Bachelor Thesis
Documentation

OST Marketplace App

for Android and iOS using Kotlin Multiplatform (KMP)

Semester: Spring 2025

Project Team: Roger Marty
Simon Peier
Tseten Emjee

Project Advisor: Martin Seelhofer

Version: 1.0
Date: 2025-06-13

School of Computer Science
OST Eastern Switzerland University of Applied Sciences

Part I

Abstract

Abstract

Whether students want to reduce clutter, generate extra income or contribute to a more sustainable economy, having an accessible way to sell and buy secondhand or new items can be beneficial. Unfortunately, existing marketplaces are not specifically designed for an academic environment. Their users are spread out and trust is limited, which discourages users from using them. To foster the usage of such marketplaces and combat the mentioned issues, a university-specific marketplace app should be created.

For the frontend, a cross-platform mobile app was developed. A major part of this thesis was the research and evaluation of Kotlin Multiplatform, which was used to develop the app. This provided both theoretical and practical insights into the technology. Various multiplatform-compatible frameworks and libraries such as Compose Multiplatform, Voyager and Koin were utilized. The backend is a REST API built with Python using FastAPI. SQLAlchemy is used as the ORM to interact with a PostgreSQL database. The entire backend, including a search engine, is hosted on AWS, leveraging services such as ECS Fargate, RDS, EC2, and S3. Firebase was used for push notifications, PubNub for real-time chat and Microsoft Entra ID for authentication.

The Kotlin Multiplatform research has concluded that the technology is sufficiently advanced to be used in productive applications. While still young in comparison to its competitors, JetBrains' support for its technology is evident and its progress is rapid. As of May 6, 2025, the iOS platform has also been marked as stable, meaning that cross-platform mobile development is now fully stable in Kotlin Multiplatform. The developed app, POSTE, is a fully functional marketplace where users can create listings, browse listings, initiate a chat with the seller and much more. OST students can sign in using their OST-specific Microsoft account. This not only enhances the trust among users but also gives a sense of community.

Part II

Management Summary

Management Summary

Initial Situation

In university settings, students and staff may own used and no longer needed items such as calculators, tablets, laptops or everyday commodities like clothes or bicycles. Instead of having these items just lying around or throwing them away, owners could resell or exchange them to reduce waste and make some money with it. While online marketplaces exist for this purpose, they are not tailored to universities. A marketplace app specifically made for the OST domain would fill this gap. It could encourage users to use such a platform more frequently, as they are geographically closer. And since the transactions are only made with other students of the same university, the trust factor would be increased which leads to more students using such a platform. Additionally, no new account would be required, as users could log in using their existing OST Microsoft account.

The idea of a university-specific marketplace is coupled with the second objective of this thesis, which is the research and evaluation of Kotlin Multiplatform (KMP). To make the app available to the widest possible audience, it should run on both Android and iOS. Kotlin Multiplatform is a relatively new cross-platform development framework that allows the sharing of both business and UI logic. The evaluation of its maturity level and real-world suitability is a central part of this thesis.

Methodology

The project was conducted using the Scrum+ methodology, which uses RUP for long-term planning and Scrum for short-term planning. In addition, risks were identified and evaluated during the inception phase, followed by the preparation of quality measures.

The elaboration phase began by defining the functional and non-functional requirements of the Minimum Viable Product (MVP). This included core features such as listing creation, browsing the marketplace and chat functionality.

Following this, extensive research into Kotlin Multiplatform was conducted to assess its capabilities and production-readiness. Additional research into real-time messaging options, authentication, search engines, cloud services and App Store deployment processes was carried out.

After evaluating said technologies and choosing the options best suited for the project, the application architecture was designed. For this, wide-spread software engineering methodologies such as the C4 model, domain analysis and component interaction diagrams were leveraged.

As a last step before the implementation phase began, wireframes of the app were designed and the required cloud infrastructure was set up and tested.

During the implementation phase, the mobile app and backend were developed in parallel. A beta and and an MVP release were performed, followed by usability tests with students to validate the app's functionality and gather real-world feedback.

Technologies

Frontend: KMP, Compose Multiplatform, Voyager, Koin

The app is built using KMP, leveraging compatible frameworks and libraries such as Compose Multiplatform for the user interface, Voyager for navigation, and Koin for dependency injection. This approach enables the sharing of business and UI logic, while keeping platform specific code to a minimum.

Backend: Python, FastAPI, SQLAlchemy

A REST API developed in Python, using the FastAPI framework, serves as the backbone of the application. It provides the required endpoints for the frontend to interact with, handling all core functionalities. It communicates with the database and additional services, such as the search engine and AWS components.

Cloud: AWS, OpenTofu

AWS is used to host the backend, database, and image storage. ECS Fargate runs the containerized backend, RDS hosts the PostgreSQL database and S3 stores the listing images. An EC2 instance runs the Meilisearch search engine required for the browsing of listings. Other services, such as load balancing, are integrated as well. Everything is configured using infrastructure as code (IaC) with OpenTofu.

External Interfaces: Microsoft Entra ID, PubNub, Firebase

To avoid implementing authentication from scratch, Microsoft Entra ID is used as the identity provider. This allows OST students and employees to sign in using their OST Microsoft account. Real-time chat is an integral part of the app, allowing users to negotiate and discuss payment or delivery details of the item. PubNub is used to provide this functionality. To send push notifications and handle events in real-time, Firebase Cloud Messaging is integrated.

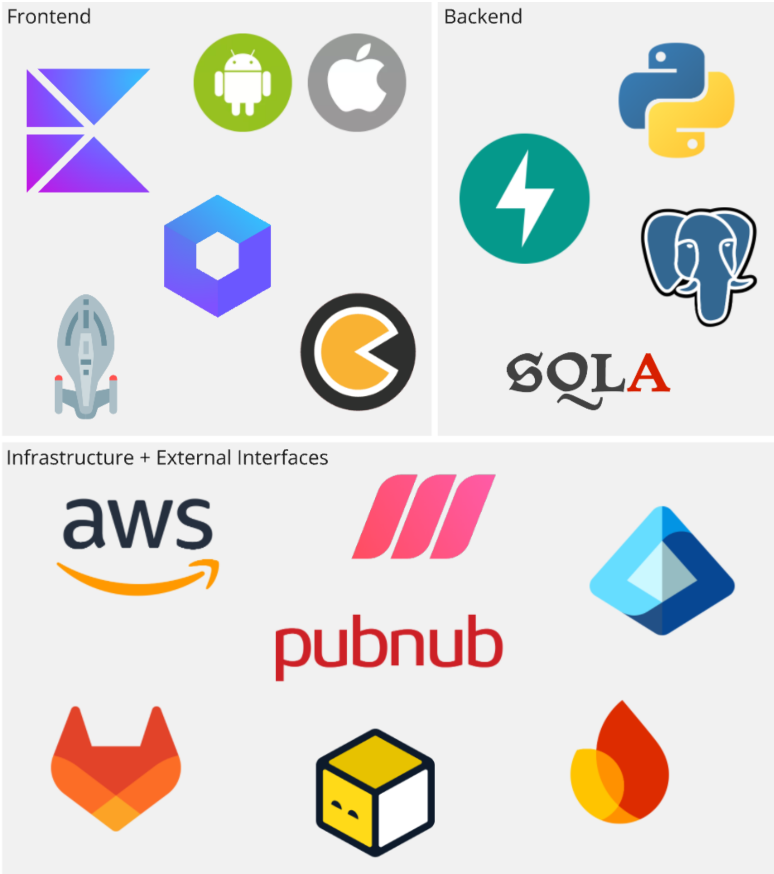


Figure 1: Used Technologies

Results

Product

In the final version of the POSTE app, everyone with a valid OST Microsoft account can sign in and start using the marketplace. The app itself is available on the Apple App Store.

As a seller, the user can create listings, attach images to them and provide additional information such as price, condition, category and a description. These listings can be updated or deleted as needed. The seller can also view and respond to incoming chat messages from interested buyers.

As a buyer, the user can browse listings using a variety of filtering and sorting options. If not looking for anything specific, the popular listings may be of interest. If an interesting item is found, it can either be saved to the personal watchlist or a chat with the seller can be initiated.

In the chat, the seller and buyer can discuss details such as pricing, payment and shipping methods. Once the payment has been sent, the seller can mark it as received. Later, upon receival of the item, the buyer can confirm the delivery. Once both parties confirm their part, the transaction is finalized and they have the opportunity to rate each other.

Other features such as a blocklist, automatic QR-bill generation and various settings like dark mode are also available.

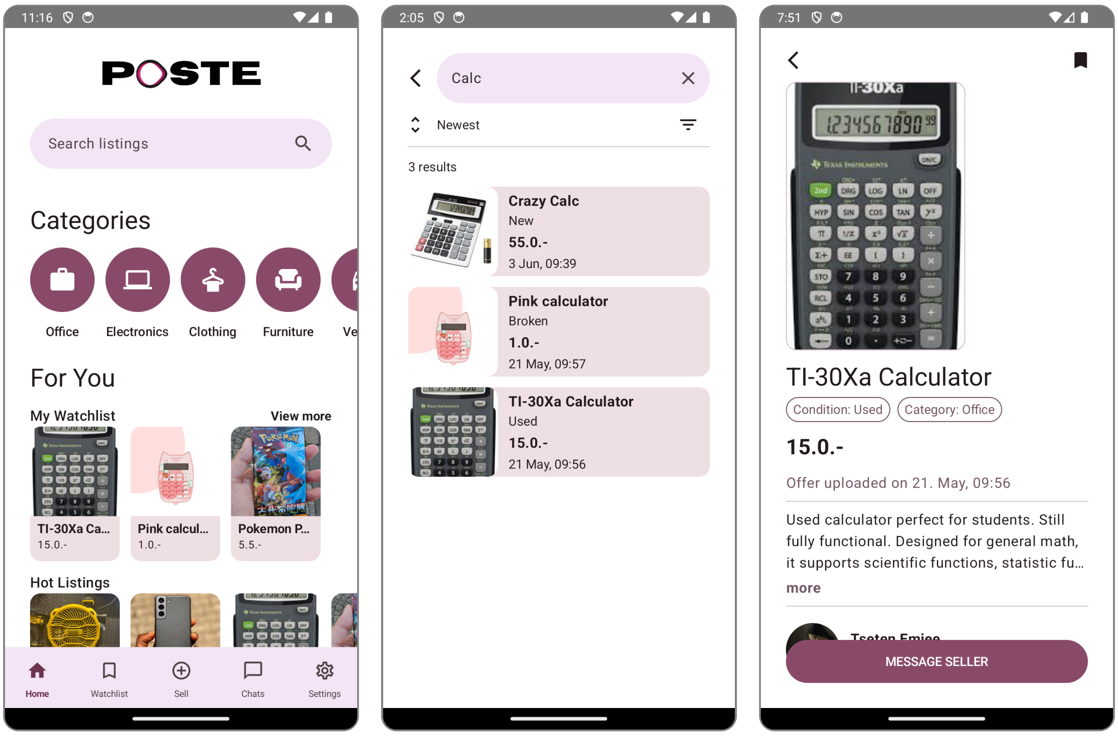


Figure 2: MVP App - Home Screen, Search and Listing

Kotlin Multiplatform

The result of the KMP research was a positive assessment of the technology. KMP is already used by international companies such as McDonald's and Forbes, providing millions of users with their services. These companies use KMP to separate their native applications from their business logic, which they develop as a shared KMP module. This approach increases the maintanability of the business logic and helps reduce the lines of code. Any changes to the business logic need to be made only once within the KMP module.

The POSTE app took a different approach, sharing 100% of its code, meaning the UI code as well. This significantly reduced development time, having to write the UI code only once using the Compose Multiplatform framework. Since this framework is based on Jetpack Compose, which the development team is highly experienced with, it also contributed to faster development. The development of POSTE was not without issues, but overall the POSTE development team deems the Kotlin Multiplatform technology as a viable alternative for productive mobile cross-platform development.

Outlook

While all planned features were implemented, there is still room for improvements and additional functionality. Several optional features were considered and could be added in the future.

One such feature is an auction system, where multiple buyers could bid on a listing. This could increase the engagement and potentially lead to better prices for in-demand items, which benefits the seller.

Another feature would be the ability to save specific searches, and receive alerts when new listings, matching a saved search, are created.

Finally, the reporting and monitoring capabilities of the platform could be improved. Users have the ability to block other users but having an option to report them would enhance safety and trust on the platform.

Contents

Part III

Glossary and Acronyms

Glossary

Table 1: Glossary

Term	Definition
AMI	An Amazon Machine Image that already contains the required software for booting up an EC2 machine.
C4 Diagram	The Context, Container, Component, and Code diagram used for software architecture modeling.
EC2	Elastic Compute Cloud are virtual servers in the cloud that can be used for various tasks (e.g. hosting the Meilisearch search engine)
ECR	A fully managed Docker container registry service called Amazon Elastic Container Registry, used to store and manage container images.
ECS Fargate	A serverless compute engine provided by Amazon Elastic Container Service, used to run containers without managing servers. ECS Fargate handles scaling and orchestration.
VPC	To isolate resources in the cloud a Virtual Private Cloud can be used.

