrift-shopping-exploring-the-origins-and-evolution-of-the-thrift-industry, 2023, (Accessed on 04/21/2024).

[14] ThriftCart. Where do thrift stores get their inventory? https://download.thriftcart.com/where_do_thrift_stores_get_their_inventory, (Accessed on 04/21/2024).

[15] Google LLC. Google. https://www.google.com/, (Accessed on 05/15/2024).

[16] Google LLC. Android Apps on Google Play. https://play.google.com/store/games?hl=en&gl=US, (Accessed on 05/15/2024).

[17] Google LLC. Olio — Share More, Waste Less - Apps on Google Play. https://play.google.com/store/apps/details?id=com.olioex.android&hl=en_US, (Accessed on 04/21/2024).

[18] Google. Freegle - Apps on Google Play. https://play.google.com/store/apps/details?id=org.ilovefreegle.direct&hl=en_US, (Accessed on 04/21/2024).

[19] Google LLC. trash nothing! - Apps on Google Play. https://play.google.com/store/apps/details?id=com.trashnothing.app3&hl=en_US, (Accessed on 04/21/2024).

[20] Gorbachenko, P. Functional vs Non-Functional Requirements. https://enkonix.com/blog/functional-requirements-vs-non-functional/, 2021, (Accessed on 04/22/2024).

[21] Lucas_DevSamurai_. Understanding the MoSCoW prioritization. https://community.atlassian.com/t5/App-Central/Understanding-the-MoSCoW-prioritization-How-to-implement-it-into/ba-p/2463999, 2023, (Accessed on 04/22/2024).

[22] ActiveCollab Team. MoSCoW Method of Prioritization. https://activecollab.com/blog/project-management/moscow-method, (Accessed on 04/24/2024).

[23] Kvartalnyi, N. Functional Vs. Non-Functional Requirements: Why Are Both Important? https://inoxoft.com/blog/functional-vs-non-functional-requirements-why-are-both-important/, 2023, (Accessed on 04/22/2024).

[24] Mishra, A. Designing Use Cases for a Project. https://www.geeksforgeeks.org/designing-use-cases-for-a-project/, 2022, (Accessed on 04/25/2024).

[25] Nishadha. Use Case Diagram Relationships Explained with Examples. https://creately.com/blog/diagrams/use-case-diagram-relationships/, 2022, (Accessed on 04/24/2024).

[26] Chursin, O. A Brief Introduction to Domain Modeling. https://olegchursin.medium.com/a-brief-introduction-to-domain-modeling-862a30b38353, 2017, (Accessed on 04/25/2024).

[27] Bandarupalli, K. Domain Model Using UML. https://www.techbubbles.com/softwarearchitecture/domain-model-using-uml/, 2009, (Accessed on 04/25/2024).

[28] The Editors of Encyclopaedia Britannica. Android — Definition, History, & Facts. https://www.britannica.com/technology/Android-operating-system, 2024, (Accessed on 04/29/2024).

[29] BasuMallick, C. Android OS: History, Features, Versions, and Benefits. https://www.spiceworks.com/tech/tech-general/articles/android-os/, 2024, (Accessed on 04/29/2024).

[30] endoflife.date. Android OS. https://endoflife.date/android, 2024, (Accessed on 04/29/2024).

[31] Morgan, C. The Rise of Android: From Obscurity to the Top. https://www.socialmediatoday.com/content/rise-android-obscurity-top, 2014, (Accessed on 04/29/2024).

[32] Trogrlic, I. 5 best languages for Android app development. https://decode.agency/article/android-app-development-best-languages/, 2023, (Accessed on 04/29/2024).

[33] Gircenko, G. Kotlin vs. Java: All-purpose Uses and Android Apps. https://www.toptal.com/kotlin/kotlin-vs-java, (Accessed on 04/29/2024).

[34] Nayak, S. K. Exploring the Disadvantages of Flutter for Mobile App Development. https://shivamkumarnayak.medium.com/exploring-the-disadvantages-of-flutter-for-mobile-app-development-9b3703b270b, 2023, (Accessed on 04/29/2024).

[35] Google LLC. Meet Android Studio — Android Developers. https://developer.android.com/studio/intro, (Accessed on 05/02/2024).

[36] Google LLC. Guide to app architecture — Android Developers. https://developer.android.com/topic/architecture, (Accessed on 05/04/2024).

[37] Lackner, P. What Is the Best Architecture for Android Apps? https://www.youtube.com/watch?v=cnU2zMnmmpg, 2022, (Accessed on 05/03/2024).

[38] Lackner, P. MVVM vs. MVI - Understand the Difference Once and for All. https://www.youtube.com/watch?v=b2z1jvD4VMQ&t=263s, 2024, (Accessed on 05/03/2024).

[39] Ankiersztajn, M. MVI Architecture Explained On Android. https://blog.stackademic.com/mvi-architecture-explained-on-android-e36ee66bceaa, 2024, (Accessed on 05/03/2024).

[40] Pathak, A. MVVM to MVI: A Guide to Migrating Your Android Architecture. https://medium.com/@myofficework000/mvvm-to-mvi-a-guide-to-migrating-your-android-architecture-8d3cb5bb9f06, 2023, (Accessed on 05/03/2024).

[41] Bahramitooran, T. Benefits and Challenges of Applying Clean Architecture to Existing Software Projects. https://medium.com/@tannazbahramitooran/benefits-and-challenges-of-applying-clean-architecture-to-existing-software-projects-b8b4ff98c5df, 2023, (Accessed on 05/04/2024).

[42] My, N. T. Clean Architecture for Android App - Reasoning Process. https://engineering.dena.com/blog/2023/06/clean-architecture-for-android-app-reasoning-process/, 2023, (Accessed on 05/04/2024).

[43] Google LLC. Domain layer — Android Developers. https://developer.android.com/topic/architecture/domain-layer, (Accessed on 05/04/2024).

[44] Mishra, A. Understanding Domain Objects, Entities, DTOs, and Models in C#. https://medium.com/@mishraabhinn/understanding-domain-objects-entities-dtos-and-models-in-c-207bb5c1d97c, (Accessed on 05/05/2024).

[45] Google LLC. Data layer — Android Developers. https://developer.android.com/topic/architecture/data-layer, (Accessed on 05/04/2024).

[46] Google LLC. Data layer — Android Developers. https://developer.android.com/topic/architecture/data-layer, (Accessed on 05/05/2024).

[47] Reppas, K. Clean Architecture in Android VS. Official Documentation. https://www.youtube.com/watch?v=t0ejplwuw3M, 2023, (Accessed on 05/05/2024).

[48] Pham, Q. Firebase vs AWS: Exploring the Core Features, Pros and Cons. https://www.orientsoftware.com/blog/firebase-vs-aws/, 2023, (Accessed on 05/03/2024).

[49] Kuten, A. AWS vs Firebase: The Key Differences. https://cloudvisor.co/blog/aws-vs-firebase/, 2023, (Accessed on 05/05/2024).