Acronyms

Table 2: Acronyms

Term	Definition
AAB	Android App Bundle
ALB	Application Load Balancer
APK	Android Package Kit
AWS	Amazon Web Services
CI/CD	Continuous Integration and Continuous Delivery/Deployment
CMP	Compose Multiplatform
DI	Dependency Injection
DTO	Data Transfer Object
ERD	Entity-Relationship Diagram
FCM	Firebase Cloud Messaging
IaC	Infrastructure as Code
IPA	iOS App Store Package
JDK	Java Development Kit
JWT	JSON Web Token
JVM	Java Virtual Machine
KMP	Kotlin Multiplatform
MADR	Markdown Architectural Decision Records
NAT	Network Address Translation
NDK	Native Development Kit
ORM	Object-Relational Mapping
OST	OST Eastern Switzerland University of Applied Sciences
PII	Personally Identifiable Information
PSP	Payment Service Provider
RDS	Relational Database Service
S3	Simple Storage Service

Part IV

Introduction

Chapter 1

Task Definition

This documentation outlines the development of the OST marketplace app, named POSTE, as well as the research of the Kotlin Multiplatform (KMP) technology. This is part of the Bachelor thesis in the Computer Science program at OST - Eastern Switzerland University of Applied Sciences. A cross-platform mobile application is developed to facilitate the exchange of secondhand or unused items within the university community. The university community includes OST students and employees. If only one party is mentioned, students and employees are included. The development of this app also serves as a practical evaluation of KMP.

To achieve this, KMP is thoroughly researched, evaluated and documented. The KMP research, while part of the thesis, is presented as a self-contained segment and is available as a separate document. In addition, a cloud-based backend is designed and developed to support the required functionality in a scalable and highly available manner. If possible, a payment provider is integrated and if not, at a conceptual design must be developed.

To ensure the app meets the user needs, functional and non-functional requirements are first defined and then discussed with the project supervisor. Further details regarding the specifications of the thesis can be found in the original task description located in Appendix A.

Chapter 2

Motivation

This Bachelor thesis is conducted by three computer science students: Tseten Emjee, Roger Marty, and Simon Peier who came up with the idea behind the OST marketplace app. The core motivation is to create a product exchange platform where the users feel safe and familiar, as all participants are part of the same university community. Whereas any user can sign up on other marketplaces, POSTE restricts sign-ups to students with access to a valid OST Microsoft account. This closed community is intended to foster trust and reduce the risk of malicious buyers or sellers.

Additionally, the project covers areas of interest of all three students with Simon and Tseten being primarily interested in app development and Roger in the cloud technologies and backend development. It is an excellent opportunity to apply and deepen the knowledge obtained during the course of the computer science studies, acquire new skills and gain practical experience.

Chapter 3

General Conditions

The project is conducted in the spring semester of 2025 and spans a total duration of 16 weeks. The Bachelor thesis is worth 12 ECTS credits. With one credit equaling to 30 hours, the whole workload amounts to 360 hours per student which is roughly 22.5 hours per week. In total, 1'080 hours are available for the completion of the project.

A meeting with the supervisor, Martin Seelhofer, is held on a weekly basis to review progress and address challenges. Important discussions and decisions are documented accordingly and can be found in the meetings minutes located in Appendix D.

An interim and final presentation must be held in the presence of the supervisor, the assigned proofreader and an external expert. These stakeholders are also responsible for evaluating the overall quality and outcome of the thesis.

Part V

Product Documentation

Chapter 4

Requirements

This chapter defines the requirements of the project, including both functional and non-functional aspects. These requirements are defined based on the task specifications outlined in Appendix A to ensure a high-quality product.

4.1 Functional Requirements

This section covers the functional requirements for the marketplace application. The relevant actors and their goals, as well as the use cases are shown.

4.1.1 Actors

Actor	**Goals**
Seller	A person with the intent of selling used or unneeded items. The main goal is to create and manage listings.
Buyer	The buyer's main focus is to search for listings and start the interaction with the seller if interested.
AWS	AWS's goal is the reliable hosting of the backend, database, search engine and object storage.
PubNub	Focuses on real-time and reliable message delivery between buyer and seller which allows for negotiations and the exchange of payment and delivery details.
Firebase	Firebase manages the sending of push notifications.

Table 4.1: Actors

4.1.2 Use Case Diagram

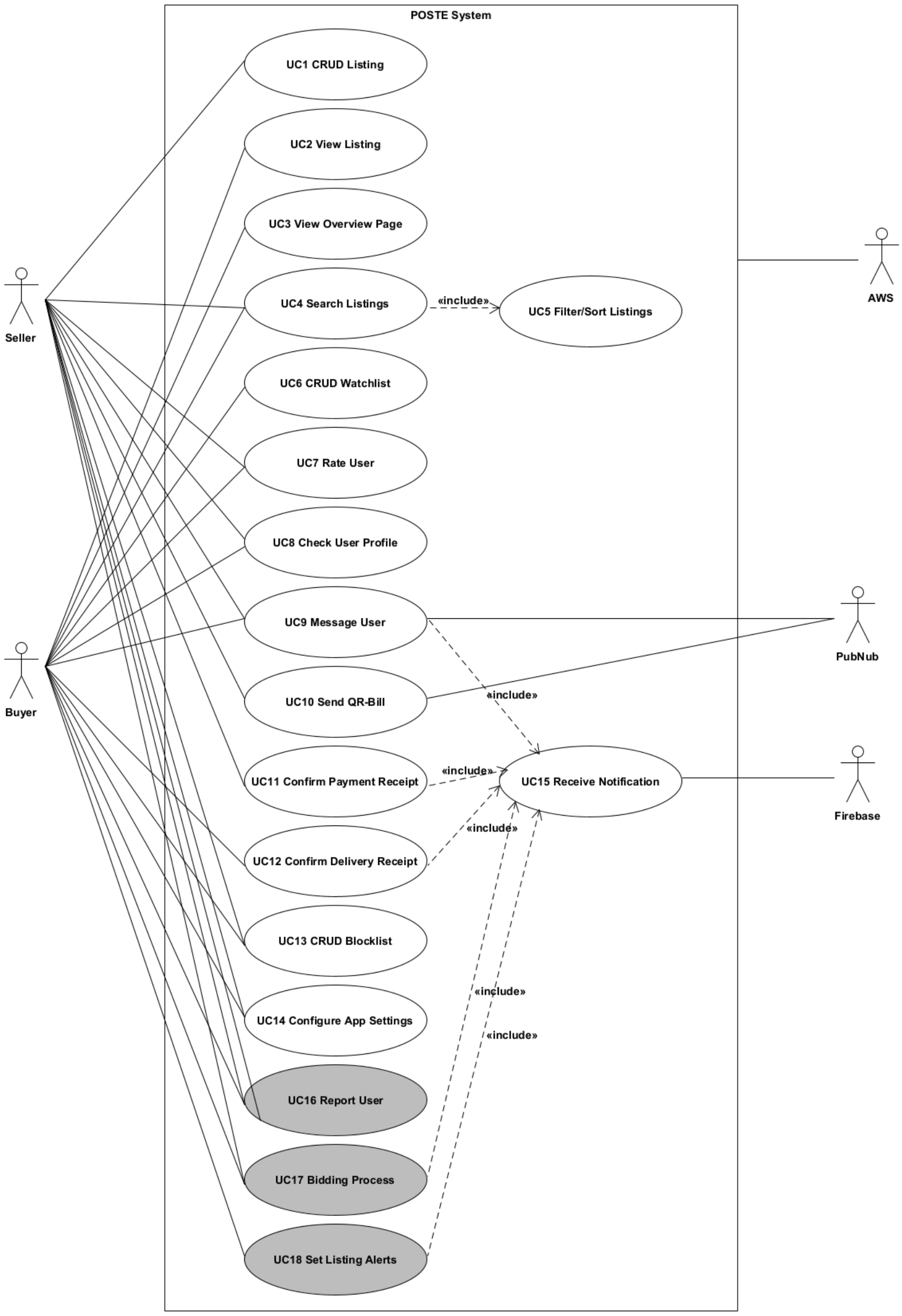


Figure 4.1: Use Case Diagram

4.1.3 Use Case Descriptions

Each use case is described below using the casual format [1].

Table 4.2: Casual Format Use Cases

ID	Description	Alternate Scenario
UC1	The seller can create listings for products intended for sale. Listings can be viewed, updated or deleted as desired.	-
UC2	A buyer can open listings of interest to see more details about them.	-
UC3	When the app is opened, an overview page is shown to the user with shortcuts to certain functionalities (e.g. search, watchlist).	-
UC4	All users can search for listings using the search function.	No results: A hint is shown to the user.
UC5	The searches can be refined using the filter and sort functions.	No results: A hint is shown to the user.
UC6	Buyers have the ability to wishlist their favorite listings so they can easily be accessed at a later time. The wishlist can be viewed and modified as needed.	-
UC7	After a purchase/sale, the buyer and seller can rate each other on a scale from 1 to 5.	-
UC8	Through the listing, a user can open the profile of the other party to see its rating and listings.	-
UC9	If a buyer is interested in a listing, they can start a conversation with the seller using the chat function.	Blocked user: If the seller has blocked the buyer, the buyer cannot start a conversation. Own listing: The message seller button is disabled if the listing is their own.

Continued on next page

Table 4.2 – continued from previous page

ID	Description	Alternate Scenario
UC10	The seller has the option to send an automatically generated QR-bill to the buyer in the chat window.	QR-bill not enabled: A click on the button shows a reminder to enable the QR-bill in the settings.
UC11	After payment details are exchanged and the buyer has sent the money, the seller can confirm the receipt of the payment to indicate that the product will be shipped soon.	-
UC12	The buyer can confirm that the product has been received.	-
UC13	Other users on the platform can be blocked and unblocked.	-
UC14	There is a settings tab to customize the app experience which includes features like dark mode and notification settings.	-
UC15	For important events the user receives a push notification.	Push notifications turned off: The user is not notified.
UC16	For proper marketplace moderation, users can be reported which informs the relevant parties.	-
UC17	Buyers can bid on listings. The highest bidder at the end of the deadline can get in touch with the seller.	-
UC18	Certain searches can be saved and if a new listing that matches the saved search appears, a notification is sent to the user.	-

Precondition

The user must sign in using a valid OST Microsoft account to perform any actions within the app. Only authorized OST students and staff can interact with the marketplace functionalities.

4.2 Non-Functional Requirements

Based on the ISO/IEC 25010 [2] standard, the below non-functional requirements are defined.

4.2.1 Performance Efficiency

ID	NFR1
Description	The search function returns results quickly.
Acceptance Criteria	Searching for a product returns results according to the specified landing zones, given the user has a mid-range device with a working 4G internet connection.
Landing Zones	• Minimal: 4 seconds • Target: 2 seconds • Outstanding: 1 seconds

Table 4.3: NFR1

ID	NFR2
Description	The backend is able to process many search requests in a given timeframe.
Acceptance Criteria	The backend returns successful responses for search results with the default pagination size of 10, as specified.
Landing Zones	• Minimal: 50 requests per minute • Target: 100 requests per minute • Outstanding: 150 requests per minute

Table 4.4: NFR2

ID	NFR3
Description	The capacity of the database is sufficient for the target audience.
Acceptance Criteria	The database can handle 120'000 listings (6000 possible users * 20 listings per user).

Table 4.5: NFR3

4.2.2 Reliability

ID	NFR4
Description	Fault tolerance is implemented by the backend server.
Acceptance Criteria	In case of a downed node, the traffic gets load-balanced to a working node and operability is maintained.

Table 4.6: NFR4

ID	NFR5
Description	Database data backups are always available.
Acceptance Criteria	All databases are backed up daily and can always be restored.

Table 4.7: NFR5

ID	NFR6
Description	No app crashes due to user input.
Acceptance Criteria	During usability and regression testing the app does not crash due to user input.

Table 4.8: NFR6

4.2.3 Maintainability

ID	NFR7
Description	The project is efficiently modifiable.
Acceptance Criteria	Industry standard code guidelines are enforced with linters, formatters and pipelines to ensure quality.

Table 4.9: NFR7

4.2.4 Security

ID	NFR8
Description	Data is only transferred over secure connections and is encrypted during transit and at rest.
Acceptance Criteria	Only HTTPS connections are used for data transfer between the app, AWS and other external services. PII in the database is never stored as plain text and is encrypted instead.

Table 4.10: NFR8

ID	NFR9
Description	A user may only see his own data.
Acceptance Criteria	Upon signing in, the user only has access to his own account and PII. This is tested by developers on an ad hoc basis.

Table 4.11: NFR9

4.2.5 Interaction Capability

ID	NFR10
Description	A user is aware when certain errors happen.
Acceptance Criteria	The user is informed with messages when connection or backend errors occur.

Table 4.12: NFR10

ID	NFR11
Description	The app demonstrates intuitive navigation and interaction, allowing users to accomplish their goals with minimal friction.
Acceptance Criteria	The feedback from the usability tests is positive.

Table 4.13: NFR11

4.2.6 Flexibility

ID	NFR12
Description	The app is installable on multiple mobile operating systems.
Acceptance Criteria	The app can be installed on Android and iOS.

Table 4.14: NFR12

Chapter 5

Domain Analysis

In this chapter the key identities and their relationships are illustrated using a domain model diagram. Certain core entities are explained in greater detail.

5.1 Domain Model

The following model shows the entities that are in the MVP and the ones that are outside the scope.

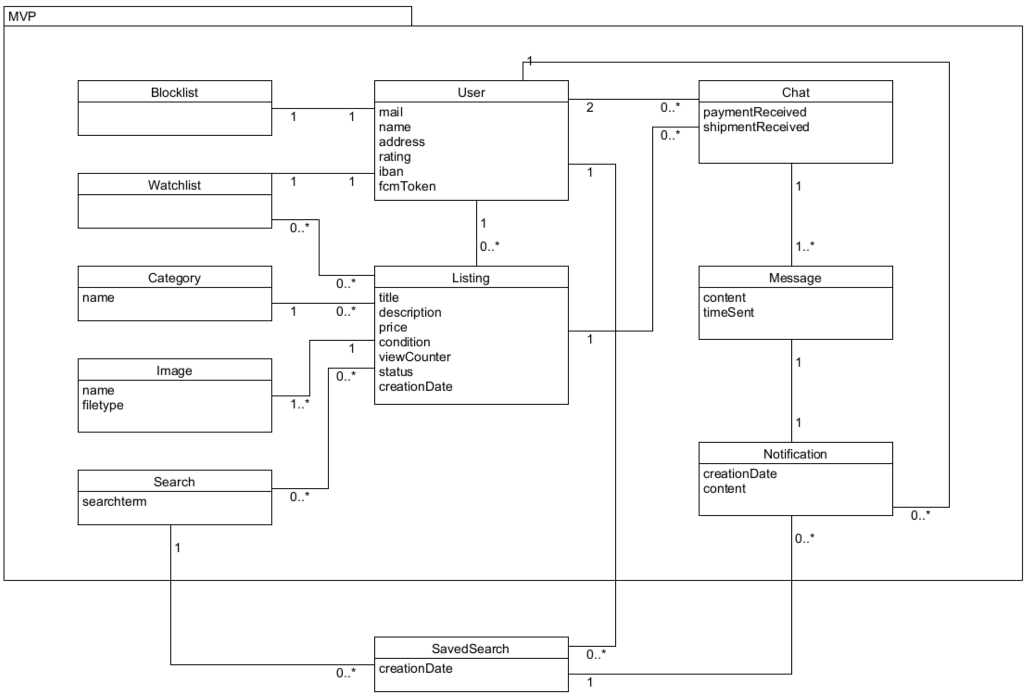


Figure 5.1: Domain Model Diagram

5.1.1 User

A user represents a registered OST student or employee in the system. The user can act as a seller or buyer depending on their actions, but in the system the user is treated the same way. Among other things, their most important abilities are creating listings and conversing with others to buy/sell things.

5.1.2 Listing

A listing contains details about the item that is being sold on the marketplace. This information is shown to buyers and can be used in searches. The listing itself also has a status field to indicate whether the product has been sold or if the offer is still open.

5.1.3 Chat & Message

These two entities are tightly coupled. A chat always consists of two users, unless one party leaves, and contains a series of messages. This allows for simple seller-buyer communication.

The status of the negotiations between sellers and buyers is bound to the chat entity.

5.1.4 Blocklist

If a user does not want to be contacted by certain users, they have the ability to block them. This can also be undone if desired.

5.1.5 Watchlist

The watchlist allows the buyer to mark certain listings. These can then be accessed via a separate tab in the navigation instead of having to search for the listing again.

5.1.6 SavedSearch

If a buyer is looking for a specific product, an alert can be created using certain search terms. A notification will then be sent to the buyer if a new listing is posted, that matches the saved search.

Chapter 6

Architecture

This chapter focuses on the architectural decisions and the system's design.

6.1 Technology Decisions

This section shows the evaluation of the technologies used in the project. Different options are compared against each other and documented using the MADR format [3].

No decision is required regarding the cross-platform development of the app, as the task definition mandates the use and evaluation of KMP.

6.1.1 Frontend Dependency Injection

Context and Problem Statement

To manage dependencies in the frontend application, a dependency injection framework is needed. It should integrate well with KMP and be officially supported.

Considered Options

- Koin
- Kotlin-inject

Decision Outcome

Chosen option: Koin, because:

- It works for Multiplatform and is officially referenced by Kotlin [4].
- It integrates well with Compose Multiplatform.
- It has more community support.

6.1.2 Frontend UI Framework

Context and Problem Statement

A UI framework is needed to build cross-platform UIs while enabling native-like experiences. Integration with KMP is required.

Considered Options

- Compose Multiplatform
- Native implementations using Jetpack Compose and SwiftUI

Decision Outcome

Chosen option: Compose Multiplatform, because:

- It allows for sharing UIs across multiple platforms.
- It seamlessly integrates with Kotlin Multiplatform shared logic.
- The team already has experience using Jetpack Compose, which Compose Multiplatform is based on.

6.1.3 Frontend Navigation Framework

Context and Problem Statement

A framework is needed enable cross-platform navigation.

Considered Options

- Inbuilt Compose Multiplatform Navigation
- Voyager
- Decompose

Decision Outcome

Chosen option: Voyager, because:

- Inbuilt Navigation framework is in alpha.[1]
- It enables cross-platform navigation in KMP.
- Offers an alternative to the experimental ViewModel API of Compose Multiplatform.[1]
- Pragmatic and easy to use API.

[1] As of May 6, 2025, the Navigation and ViewModel package have been marked Stable [5]

6.1.4 Backend API Framework

Context and Problem Statement

To create the backend endpoints, a fitting framework is needed. It should be lightweight, well-documented and support features such as automatic request validation and dependency injection.

KMP is not considered for the backend part of the project, as its evaluation is focused on the mobile app part. Instead, a backend framework is chosen based on the team's experience and suitability for the defined requirements.

Considered Options

- FastAPI
- Flask
- Django
- Express.js

Decision Outcome

Chosen option: FastAPI, because:

- It is modern, lightweight and widely used.
- It is a high-performance framework with many built-in features.
- It integrates seamlessly with Pydantic for request validation and data serialization.
- FastAPI has strong community support and high quality documentation.
- Some team members already have experience with it.

Consequence

Through this decision, the backend language is set to Python.

6.1.5 Backend ORM

Context and Problem Statement

A relational database is used for storing data, which the backend interacts with frequently. For these operations an ORM is needed that integrates well with FastAPI and PostgreSQL.

Considered Options

- SQLAlchemy
- SQLModel

Decision Outcome

Chosen option: SQLAlchemy, because:

- SQLAlchemy is more mature and flexible.
- It works well with Pydantic, which allows for clean separation between DB and API models.
- SQLModel is less flexible and generally less favored within the developer community.

6.1.6 Database

Context and Problem Statement

A relational database that integrates well with the chosen frameworks and platforms is required.

Considered Options

- PostgreSQL
- MySQL

Decision Outcome

Chosen option: PostgreSQL, because:

- It is supported by SQLAlchemy.
- It is available on the AWS RDS service.
- It is open-source, powerful, highly reliable and familiar to work with.

6.1.7 Infrastructure as Code

Context and Problem Statement

Instead of creating the required cloud resources by hand, an IaC tool should be used. With it, the infrastructure should be managed reproducibly and version controlled. It is required to have GitLab CI/CD integration.

Considered Options

- Terraform
- OpenTofu
- CloudFormation

Decision Outcome

Chosen option: OpenTofu, because:

- GitLab officially supports OpenTofu (Terraform deprecated).
- It is open-source and well supported.

6.1.8 Cloud and Infrastructure

Context and Problem Statement

For the hosting of the backend, storage, search engine and database, a cloud provider is needed.

Considered Options

- AWS
- Azure
- GCP

Decision Outcome

Chosen option: AWS, because:

- It provides all services required for the use cases.
- It is widely adopted and has detailed documentation.
- There is existing knowledge of AWS and a personal interest in using it.

6.1.9 Authentication

Context and Problem Statement

Instead of creating an authentication system, an existing platform should be used. This is more secure and allows for easier setup and integration into the frontend and backend.

Considered Options

- Switch edu-ID integration
- OST Microsoft Account integration
- Firebase Authentication
- Auth0

Decision Outcome

Chosen option: OST Microsoft Account integration, because:

- This automatically restricts the app to OST students as is intended.
- The entire authentication logic (including 2FA) is handled by Microsoft Entra ID.
- JWT tokens can then be used to authenticate for the backend API.

6.1.10 Push Notifications

Context and Problem Statement

To inform users about important events or changes, push notifications need to be sent. A service is required that supports all required platforms.

Considered Options

- Firebase Cloud Messaging

Decision Outcome

Chosen option: Firebase Cloud Messaging, because:

- It is the industry standard.
- It supports all required operating systems.

6.1.11 Messaging

Context and Problem Statement

For real-time messaging between the users, a reliable messaging solution is needed.

Considered Options

- Lambda + Firestore/DynamoDB
- Stream Chat
- Sendbird
- PubNub

Decision Outcome

Chosen option: PubNub, because:

- It provides integration with FCM.
- It has Android and iOS SDKs and the integration of a native solution is valuable to the KMP evaluation.
- The free tier is sufficient for the scope of this project.

6.1.12 Search

Context and Problem Statement

The search function is an important aspect of the app. Instead of relying on a full-text search, a search engine that supports features like fuzzy matching will be used.

Considered Options

- Meilisearch
- OpenSearch

Decision Outcome

Chosen option: Meilisearch, because:

- It is fast, lightweight and easy to set up.
- It has a Python SDK.
- It is open-source and has its own AMI on AWS.

6.1.13 CI/CD

Context and Problem Statement

To automate the building, testing and deployment of the code, a CI/CD platform is a must. This is needed for the backend as well as the frontend which includes iOS. This implies that the platform must offer macOS runners.

Considered Options

- GitLab CI/CD
- GitHub Actions
- Codemagic
- Bitrise

Decision Outcome

Chosen option: Gitlab CI/CD and Codemagic, because:

- GitLab is already used for the code repositories and thus used in combination with Codemagic.
- Codemagic is optimized and focused for mobile CI/CD and supports KMP.
- Codemagic is also required because the OST GitLab instance offers no macOS runners.

6.2 C4 Model

This section contains the C4 model of the project, giving an overview of the system architecture. It depicts the architecture on different levels of abstraction, with level one, Context, being the most superficial and level four being the most detailed. Only level one to three have been modeled, on the basis that the level four diagram would be too specific and subject to too many changes. The diagrams from level one to three are sufficient enough to gain a good understanding of the architecture.

6.2.1 Context

The context diagram gives an overview of how the system interacts with all external systems and actors.

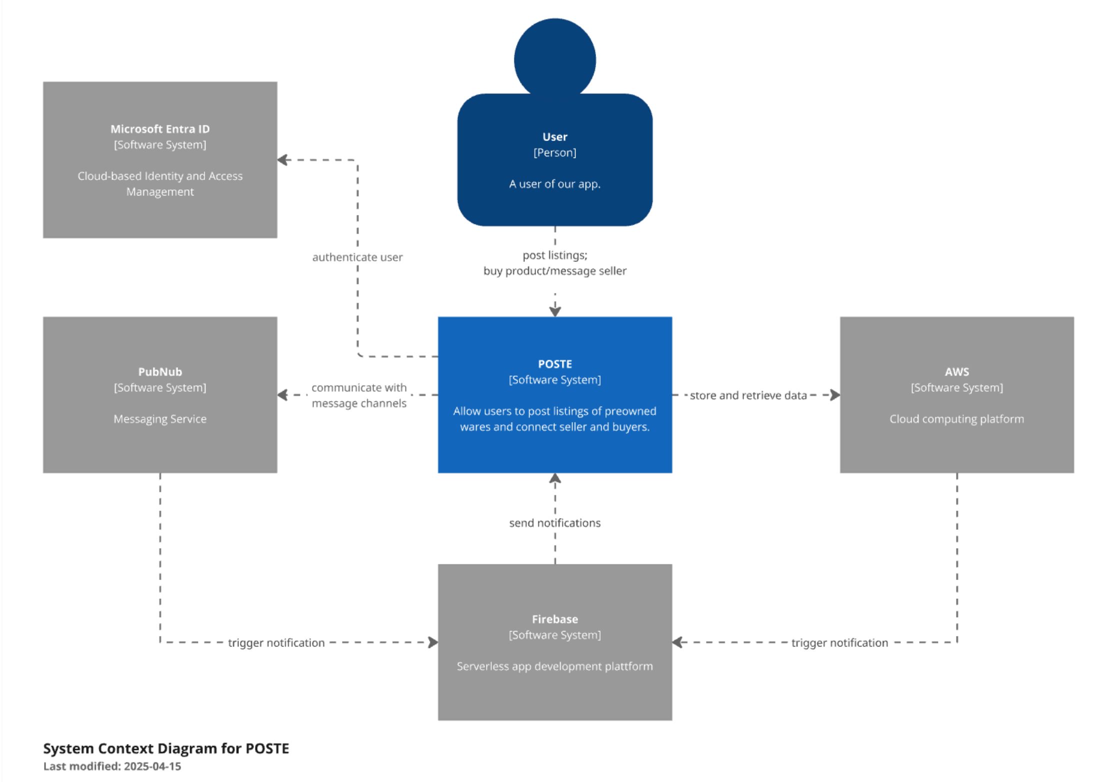


Figure 6.1: C4 Context Diagram

6.2.2 Container

The container diagram takes a closer look at the POSTE system itself, revealing how the system is split up and how the containers interact with each other.