[50] amangfg. Firebase vs AWS: Top Differences. https://www.geeksforgeeks.org/firebase-vs-aws/, 2024, (Accessed on 05/03/2024).

[51] Shokoya, A. XML LAYOUT VS JETPACK COMPOSE, WHICH IS BETTER? https://medium.com/@MeenoTeK/xml-views-or-jetpack-compose-which-is-the-best-option-for-your-next-project-ccf38573a82, 2023, (Accessed on 05/02/2024).

[52] Bellini, A.-C. Android Developers Blog: Jetpack Compose is now 1.0: announcing Android's modern toolkit for building native UI. https://android-developers.googleblog.com/2021/07/jetpack-compose-announcement.html, 2021, (Accessed on 05/02/2024).

[53] Raza, S. A. XML vs Jetpack Compose: Choosing the Best UI Approach for Android. https://aliraza112.medium.com/xml-vs-jetpack-compose-choosing-the-best-ui-approach-for-android-463d56aca983, 2023, (Accessed on 05/02/2024).

[54] Google LLC. Develop a UI with Views — Android Studio — Android Developers. https://developer.android.com/studio/write/layout-editor, (Accessed on 05/02/2024).

[55] Google LLC. Layouts in Views — Android Developers. https://developer.android.com/develop/ui/views/layout/declaring-layout, (Accessed on 05/02/2024).

[56] Ahire, V. When to use Fragments VS Activities in Android App? https://www.tutorialspoint.com/when-to-use-fragments-vs-activities-in-android-app, 2023, (Accessed on 05/02/2024).

[57] Saumell, J. Compose (UI) beyond the UI (Part I): big changes. https://proandroiddev.com/compose-ui-beyond-the-ui-part-i-big-changes-bfe824ee8ed4, 2021, (Accessed on 05/02/2024).

[58] Google LLC. Jetpack Compose UI App Development Toolkit - Android Developers. https://developer.android.com/develop/ui/compose, (Accessed on 05/02/2024).

[59] Google LLC. Why Compose — Jetpack Compose — Android Developers. https://developer.android.com/develop/ui/compose/why-adopt, (Accessed on 05/02/2024).

[60] Simic, P. Everything you need to know about mobile app wireframing. https://decode.agency/article/mobile-app-wireframing/, 2021, (Accessed on 04/30/2024).

[61] Protoio Inc. Proto.io - Prototyping for all. https://proto.io/, (Accessed on 05/02/2024).

[62] JetBrains s.r.o. Kotlin Multiplatform for Cross-Platform Development. https://www.jetbrains.com/kotlin-multiplatform/, (Accessed on 05/14/2024).

[63] Google LLC. Guide to Android app modularization — Android Developers. https://developer.android.com/topic/modularization, (Accessed on 05/14/2024).

[64] Gradle Inc. Gradle Build Tool. https://gradle.org/, (Accessed on 05/14/2024).

[65] Google LLC. Firebase — Google's Mobile and Web App Development Platform. https://firebase.google.com/, (Accessed on 05/14/2024).

[66] Ugaz, C. How to implement Firebase Firestore in Kotlin Multiplatform Mobile (KMM) with Compose-Multiplatform. https://medium.com/@carlosgub/how-to-implement-firebase-firestore-in-kotlin-multiplatform-mobile-with-compose-multiplatform-32b66cdba9f7, 2023, (Accessed on 05/05/2024).

[67] Google LLC. Configure your build — Android Studio — Android Developers. https://developer.android.com/build, (Accessed on 05/05/2024).

[68] GitLive. GitHub - GitLiveApp/firebase-kotlin-sdk: A Kotlin-first SDK for Firebase. https://github.com/GitLiveApp/firebase-kotlin-sdk?tab=readme-ov-file, (Accessed on 05/05/2024).

[69] Google LLC. Dependency injection in Android — Android Developers. https://developer.android.com/training/dependency-injection, (Accessed on 05/05/2024).

[70] Koin & Kotzilla. Koin - The pragmatic Kotlin Injection Framework - developed by Kotzilla and its open-source contributors. https://insert-koin.io/, (Accessed on 05/14/2024).

[71] Google LLC. Dependency injection with Hilt — Android Developers. https://developer.android.com/training/dependency-injection/hilt-android#hilt-and-dagger, (Accessed on 05/14/2024).

[72] Gec. The Crucial Role of Documentation in Coding: A Developer's Guide. https://medium.com/@gecno/the-crucial-role-of-documentation-in-coding-a-developers-guide-16c6a741bed8, 2024, (Accessed on 05/14/2024).

[73] JetBrains s.r.o. Document Kotlin code: KDoc — Kotlin Documentation. https://kotlinlang.org/docs/kotlin-doc.html#kdoc-syntax, 2024, (Accessed on 05/11/2024).

[74] JetBrains s.r.o. Introduction — Kotlin Documentation. https://kotlinlang.org/docs/dokka-introduction.html, 2024, (Accessed on 05/11/2024).

[75] Khatri, M. F. Beyond Bugs: Exploring the Depths of Software Testing. https://medium.com/@iamfaisalkhatri/beyond-bugs-exploring-the-depths-of-software-testing-6a3b7057060e, 2023, (Accessed on 05/14/2024).

[76] pp_pankaj. Unit Testing - Software Testing. https://www.geeksforgeeks.org/unit-testing-software-testing/, 2024, (Accessed on 05/11/2024).

[77] JetBrains s.r.o. Test code using JUnit in JVM – tutorial — Kotlin Documentation. https://kotlinlang.org/docs/jvm-test-using-junit.html, 2023, (Accessed on 05/13/2024).

[78] Szczepan Faber and friends. GitHub - mockito/mockito: Most popular Mocking framework for unit tests written in Java. https://github.com/mockito/mockito, (Accessed on 05/13/2024).

[79] Omniconvert. What is User testing? Definition. https://www.omniconvert.com/what-is/user-testing/, 2024, (Accessed on 05/13/2024).

Acronyms

AS Android Studio. 19, 25, 26, 34

AWS Amazon Web Services. 24, 25

CD Continuous Deployment. 58, 59

CI Continuous Integration. 58

DI Dependency injection. 37, 38

HTML HyperText Markup Language. 53–55

IDE Integrated Development Environment. 19, 54

KMP Kotlin Multiplatform. 33, 36, 37

MVI Model-View-Intent. xi, 20–22, 38, 45

MVVM Model-View-ViewModel. xi, 20, 21

NDK Native Development Kit. 18, 19

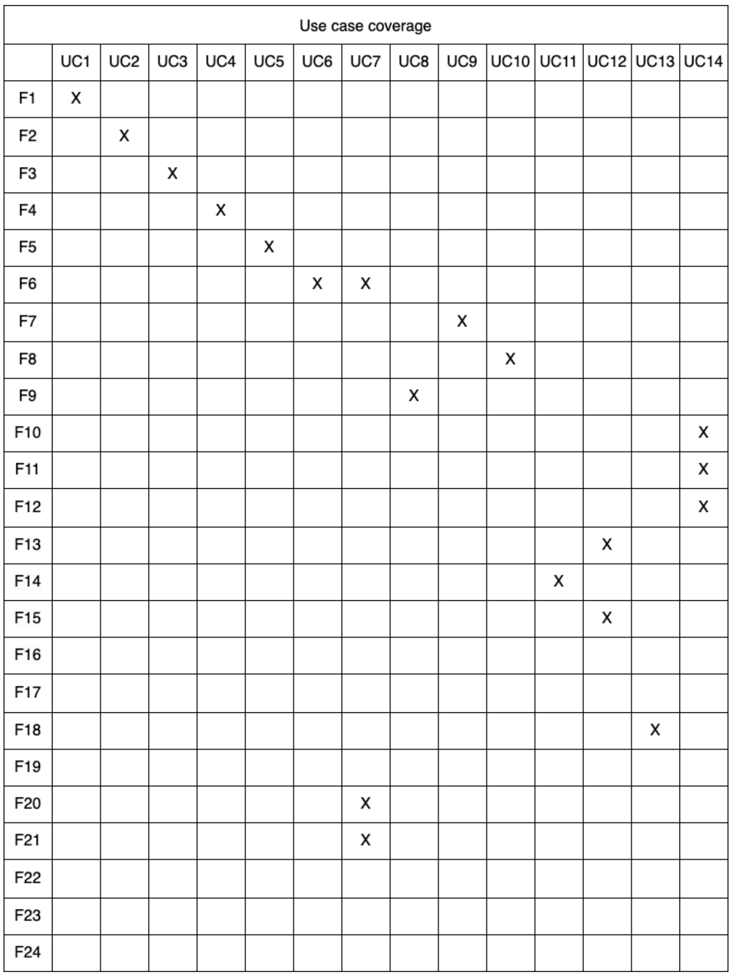
UI User Interface. 17, 19, 20, 22, 23, 25, 26, 38, 41, 43–45

URL Uniform Resource Locator. 15

XML Extensible Markup Language. 25, 26

Appendix **A**

Use case coverage



Use case coverage														
	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	UC10	UC11	UC12	UC13	UC14
F1	X													
F2		X												
F3			X											
F4				X										
F5					X									
F6						X	X							
F7									X					
F8										X				
F9								X						
F10														X
F11														X
F12														X
F13												X		
F14											X			
F15												X		
F16														
F17														
F18													X	
F19														
F20							X							
F21							X							
F22														
F23														
F24														