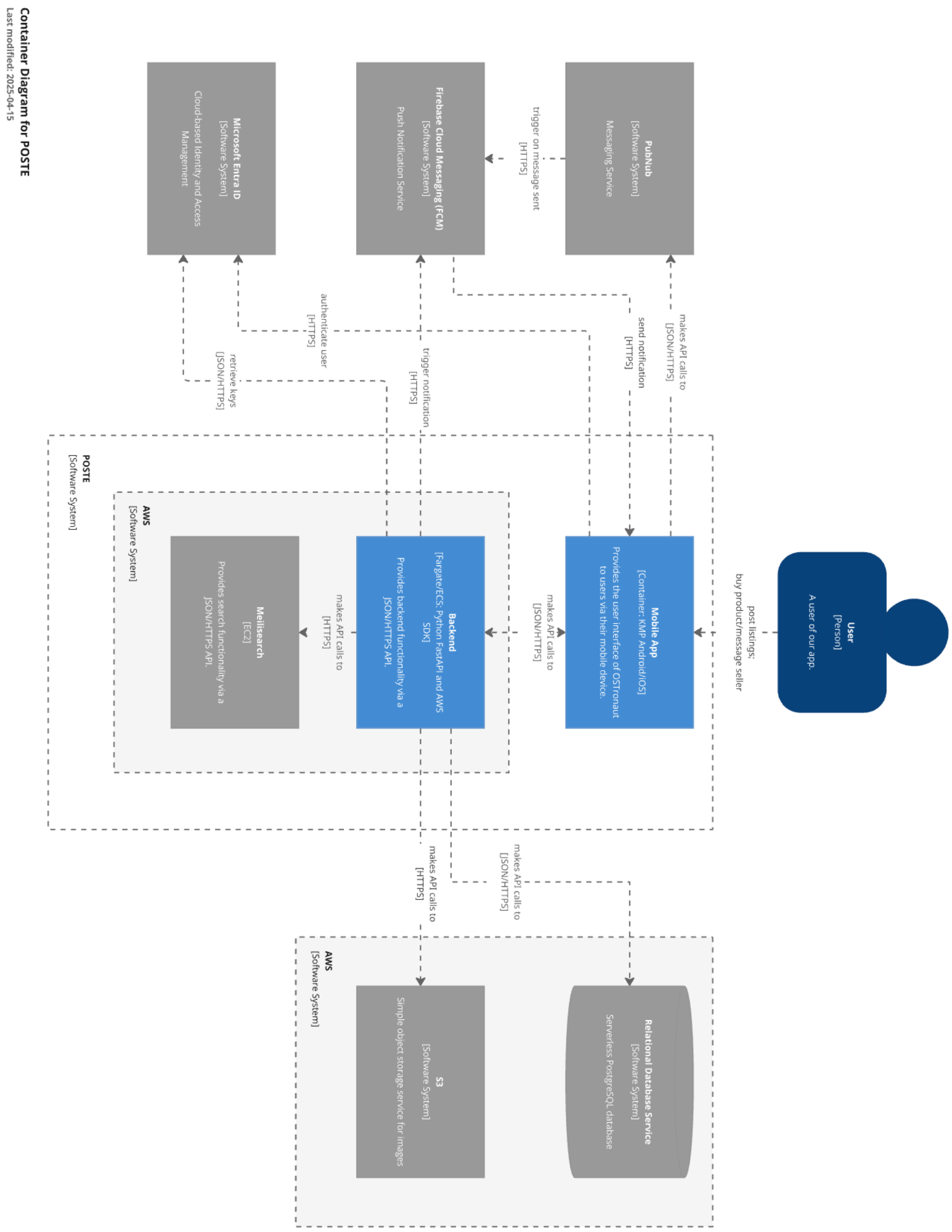


Figure 6.2: C4 Container Diagram

6.2.3 Component

The component diagram depicts the internal structure of POSTE's own containers "Mobile App" and "Backend", showing their key components and their interactions.

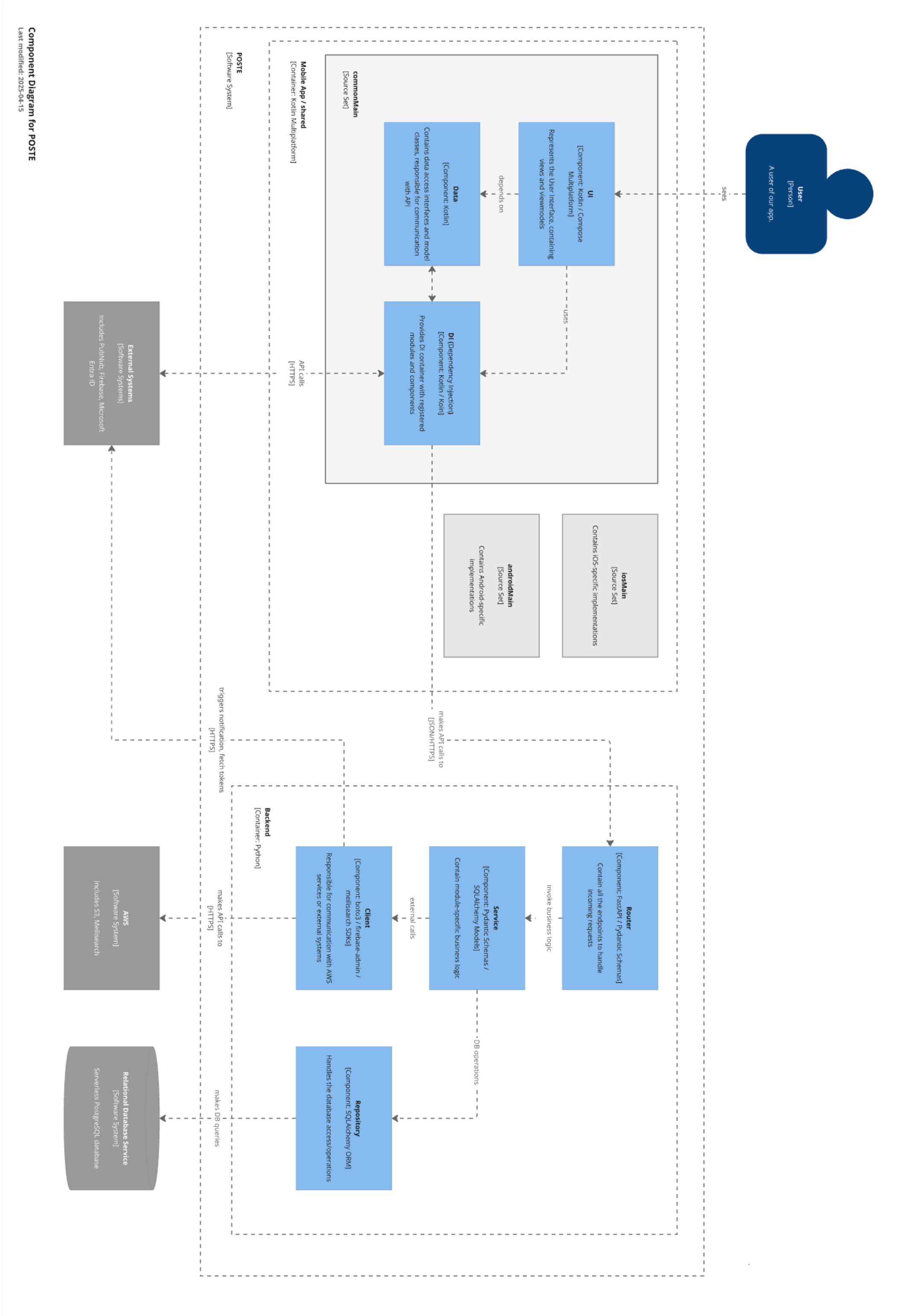


Figure 6.3: C4 Component Diagram

6.3 Component Interaction Diagrams

This section shows high-level component interaction diagrams. They focus on key use cases that cover all connections and components. Their purpose is to showcase the different interfaces in the POSTE system.

6.3.1 Login Flow

Following diagram shows the login process and the interactions between the app and Microsoft Entra ID. After the authentication succeedes, all backend requests must be made with the required JWT or else they will not be accepted.

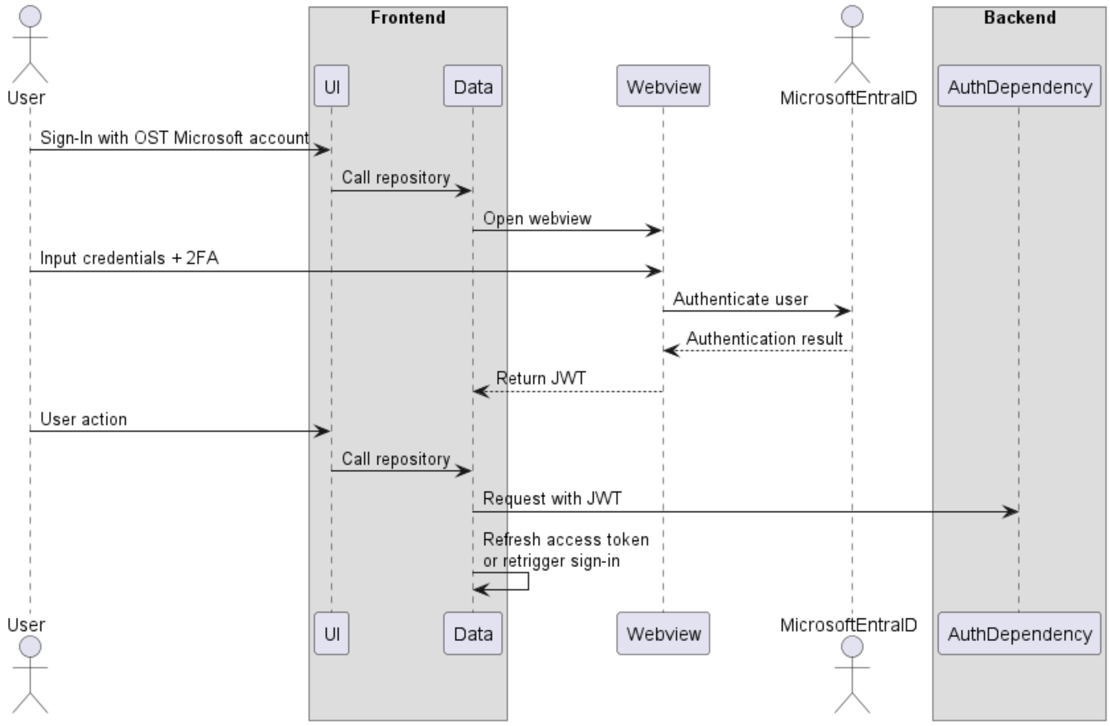


Figure 6.4: CID Login Flow

6.3.2 Create Listing

To create a new listing, the user submits all required information. Once the user confirms, the listing is created in the database, indexed in the search engine and afterwards the images are uploaded via a separate call.

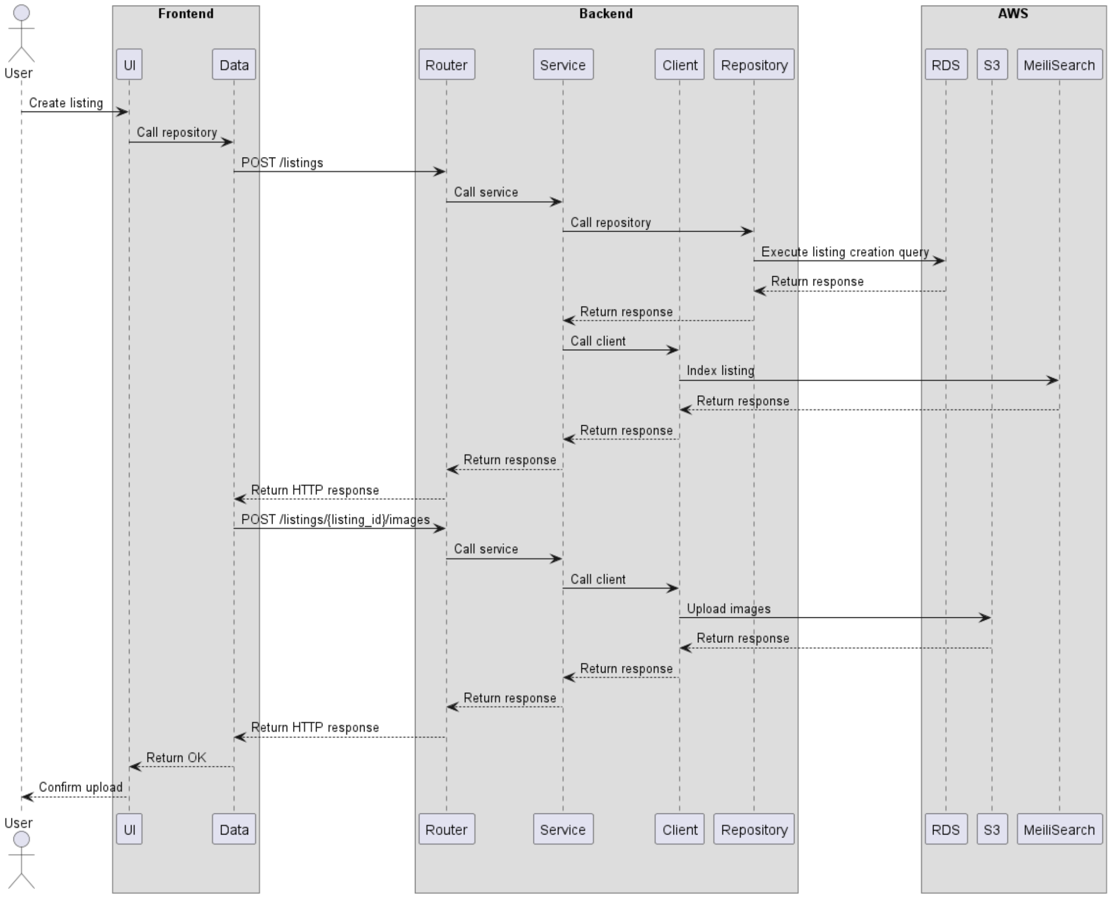


Figure 6.5: CID Create Listing

6.3.3 Search

The search starts off by retrieving all listings that match the query and filter parameters. That response also contains a temporary pre-signed URL to the thumbnail which resides in the S3 bucket. The results are then shown to the user while the thumbnails are fetched in the background. This usually happens quickly enough that the thumbnails load at the same time as the listings.