Figure A.1: Use case coverage of functional requirements

Appendix B

Application screens

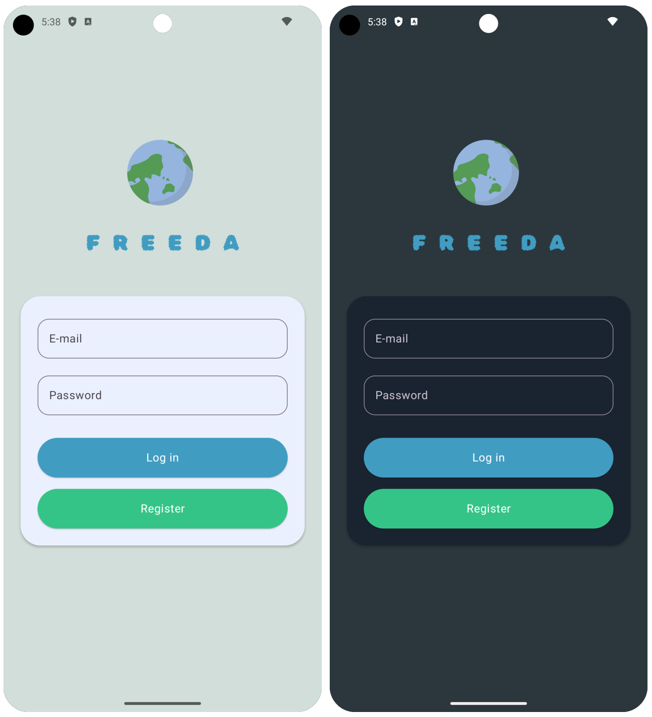


Figure B.1: Login screen

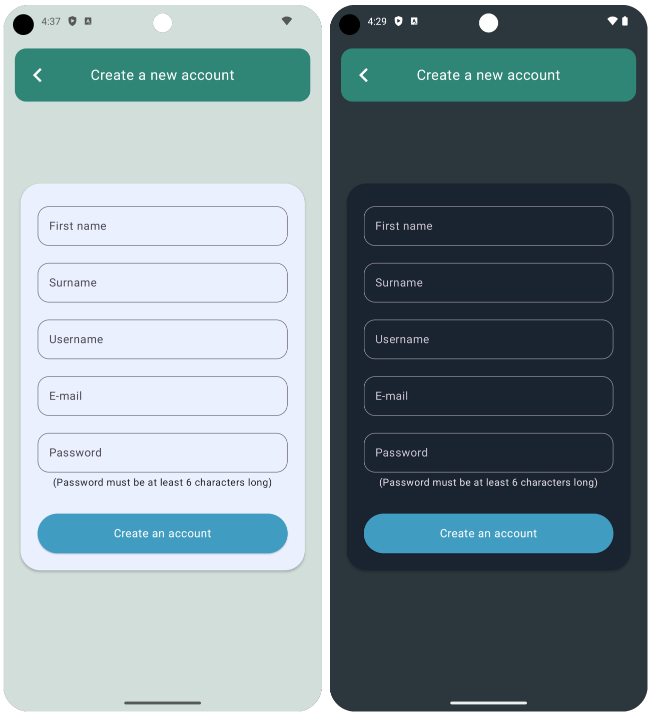


Figure B.2: Create a new account screen

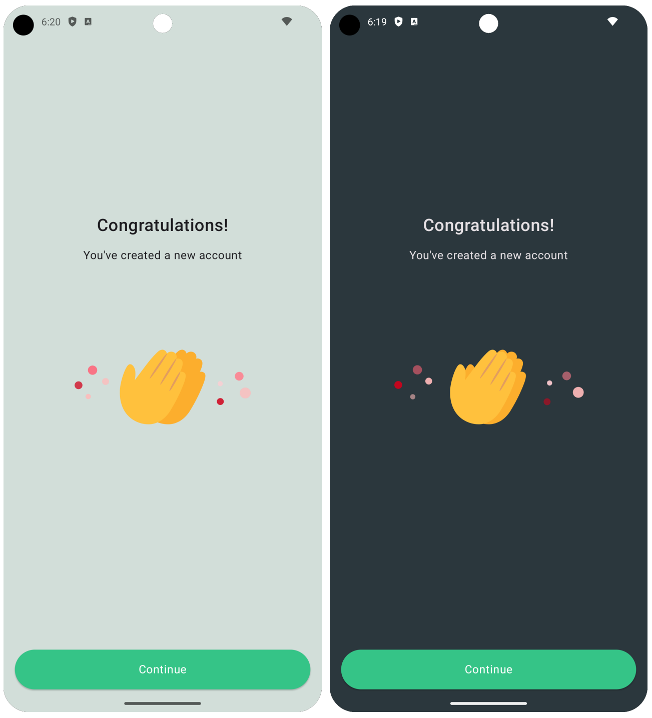
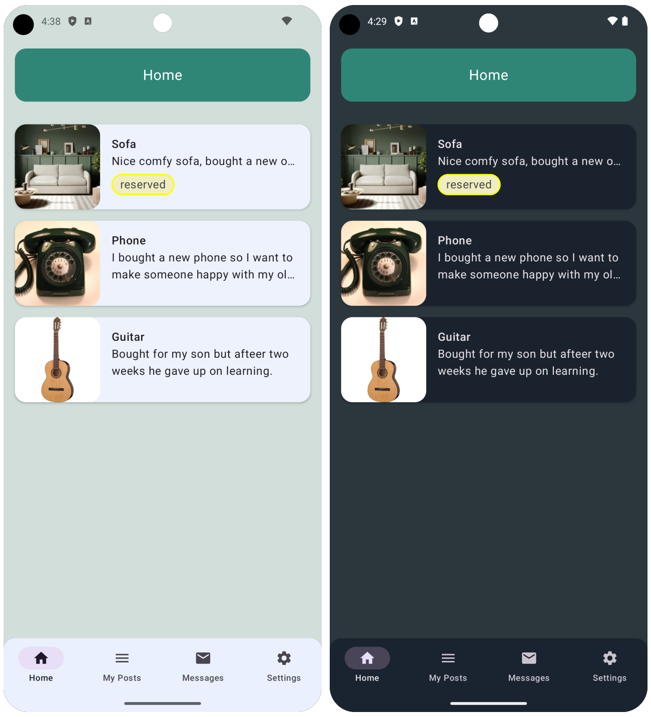


Figure B.3: New account created successfully screen



Sofa
Nice comfy sofa, bought a new o…
reserved

Phone
I bought a new phone so I want to make someone happy with my ol…

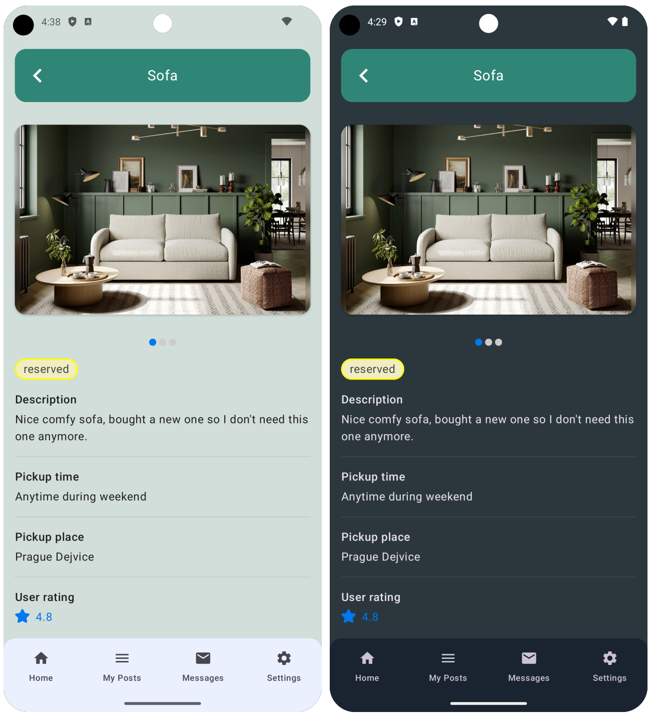
Guitar
Bought for my son but afteer two weeks he gave up on learning.

Sofa
Nice comfy sofa, bought a new o…
reserved

Phone
I bought a new phone so I want to make someone happy with my ol…

Guitar
Bought for my son but afteer two weeks he gave up on learning.

Figure B.4: Home screen



reserved

Description
Nice comfy sofa, bought a new one so I don't need this one anymore.

Pickup time
Anytime during weekend

Pickup place
Prague Dejvice

User rating
4.8

reserved

Description
Nice comfy sofa, bought a new one so I don't need this one anymore.

Pickup time
Anytime during weekend

Pickup place
Prague Dejvice

User rating
4.8

Figure B.5: Post detail screen

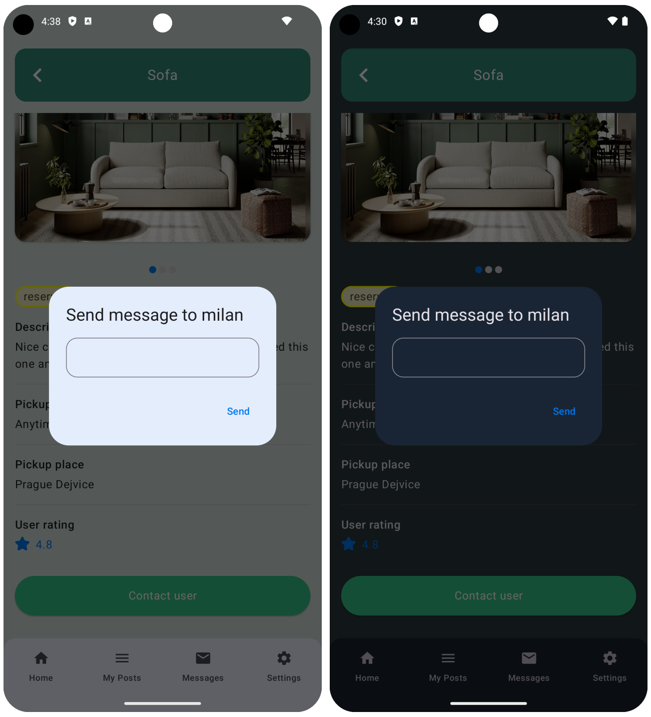
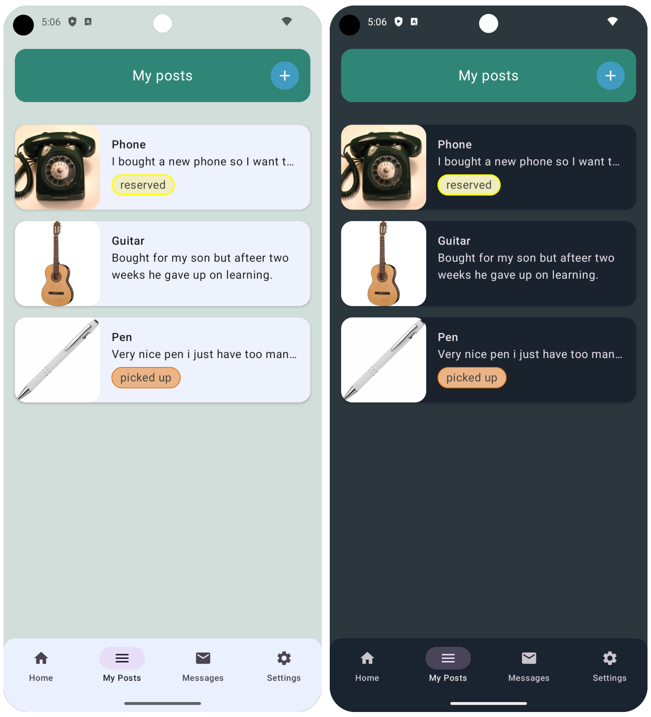


Figure B.6: Send message dialog

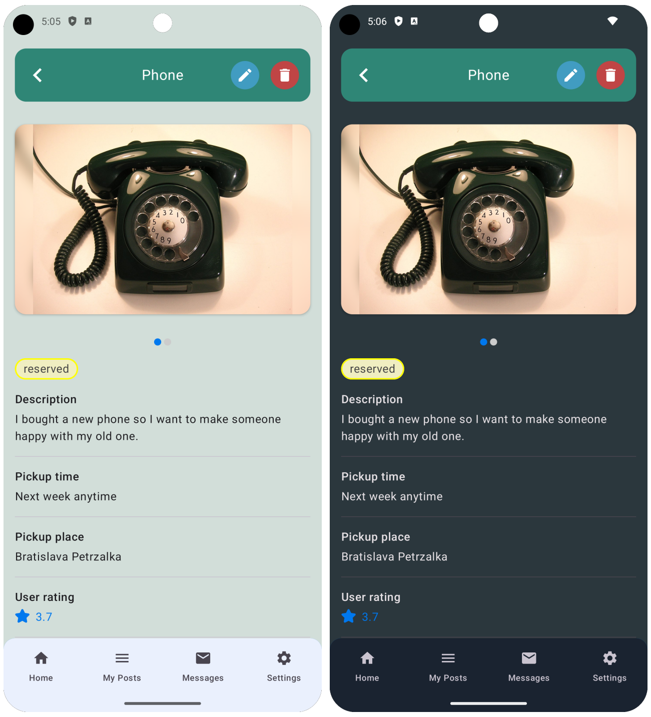


Phone
I bought a new phone so I want t...

Guitar
Bought for my son but afteer two weeks he gave up on learning.

Pen
Very nice pen i just have too man...

Figure B.7: My posts screen


5:05

Phone

reserved

Description
I bought a new phone so I want to make someone happy with my old one.

Pickup time
Next week anytime

Pickup place
Bratislava Petrzalka

User rating
3.7

5:06

Phone

reserved

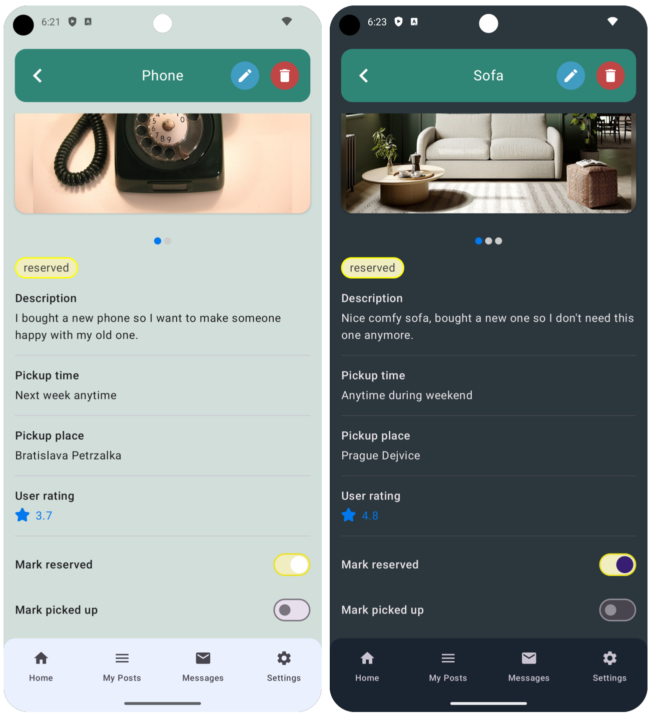
Description
I bought a new phone so I want to make someone happy with my old one.

Pickup time
Next week anytime

Pickup place
Bratislava Petrzalka

User rating
3.7

Figure B.8: My post detail screen


6:21

Phone

reserved

Description
I bought a new phone so I want to make someone happy with my old one.

Pickup time
Next week anytime

Pickup place
Bratislava Petrzalka

User rating
3.7

Mark reserved

Mark picked up

6:23

Sofa

reserved

Description
Nice comfy sofa, bought a new one so I don't need this one anymore.

Pickup time
Anytime during weekend

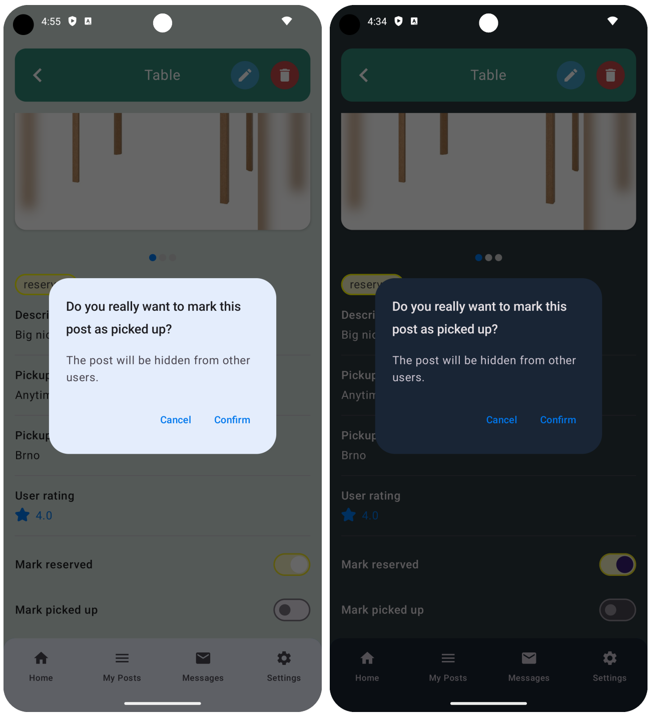
Pickup place
Prague Dejvice

User rating
4.8

Mark reserved

Mark picked up

Figure B.9: My post detail screen



Do you really want to mark this post as picked up?