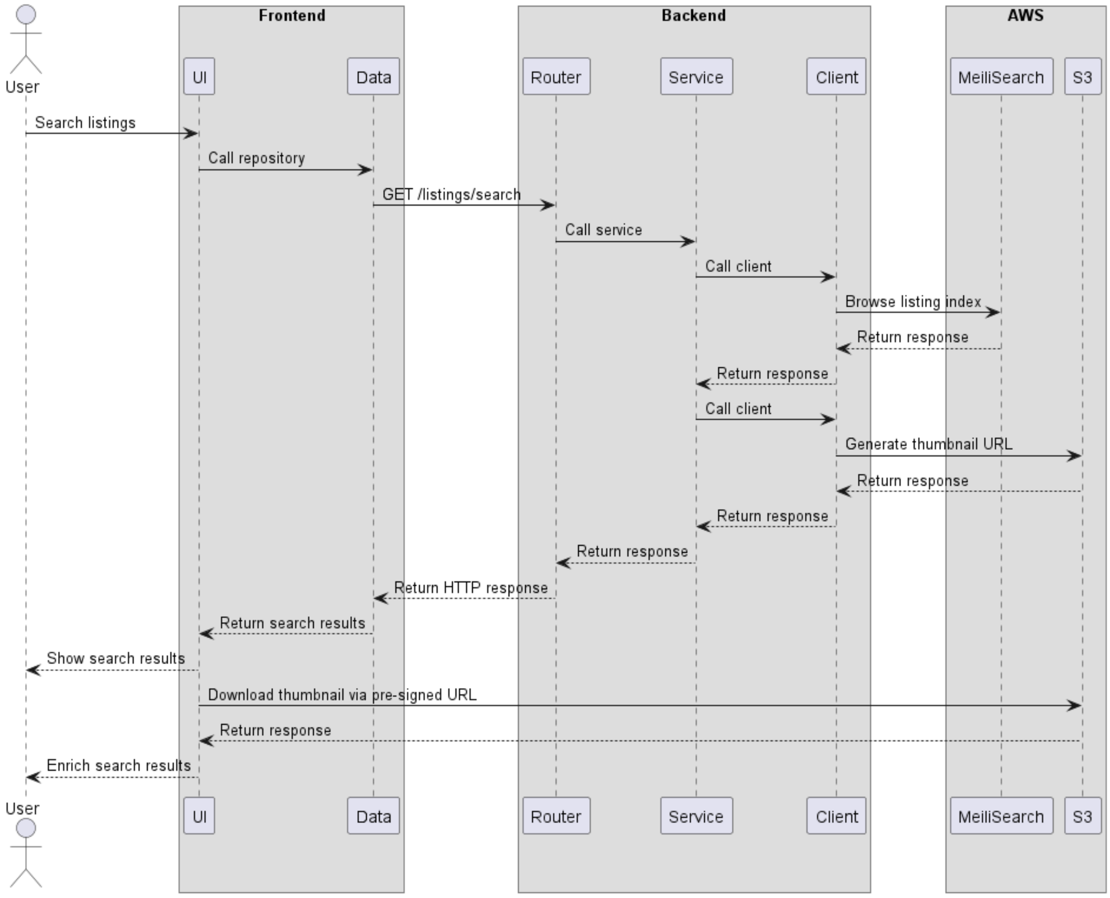


Figure 6.6: CID Search

6.3.4 Chat

All interactions in a chat happen between the frontend, PubNub and Firebase as seen in the following diagram.

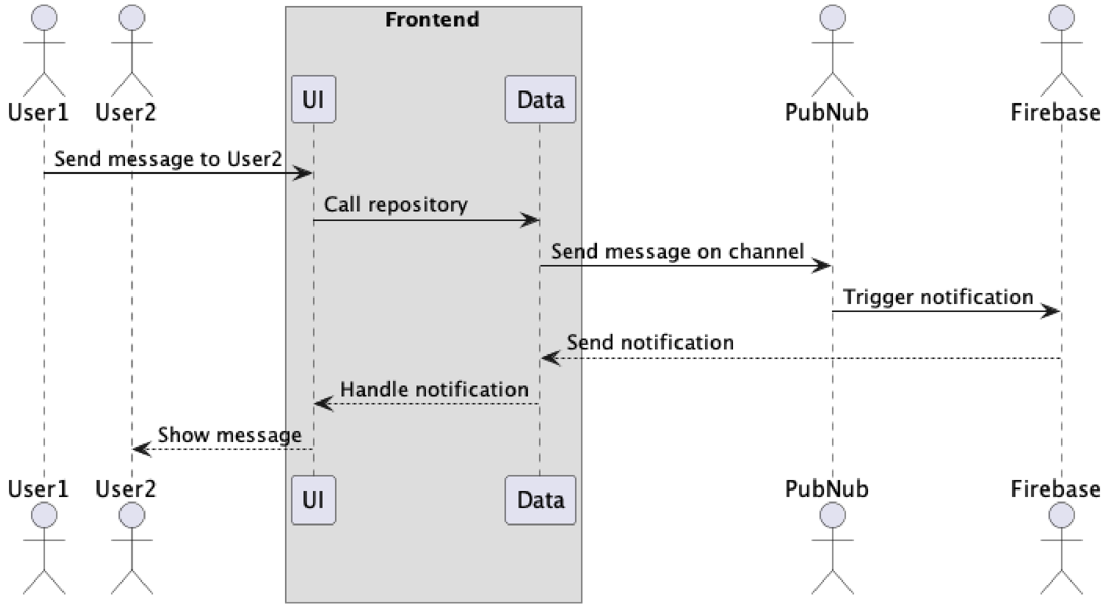


Figure 6.7: CID Chat

6.4 Frontend

Kotlin Multiplatform (KMP) applications built with Compose Multiplatform share a similar project architecture to native Android applications that use Jetpack Compose, with the shared code residing in the common source sets.

This similarity enables the use of the same architectural principles applied in Android apps. Consequently, the frontend of the POSTE system follows the typical app architecture described by Google.

Leveraging this uni-directional data flow, with Koin as the chosen dependency injection framework, the app remains flexible, expandable and testable throughout the development process.

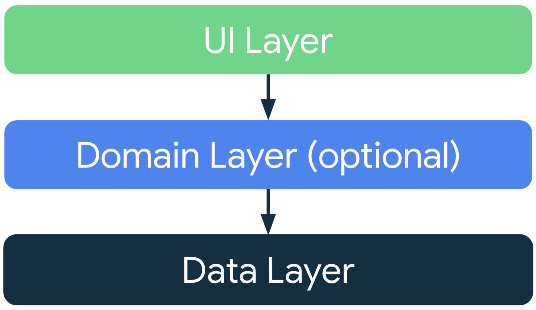


Figure 6.8: Typical App Architecture [6]

The actual project structure and architectural patterns used in the frontend are based on the recommendations by Ignacio Carrión, an experienced member of the KMP community and Koin contributor, as well as Volodymyr Tarasov, a KMP Medium poster.

For the architectural pattern MVVM is used, promoting a reactive approach and clean separation of concerns [7]. This decision was made according to the experience of the development team.

The project structure is inspired by Volodymyr Tarasov's Medium post [8], specifically the `:mobile` module. As the diagram below shows, the structure inside the `composeApp` module is split into `commonMain` and its platform-specific counterparts, all containing following packages:

data: The outer most package, handling connections to the backend and other external services.

di: The dependency injection package, responsible for setting up and providing the DI container.

ui: The ui package, containing the common UI code written with Compose Multiplatform.

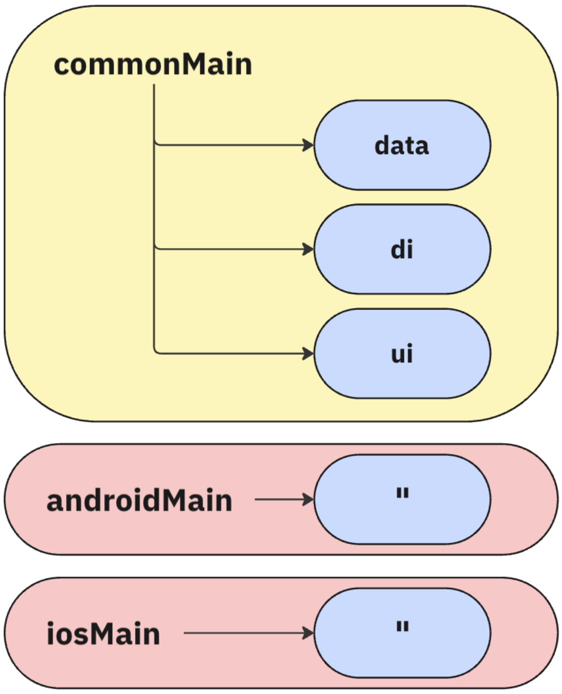


Figure 6.9: Planned Project Structure inside `composeApp`

6.5 Backend

The backend architecture is based on multiple well-known community standards. As the backend REST API is built using FastAPI, the best practices and conventions from its community are used [9]. These recommendations are inspired from the Netflix Dispatch project, which is also based on FastAPI [10]. Thus, the backend follows a domain-driven design where each package has its own router, service, schemas, models and so forth.

Additionally, the architecture is extended with a repository layer for the data access instead of handling that in the service layer [11]. This leads to a decoupling of the business and data access logic.

Following diagram depicts the flow that the requests go through.

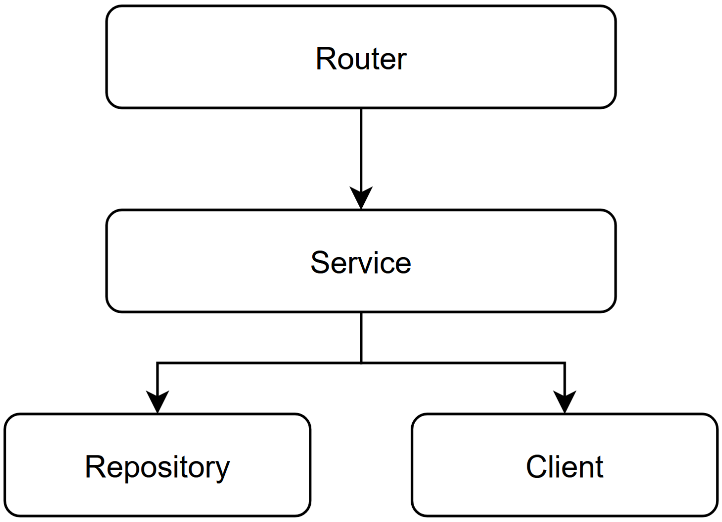


Figure 6.10: Backend Architecture

Router: Contains the endpoints which are exposed to the frontend and consumes the services provided by the service layer.

Service: Intermediary between the API endpoints and the repository/client layer, containing all business logic.

Repository: Interactions with the database happen through the repository layer.

Client: Used for external service communication.

6.6 Database

This section covers the database. The following entity relationship diagram shows the different tables of the database, including their attributes. The relationships between the tables and their cardinalities are depicted using the crow's foot notation.

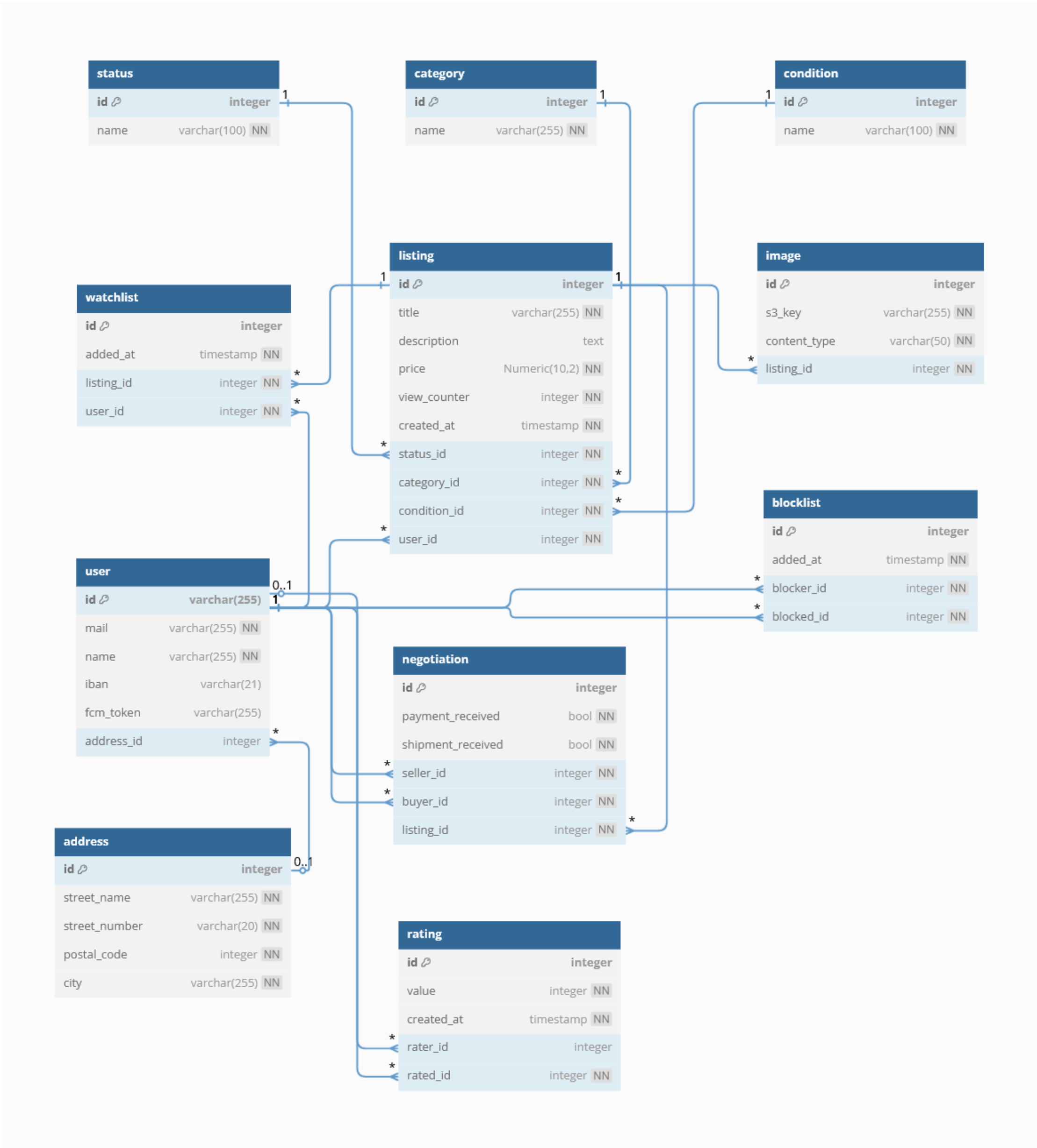


Figure 6.11: Entity Relationship Diagram

Something that is added in the ER diagram, which is not visible in the domain analysis, is the `negotiation` table. This extra table is required because a seller wants to interact and start the negotiation process with multiple buyers at the same time and vice versa. Each negotiation has an `is_completed` status that is computed from the two statuses payment and shipment received. If both are fulfilled, the negotiation is completed and the listing status is set to closed.