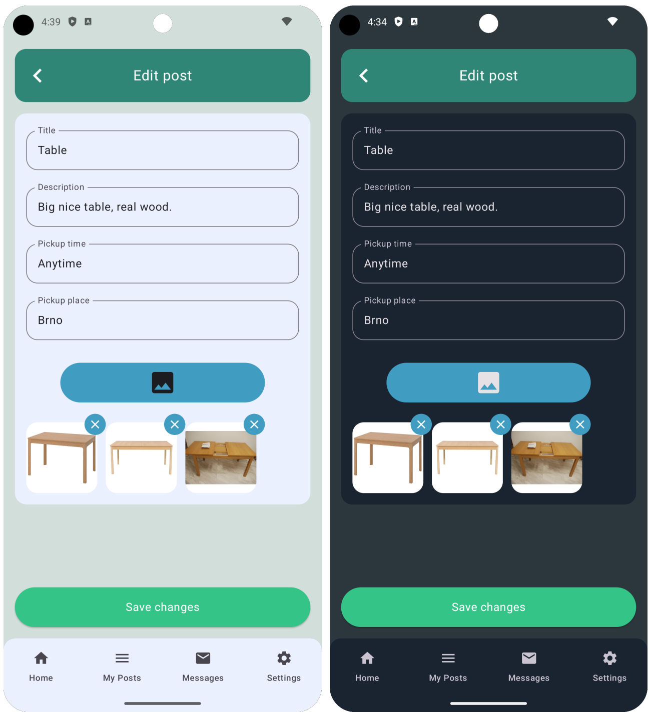
The post will be hidden from other users.

User rating
4.0

Mark reserved

Mark picked up

Figure B.10: Pickup mark confirmation dialog



Title
Table

Description
Big nice table, real wood.

Pickup time
Anytime

Pickup place
Brno

Figure B.11: Edit post screen

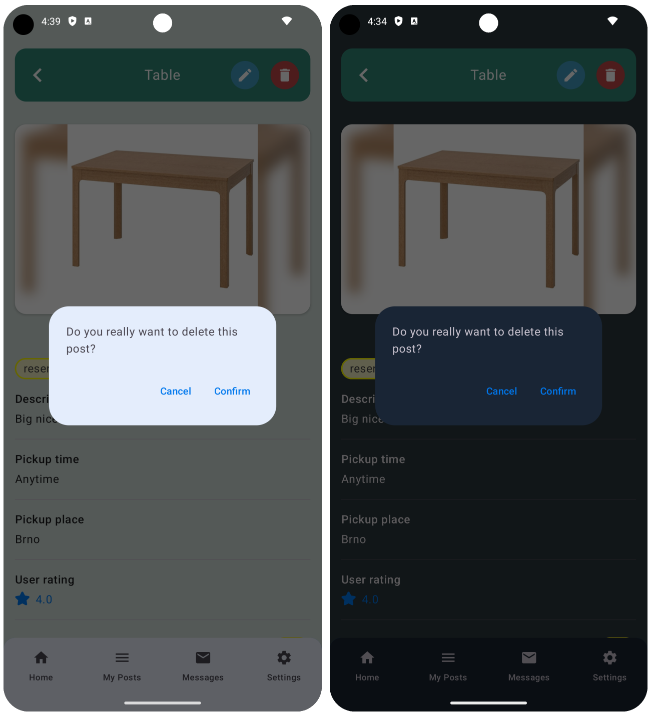


Figure B.12: Delete post dialog

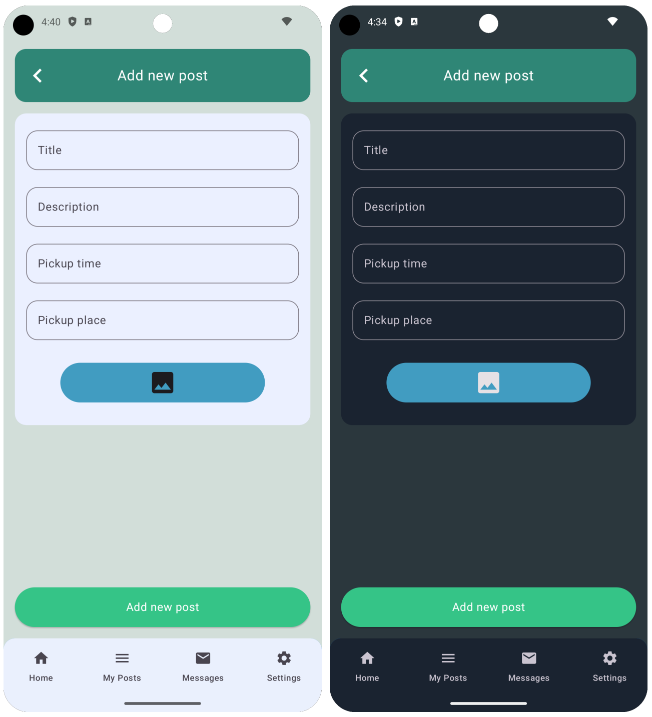
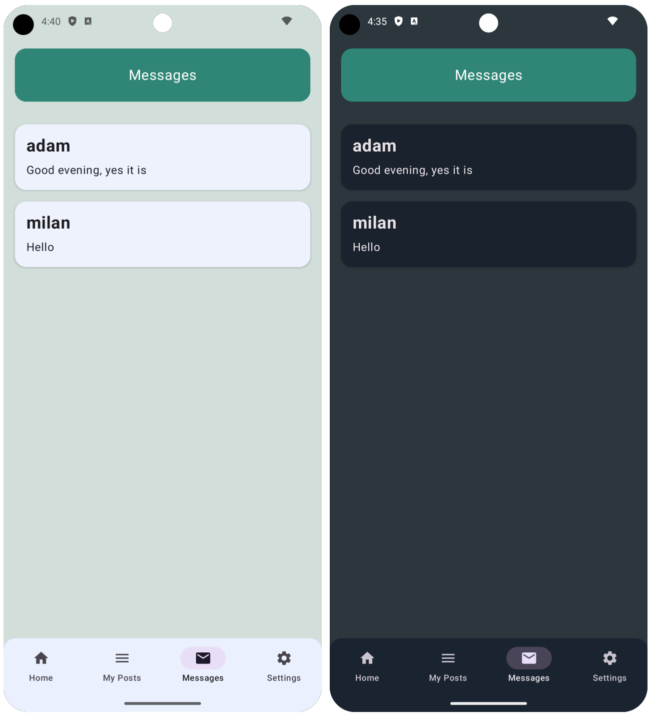