6.7 CI/CD

This section describes the setup of all pipelines used in the various repositories.

6.7.1 Frontend

The frontend CI/CD process is split up into two parts. Initial steps such as linting, unit testing and the measurement of the test coverage are done directly on GitLab where the repository is hosted.

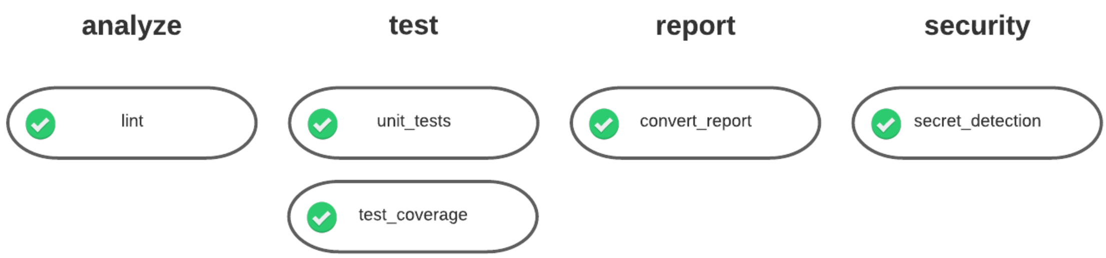


Figure 6.12: Frontend Pipeline - GitLab

Since no macOS runners are available on the OST GitLab instance, the building process it outsourced to Codemagic. Unit tests of all platforms are executed again, the respective artifacts are then built and signed. In the Android workflow, the application archives APK and AAB are shared via mail, whereas in the iOS workflow, the application archive IPA is also submitted to TestFlight.

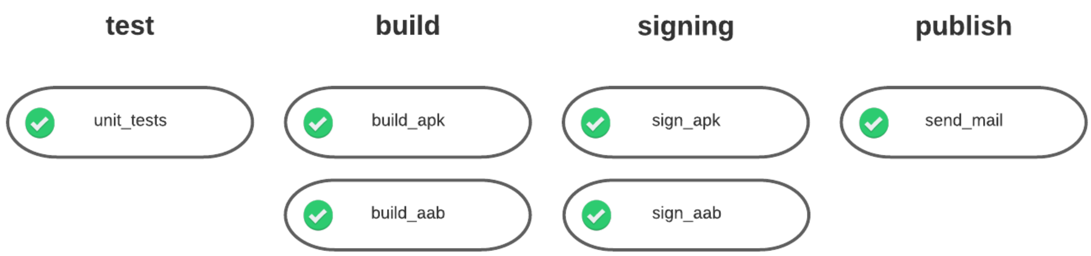


Figure 6.13: Frontend Pipeline - Codemagic Android

Figure 6.14: Frontend Pipeline - Codemagic iOS

6.7.2 Backend

The backend pipeline starts by linting the codebase and executing all integration tests, while measuring the percentage of code that is covered by them. Depending on the branch, either a dev or prod Docker image is built and pushed to the GitLab registry. This is followed by security checks and a container scanning task that scans for vulnerabilities. If on the main branch, the built image is also published to the AWS registry. This triggers a redeployment of all ECS containers.

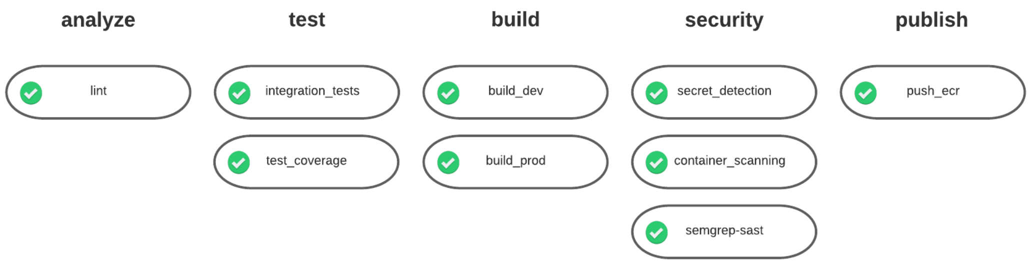


Figure 6.15: Backend Pipeline

6.7.3 AWS

The AWS repository, containing all the IaC code, has a pipeline that validates the configuration, generates a plan and then applies it.

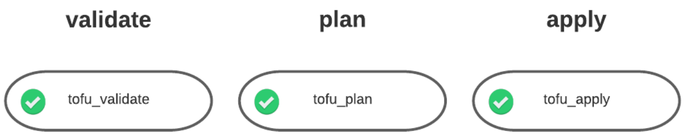


Figure 6.16: AWS Pipeline

6.8 Infrastructure

For the hosting of the backend instances, database, image storage and search engine, the AWS cloud is used. Figure 6.17 below illustrates the core infrastructure setup.

There is one VPC that contains all services except S3, which lives outside of it. For enhanced security and network segregation, public and private subnets are created. Only the ALB is publicly reachable and is the main entry point through the internet gateway. It listens on a specific domain on ports 80 and 443. Port 80 is redirected to

443 to enforce HTTPS traffic. The load balancer then routes the traffic to the backend instances hosted on ECS Fargate. These instances are the only ones with access to the RDS PostgreSQL database and the EC2 instance which hosts the Meilisearch search engine. The S3 bucket can be accessed through a VPC endpoint. A NAT gateway is configured to enable outgoing traffic from the backend instances, such as FCM calls. This design follows the AWS ECS networking best practices [12].

The backend container images are pushed from the GitLab CI/CD to the ECR.

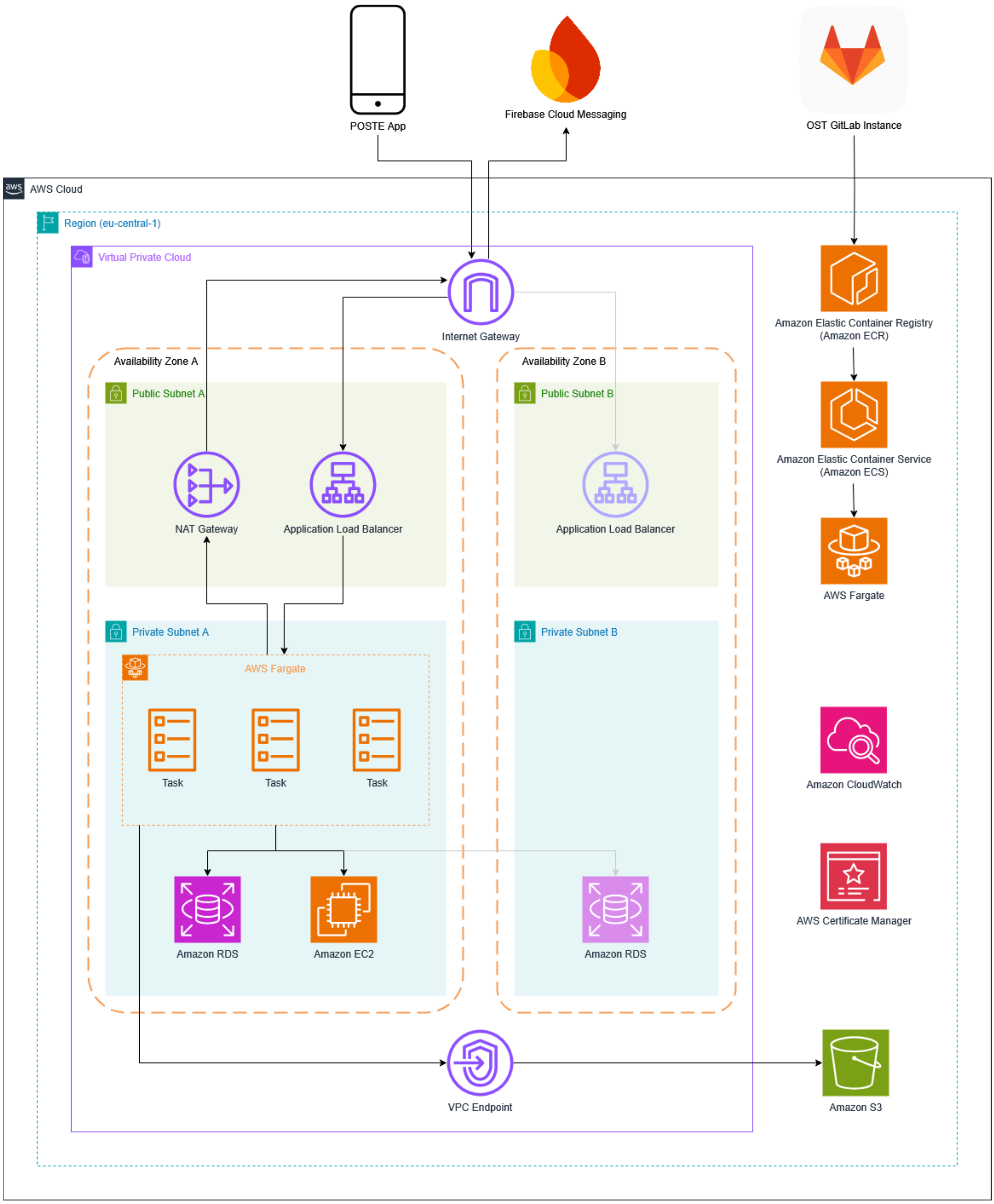


Figure 6.17: Cloud Infrastructure Diagram

6.9 PSP Integration Concept

This section outlines the conceptual approach to integrating a Payment Service Provider (PSP), as required by the task definition. First, the planned negotiation process of the MVP is presented. Subsequent subsections go over the motivation for the integration of a PSP and what changes would be required.

6.9.1 Planned Implementation

In the planned final product, a PSP is deliberately omitted. Instead, buyers and sellers can use the planned chat feature to conduct their transactions. This gives them maximum flexibility in how they handle their transactions. They can freely negotiate prices, shipping methods and payment preferences.

Additionally, there is a QR-bill feature. If a seller enters their swiss IBAN and address information, they have the option to send a QR-bill to the buyer. The Swiss QR-bill is automatically generated in the backend. If the buyer scans the bill using his e-banking app, it automatically fills in the necessary fields. If the agreed price changes during the negotiation, the buyer can adjust the amount before confirming.

Furthermore, this implementation bypasses the administrative and timely burden of integrating a PSP, which is beneficial given the limited timeframe of this thesis.

6.9.2 Motivation for PSP

In the long term, integrating a PSP can improve the security and the traceability. With the aforementioned solution, users of the app have greater flexibility, but in instances of fraud not much can be done. That risk is accepted for now as everything is limited to OST students and user names are visible and non-modifiable.

With a PSP, the funds could also be held until the shipment has been received. Integrated refund mechanisms would also be beneficial.

6.9.3 PSP Integration

Such an integration would require changes in the following areas.

User Flow

The payment flow would change significantly from the current chat-based process. Currently, buyers can initiate a chat with the seller, negotiate terms and then confirm if the payment or shipment has been received. The methods of payment and shipment are flexible.

However, integrating a PSP and automating the purchasing process would require streamlining the user flow. The existing onboarding process, which collects seller information, would have to be extended to include additional details such as a phone number and date of birth. Sellers would also be required to specify a fixed price and shipping method when creating a listing.

The purchase flow would change to the following:

1. Seller signs up and provides required information for payout setup.
2. Buyer browses items and clicks “Buy” on a listing.
3. Buyer enters payment details into hosted, pre-built page provided by the PSP.
4. Payment is securely processed.
5. Buyer gets payment confirmation; seller is notified of sale.
6. Possible fees are deducted and PSP transfers payout to the seller’s bank account.

Backend

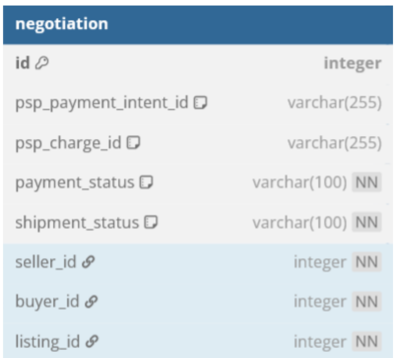
To support PSP integration, the backend must include the following functionalities:

- Forwarding collected user information to the PSP API for account creation.
- Storing PSP account ID in user database entry.
- API endpoints to process payments.
- Logic to delay payouts until the item is marked as received by buyer.
- Webhook listeners to receive payment status updates from the PSP (e.g., payment succeeded, failed, payout completed).

Database Model

The following changes to the database models are designed under the assumption that Stripe Connect is the chosen PSP.

The user table would need to be extended with the phone number, date of birth and a `psp_account_id` field to store the ID of the created PSP account. Additionally, the existing negotiation table could be adjusted and reused as follows:



negotiation		
id	integer	
psp_payment_intent_id	varchar(255)	
psp_charge_id	varchar(255)	
payment_status	varchar(100)	NN
shipment_status	varchar(100)	NN
seller_id	integer	NN
buyer_id	integer	NN
listing_id	integer	NN

Figure 6.18: Updated Negotiation Table

6.9.4 Comparison of PSPs

Following table compares major PSP platforms based on their official documentation.