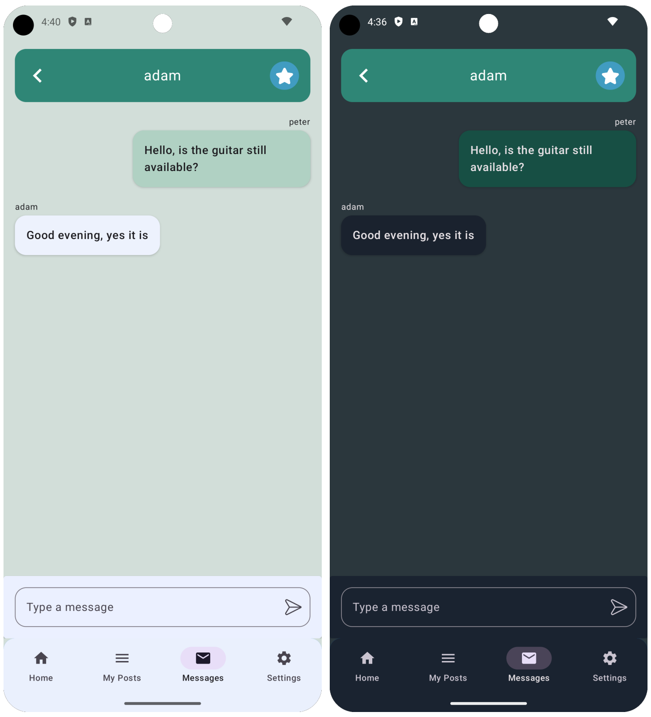
Figure B.13: Add new post screen


Messages

adam
Good evening, yes it is

milan
Hello

Figure B.14: Message screen

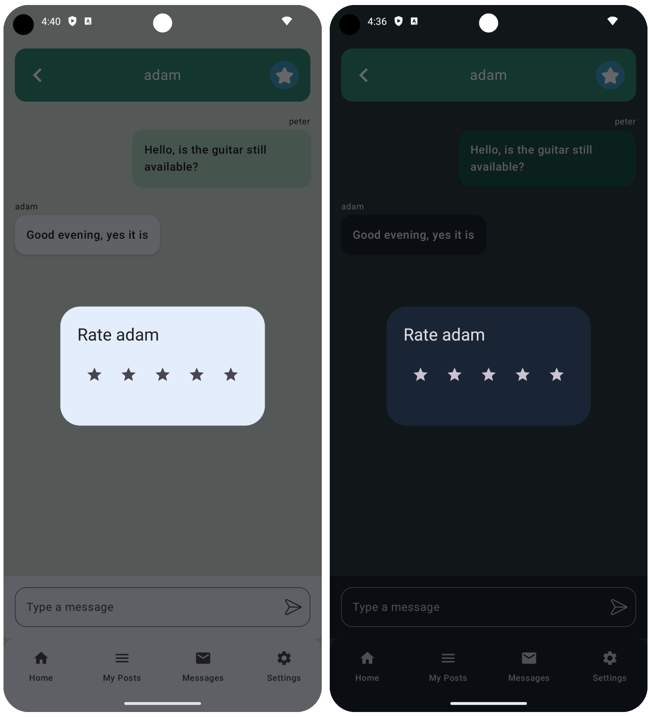

adam

peter
Hello, is the guitar still available?

adam
Good evening, yes it is

Type a message

Figure B.15: Message detail screen



4:40

adam

peter

Hello, is the guitar still available?

adam

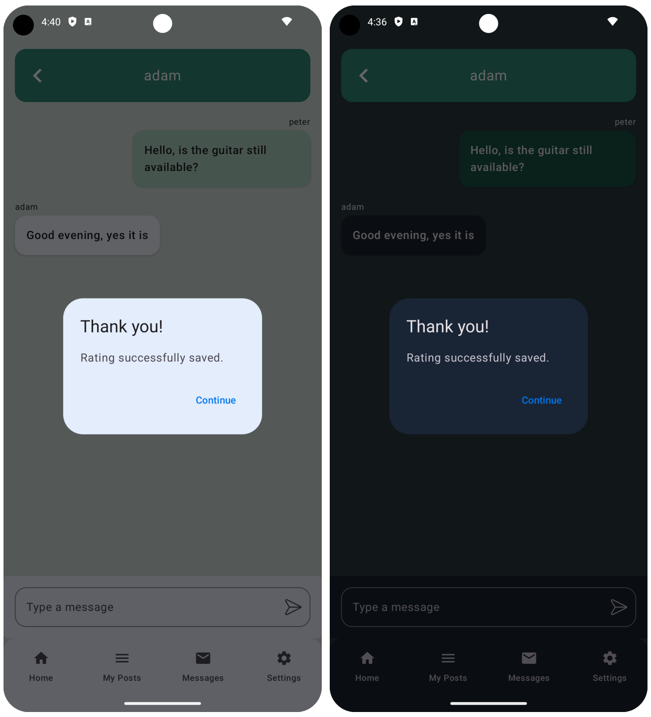
Good evening, yes it is

Rate adam

★ ★ ★ ★ ★

Type a message

Figure B.16: Rate user dialog



4:40

adam

peter

Hello, is the guitar still available?

adam

Good evening, yes it is

Thank you!

Rating successfully saved.

Continue

Type a message

Figure B.17: Rating successful dialog

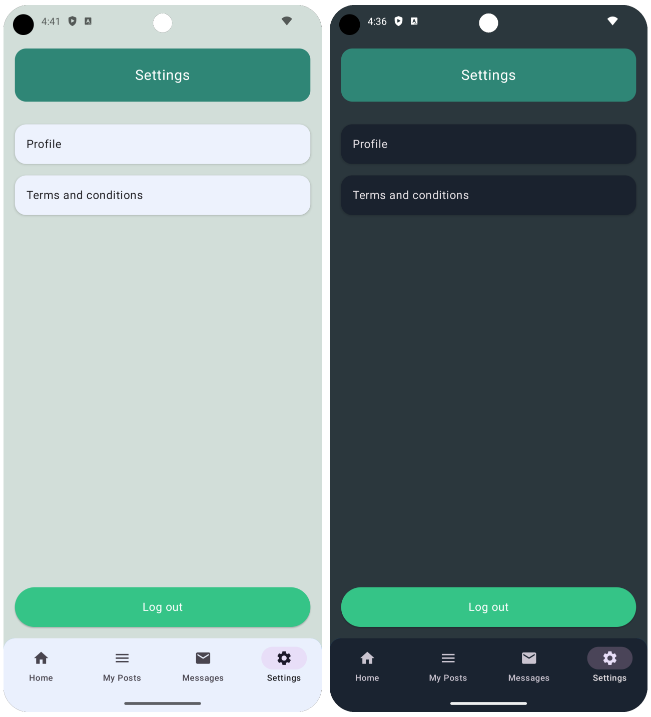
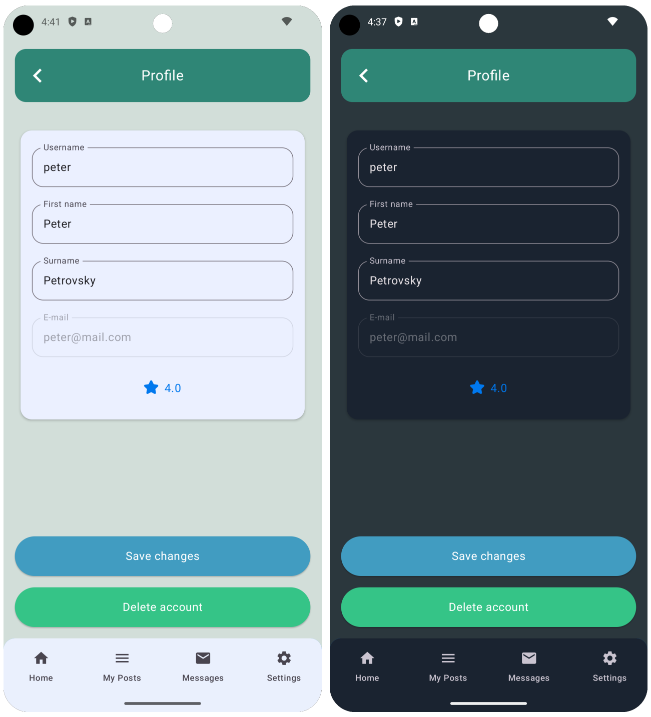


Figure B.18: Settings screen



Username: peter
First name: Peter
Surname: Petrovsky
E-mail: peter@mail.com
4.0

Username: peter
First name: Peter
Surname: Petrovsky
E-mail: peter@mail.com
4.0

Figure B.19: Profile screen

Appendix **C**

Testing scenarios

C.1 Registration and Logging in

For this testing scenario, the tester needs to have a valid e-mail address.