	Stripe Connect [13, 14]	**Adyen for Platforms [15, 16]**	**PayPal for Marketplaces [17, 18]**
Onboarding	Custom via API or hosted UI	Custom via API or hosted UI	PayPal account required, onboarding via PayPal UI only
SDKs/Developer Tools	Server / web / mobile / terminal SDKs, REST API, CLI	Server / web / mobile / terminal SDKs, REST API (advanced/detailed explorer)	Server / web / mobile / terminal SDKs, REST/-GraphQL API
Compliance	PCI DSS Level 1[2]	PCI DSS Level 1[2]	PCI DSS Level 1[2]
Payment Methods	120+ payment methods (including TWINT)	100+ payment methods (including TWINT)	10+
Country Availability	46+ countries	33+ countries	200+ markets
Designed For	Small/medium companies, startups, MVPs	Enterprises, global platforms	Businesses of all sizes

Table 6.1: PSP Comparison

6.10 Extension

This section shows ways in which the current architecture could be extended in the future.

The project follows a modular architecture that promotes maintainability, flexibility and scalability. New features can be introduced independently, without disrupting existing functionality. Code quality is enforced through the pre-commit hooks and pipelines which guarantees that the code adheres to the guidelines.

[2]Payment Card Industry Data Security Standard; assures that businesses which accept, handle, store or transfer credit card information operate in a secure manner.

6.10.1 Development Environment

To enhance the developer experience and simulate the production environment more accurately, a possible extension is to run a second instance of all cloud infrastructure components, serving as the developer environment to support and extend local development. Having a mirrored production environment for development purposes would lead to smoother deployments, reduce the risk of environment-specific bugs and ease of use for frontend developers.

6.10.2 Backend Microservices

A future improvement would be to split up the backend into multiple services instead of one big application. This would enhance the scalability, fault isolation and flexibility.

Additionally, the interactions with PubNub could be handled over a separate service in the backend, instead of handling it in the mobile app. This would lead to less external systems that the frontend has to interact with.

6.11 Scaling

The following section discusses how the project could be scaled further.

6.11.1 Performance

The backend is the main component that would benefit from scaling or performance increases, as this is limited on the mobile app side. Following are possible options to enhance the performance.

ECS Fargate: Autoscaling could be enabled based on CPU and memory utilization. Thus, the backend API could handle varying loads efficiently. Other solutions like AWS EKS could also be considered.

RDS: A possibility would be to add read replicas which could take on read-heavy workloads. This would reduce the utilization on the primary database.

6.11.2 Platforms

The POSTE app is currently supported on both Android and iOS. However, using KMP, the app could be made available on many more platforms.

Web: KMP supports the development of web applications, which would allow POSTE to run in modern web browsers.

Desktop: KMP also supports desktop applications, making it possible to deploy POSTE on various computer operating systems.

Chapter 7

Design

This chapter documents all decisions regarding the user interface and visuals.

7.1 Colors

As the POSTE app is to be used in the OST environment, the colors used in the app are based on the official OST logo.

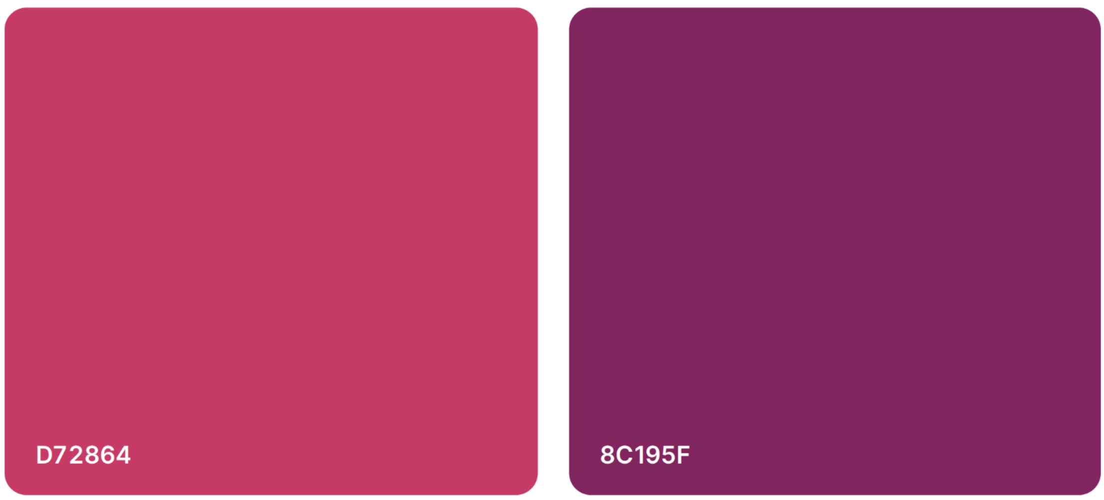


Figure 7.1: Main Colors

7.2 Logo

The following logo is used for the app icon and other key placements. The name of the app, "POSTE", is combined with the official OST logo, to create a context between buying/selling items and OST.

Figure 7.2: POSTE App Logo

7.3 Prototyping

To ensure a smooth implementation process and minimize the risk of hasty decisions, considerable attention is given to the design prototypes. All the designs can be found in Appendix C.

7.3.1 Low-Fidelity

The focus of the low-fidelity designs lies on showcasing the arrangement of components and their interactions. It should be intuitive to use and navigate.

7.3.2 High-Fidelity

The high-fidelity prototype shows the precise design of how the app should look like after implementation. The prototype is made with Figma [19] and the color scheme is generated based on the colors above using the Material Theme Builder plugin [20] in Figma. Some variation in the actual implementation is acceptable, but it should closely resemble the prototype.

Chapter 8

Implementation

This chapter describes selected implementations as well as certain technologies, dependencies, philosophies and code examples.

8.1 Frontend

In this section, interesting parts of the frontend implementation are discussed.

8.1.1 App Architecture

The app is structured as seen below:

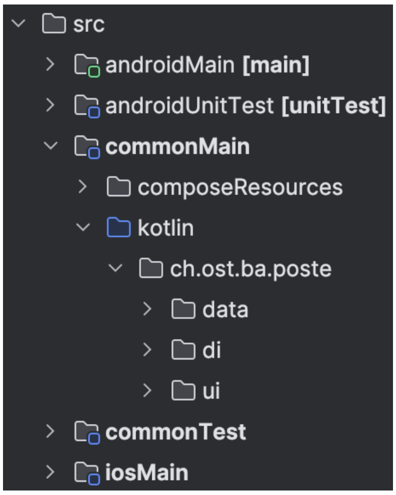


Figure 8.1: App Structure

The shared code is inside the commonMain package and split up into `data`, `di` and `ui`. `data` contains the repository and DTO classes and is split up into domains such as `listing` and `user`. The `di` package holds the configuration for the Koin DI framework. Lastly, `ui` houses UI code, including views, viewmodels, theme and custom composables.

8.1.2 Navigation

Implementing the navigation is a key part of any mobile app. At the beginning of this thesis, the native navigation library of CMP was still experimental, meaning that breaking changes may occur frequently [21]. Therefore, an open source navigation library called Voyager is used. It was originally built for Jetpack Compose, but is fully compatible with CMP. In addition, it seamlessly integrates with Koin, the dependency injection framework used in this project.

Navigation Structure

POSTE has a navigation bar at root level to allow users to navigate between the most important functionalities of the app.

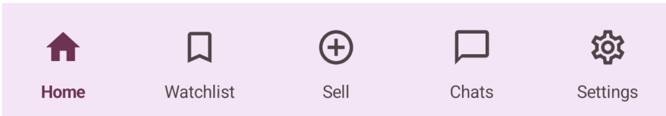


Figure 8.2: Navigation Bar

This is realised by using a Voyager Tab Navigator at the root, which contains and manages the five main tabs displayed in the bottom navigation bar:

```
// commonMain/PosteApp.
TabNavigator(HomeTab) {
    Scaffold(
        /* ... */
        content = { CurrentTab() },
        bottomBar = {
            val tabNavigationVisible
                by navStateManager.isTab.collectAsState()

            AnimatedVisibility(/* ... */) {
                NavigationBar(/* ... */) {
                    TabNavigationItem(HomeTab)
                    TabNavigationItem(WatchlistTab)
                    TabNavigationItem(CreateTab)
                    TabNavigationItem(ChatsTab,
                        TestTag.CHAT_ICON_TAB)
                    TabNavigationItem(SettingsTab)
                }
            }
        },
    )
}
```

Listing 8.1: Tab Navigator

Each tab is implemented as a "PosteTab", a customized version of the Voyager Tab, to allow the modification of the tab icons. Each "PosteTab" instance encapsulates its own Voyager Navigator instance, enabling independent navigation stacks within each tab section. This design allows users to navigate to deeper screens within any tab while maintaining separate navigation histories.

Furthermore, a `NavigationStateManager` is implemented to manage the visibility of the navigation bar. It ensures that the navigation bar is only displayed when users are at the root level of each tab, automatically hiding it when navigating to deeper screens.

8.1.3 Chat

The chat is implemented with a real-time messaging platform, enabling functionalities such as chatting, push notifications and real-time updates. It operates on a publish/-subscribe model.

As PubNub does not provide an SDK specifically for KMP, the chat functionality is implemented natively using the Kotlin Chat SDK and Swift Chat SDK for Android and iOS respectively.

PubNub

The `commonMain` directory contains the `PubNubCommon` interface, which provides all functions necessary to communicate with PubNub. These functions enable CRUD operations on chats and allow subscriptions to chats in order to receive live updates. The actual implementation of these functions are inside the respective native classes `PubNubAndroid` and `PubNubImpl` in the `androidMain` and the `iosApp`, which implement `PubNubCommon`.

Ultimately, the classes are used by the chat related *ScreenModels*, the Voyager equivalent of ViewModels. They do not depend on the implementations but rather on the `PubNubCommon` interface. The implementations get binded during compilation by the Koin DI framework.

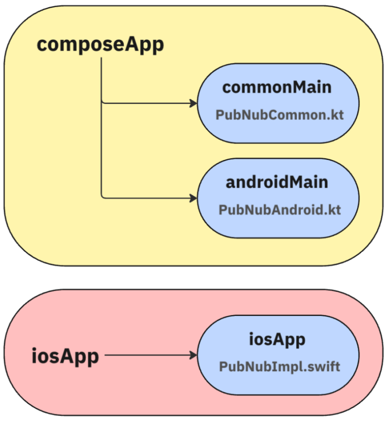


Figure 8.3: PubNub Implementation

Sequences

The flow of the chat is planned using sequence diagrams. They depict the entire flow with all possible cases and alternative scenarios.

Figure 8.4 illustrates the flow when a buyer initiates a chat session with a seller. When a buyer clicks a "MESSAGE SELLER" button, a negotiation is created in the backend, a chat session is created via PubNub and then the `ChatScreen` is shown to the user.

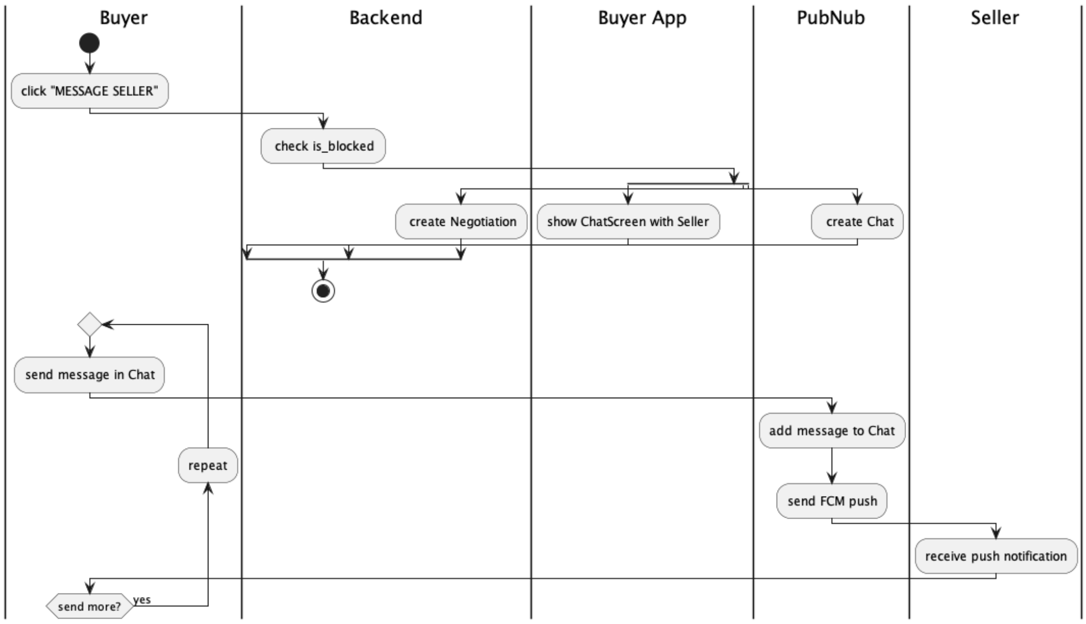


Figure 8.4: Start Chat Sequence Diagram

Figure 8.5 depicts the comprehensive workflow when a seller confirms payment receival, leading to transaction completion. If the buyer and seller both marked the package and payment as received respectively, the negotiation is completed and all chats between the seller and other potential buyers are deleted. Afterwards, FCM pushes are sent to the `ChatScreens` to display the rating UI. Upon rating or choosing to leave the chat, the cleanup of the chat session is handled. If the opposite party is still a member of the chat session, the user simply leaves the session. If the user is the only remaining member, the chat session is deleted entirely.