Steps:

1. Launch the application and wait for the *Login* screen to appear.
2. Click on the *Register* button.
3. Enter the required information into the fields.
4. Click the *Create an account* button.
5. If any fields are incomplete or contain errors, an error notification will appear. Correct any issues and click the button again to proceed.
6. Once registration is successful, a confirmation screen will display, featuring a *Continue* button. Click this button and wait to be redirected back to the login screen.
7. Enter e-mail and password that was used in registration.
8. Click the *Log in* button and wait for the home screen to load.

C.2 Send a message

For this test scenario, the tester must be logged in and located on the *Home* screen.

Steps:

1. Select any post and click on it to view its detail.
2. Once the post detail is fully loaded, scroll to the bottom of the screen and click on the *Contact user* button.
3. When the dialog appears, type your message into the provided field.
4. Click the *Send* button to dispatch the message.

5. Once the message has been successfully sent, a dialog will appear displaying a success message along with a *Continue* button. Click on this button to proceed.
6. Click the *Back* button to return to the *Home* screen.
7. Click the *Messages* tab from the bottom navigation bar.
8. After the screen is loaded verify that the message you sent appears at the top of the conversations list.
9. Open the top conversation and check if the initial message you sent is displayed.
10. Type a new message in the field.
11. Click on the *Send* button.
12. Confirm that the new message appears at the end of the conversation history.

C.3 Rate user

For this test scenario, the tester must be logged in and located on the *Home* screen.

Steps:

1. Click the *Messages* tab from the bottom navigation bar.
2. After the screen is loaded open the conversation on the top of the list.
3. Locate and click on the button with a star symbol in the header of the screen. Wait for the rating dialog to appear.
4. Select the last star in the row to give the highest rating, and then wait for a success dialog to confirm the rating has been successfully submitted.
5. Click on the *Continue* button and verify that you are redirected back to the conversation detail screen.

C.4 Add new post

For this test scenario, the tester must be logged in and located on the *Home* screen.

Steps:

1. Tap on the *My Posts* tab located in the bottom navigation bar and wait for the posts to load.
2. Select the first post from the list and wait for the post detail to load.
3. Locate the button with a plus symbol in the header of the screen and click on it, then wait to be redirected to the *Add New Post* screen.

4. Enter “title” in the *Title* field.
5. Type “description” into the *Description* field.
6. Fill in “pickup time” in the *Pickup Time* field.
7. Input “pickup place” in the *Pickup Place* field.
8. Select at least one picture from the phone’s library to include with the post.
9. After ensuring all information is entered, click the *Add post* button. If any fields are left incomplete or no picture has been added, an alert will notify you to provide the missing information. Once all fields are correctly filled, you will be automatically redirected back to the *My Posts* screen.

C.5 Edit post

For this test scenario, the tester must be logged in and located on the *Home* screen.

Steps:

1. Tap on the *My Posts* tab in the bottom navigation bar and wait for the posts to load.
2. Select the first post from the list and wait for the post detail screen to load.
3. Locate the button with an edit symbol in the screen’s header. Click on this button and wait to be redirected to the *Edit Post* screen.
4. OOnce the *Edit Post* screen is displayed, update the title of the post to “Updated Title” and the pickup time to “Today”.
5. Delete the first picture listed in the post.
6. AAdd a new picture by clicking on the button marked with an image symbol.
7. After updating the information and managing the images, click on the *Save changes* button. Wait to be redirected back to the post detail screen. Before saving, make sure all fields are correctly filled. If any errors occur, such as missing mandatory information or an upload issue, verify that the system displays an error message.
8. On the post detail screen, click the *Reserved* toggle button.
9. Verify that the *Reserved* mark appears below the image.
10. Click the *Picked up* toggle button.
11. Check that the *Picked up* mark is displayed below the image, replacing the *Reserved* mark.
12. Confirm that the *Delete* and *Edit* buttons are no longer present in the header.

C.6 Delete post

For this test scenario, the tester must be logged in and located on the *Home* screen.

Steps:

1. Click on the *My Posts* tab in the bottom navigation bar and wait for the posts to load.
2. Select the first post from the top of the list and wait for the post detail screen to load.
3. In the header of the screen, locate the button marked with a delete symbol. Click on this button and wait to be redirected back to the *My Posts* screen.
4. If an error occurs during the deletion process, verify that an error message is displayed.

C.7 Edit profile

For this test scenario, the tester must be logged in and located on the *Home* screen.

Steps:

1. Tap on the *Settings* tab in the bottom navigation bar and wait for the *Settings* screen to load.
2. Click on the *Profile* button and wait for the *Profile* screen to load.
3. Update the *Username* field to "changedusername".
4. Change the *First Name* to "changedname".
5. Modify the *Surname* to "changedsurname".
6. Click on the *Save changes* button. Upon successful update, a success message should be displayed. If an error occurs, ensure that an error message is shown.
7. Navigate back to the *Settings* screen using the button in the top left corner.
8. Click on the *Profile* button again and wait for the Profile screen to load.
9. Confirm that the changes have been saved by verifying that the *Username* is "changedusername", *First Name* is "changedname", and *Surname* is "changedsurname".

C.8 Delete profile

For this test scenario, the tester must be logged in and located on the *Home* screen.

Steps:

1. Click on the *Settings* tab in the bottom navigation bar and wait for the *Settings* screen to load.
2. Select the ”Profile” button and wait for the *Profile* screen to load.
3. Click on the *Delete account* button. Upon successful deletion, confirm that the user is redirected to the *Login* screen. If an error occurs, verify that an error message is displayed.

C.9 Log out

For this test scenario, the tester must be logged in and located on the *Home* screen.

Steps:

1. Click on the *Settings* tab in the bottom navigation bar and wait for the *Settings* screen to load.
2. Click on the *Log out* button and wait to be redirected to the *Login* screen. If any errors occur during this process, ensure that an error message is displayed.

Appendix D

Contents of attachments

```
 readme.txt ........................ the file with CD contents description
└─ src....................................... the directory of source codes
   ├─ mobile_app ................................ implementation sources
   └─ thesis............... the directory of LaTeX source codes of the thesis
└─ text .......................................... the thesis text directory
   └─ thesis.pdf........................... the thesis text in PDF format
└─ extra............................................ the extra attachments
   ├─ screenshots ....................... the screenshots of the application
   └─ videos ................................ the videos of the application
```