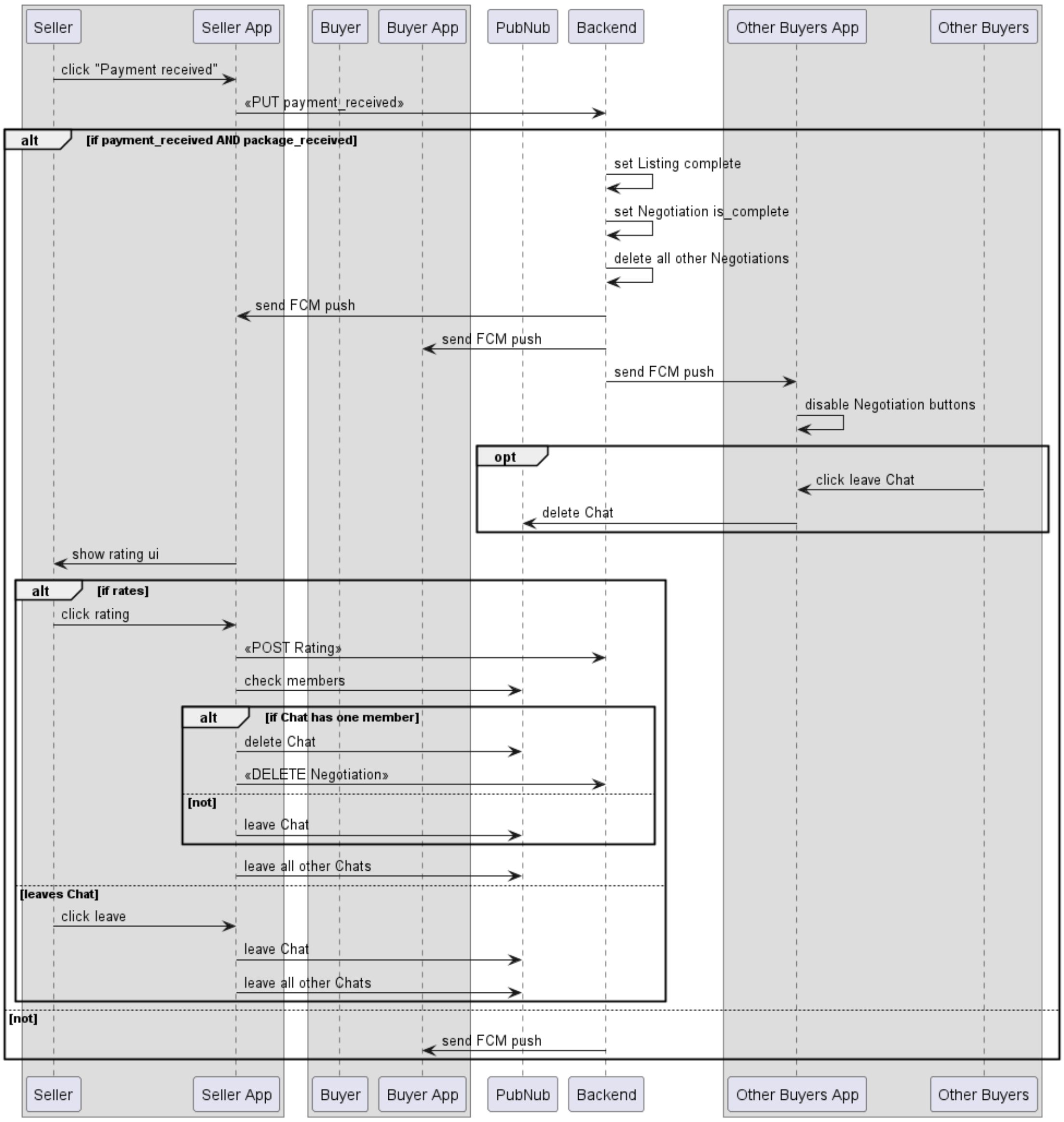


Figure 8.5: Completion Sequence Diagram

In Figure 8.6, the process when a seller decides to delete his listing, is shown. Upon doing so, a backend call is triggered which closes the listing and deletes all negotiations associated with it. Then, an FCM push is triggered which disables sending messages and marks the chat as closed. The buyer and seller can then leave the chat session.

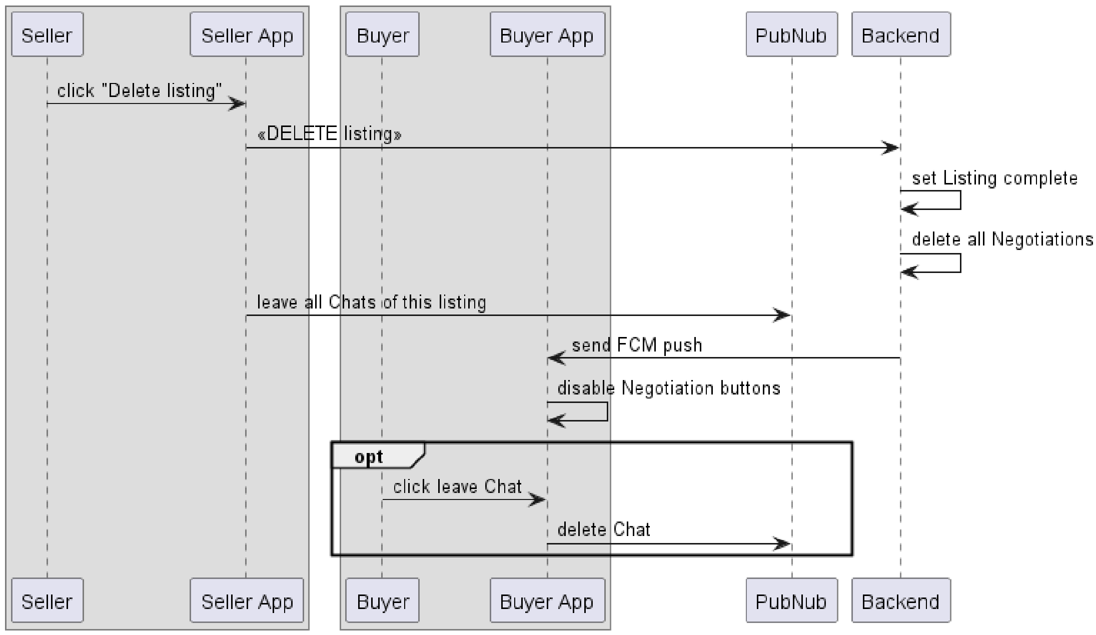


Figure 8.6: Delete Sequence Diagram

Figure 8.7 outlines the blocking mechanism and its impact on the chat. When a seller decides to block a buyer, the system removes the seller from all chats involving said buyer. The blocked user will still see the chat but interactions with the negotiation buttons is disabled.

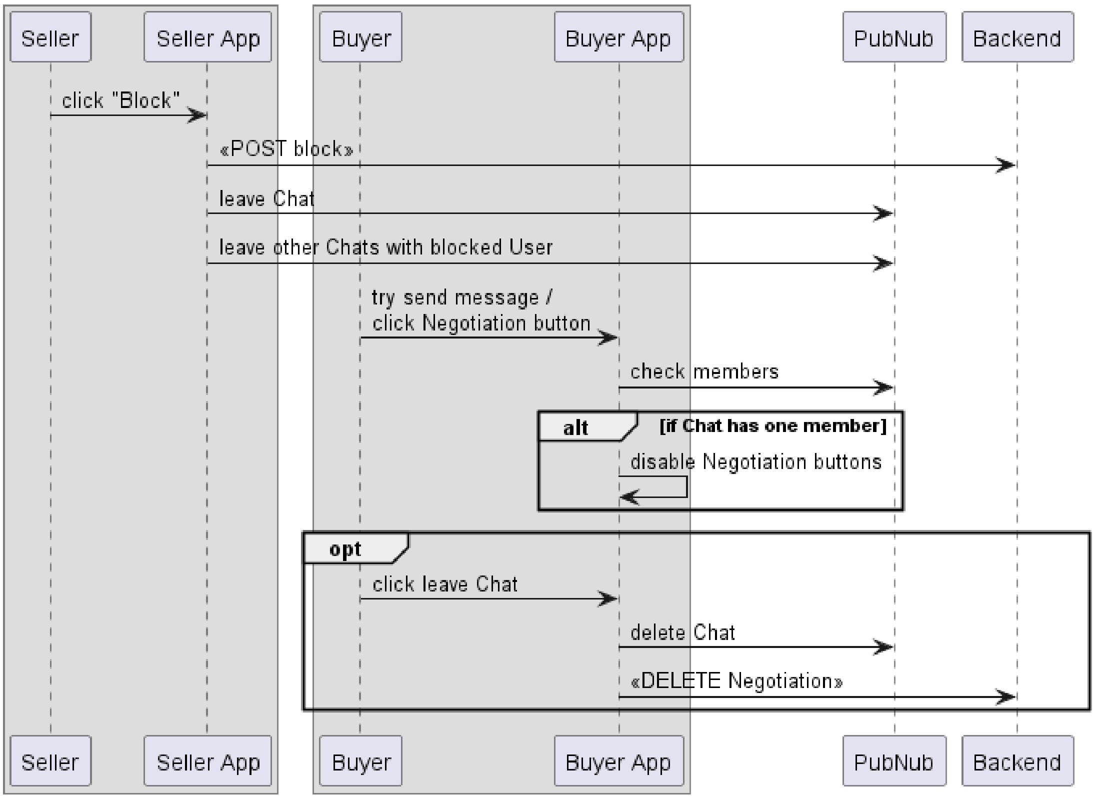


Figure 8.7: Block Sequence Diagram

8.1.4 Instrumented Testing

The implementation relies on instrumented tests. Nevertheless, some unit tests exist, but they were solely created to test how they function in regards to KMP and CMP.

The following code example shows an instrumented test which tests the tab navigation. This is done using the Compose UI testing framework [22]. First, the application context is initialized for Koin, the DI framework. The test initialization uses stub implementations for MSAL and PubNub, but the calls to the local backend are real, which covers the end-to-end aspect of the tests. Then the UI content to be tested is set. The `TestPosteApp` composable function used in this example, is the `PosteApp` composable with a custom `LifeCycleOwner`. This is necessary to prevent iOS UI tests from failing. Subsequently, the actual testing is conducted. It asserts that the `HomeScreen` is initially shown, then simulates a click onto the chat icon in the navigation bar and finally asserts that the content displayed changed to the `ChatScreen`.

```kotlin
// commonTest/NavigationTest.kt
@OptIn(ExperimentalTestApi::class)
@Test
fun tabNavigateChat_ContentIsChat() =
    runComposeUiTest {
        val koinHelper = KoinHelper()
        koinHelper.initTestKoin()

        setContent {
            TestPosteApp()
        }

        onNodeWithTag(TestTag.HOME_CONTENT).assertIsDisplayed()
        onNodeWithTag(TestTag.CHAT_ICON_TAB).assertIsDisplayed()
        onNodeWithTag(TestTag.CHAT_CONTENT).assertDoesNotExist()

        onNodeWithTag(TestTag.CHAT_ICON_TAB).performClick()
        onNodeWithTag(TestTag.CHAT_CONTENT).assertIsDisplayed()
    }
```

Listing 8.2: Instrumented Test Example

To be able to test if some content is shown, the content to be tested has to be tagged. The `TestTag` object, as used above, consists of string constants. These are used to tag UI components:

```kotlin
Column(
    modifier = modifier.testTag(TestTag.HomeContent)
) {
    // HomeScreen content
}
```

Listing 8.3: Usage of test tag

8.2 Backend

This section covers the backend implementation aspects.

8.2.1 API Architecture

As planned, the architecture of the API follows a domain-driven design and is structured as follows:

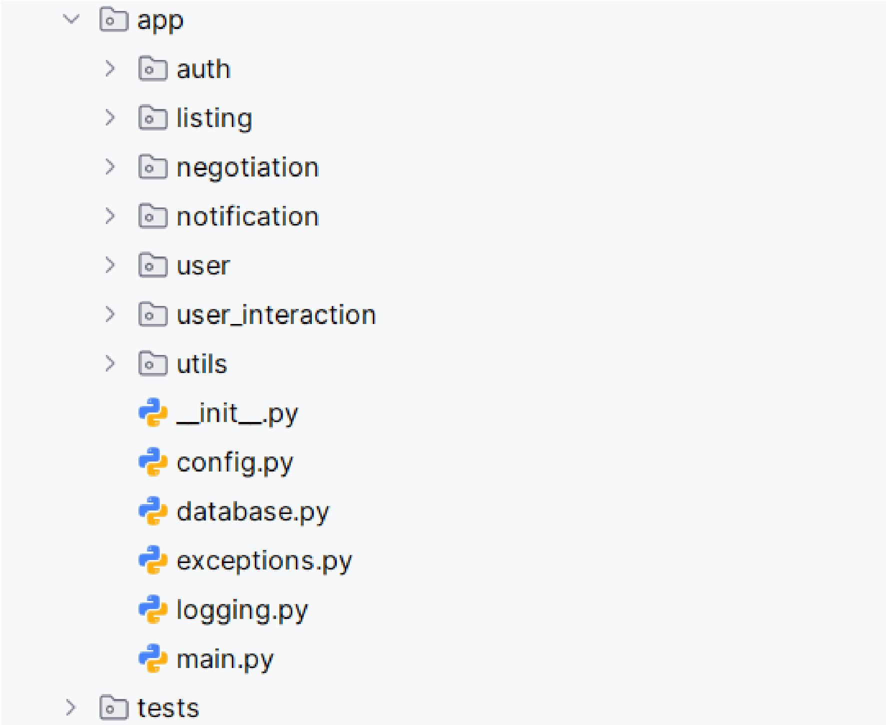


Figure 8.8: API Project Structure

Each domain contains all the components required. Following is an example of the listing domain:

Figure 8.9: Listing Domain Setup

Schemas

To automate the validation of incoming requests and the corresponding responses, Pydantic schemas are used [23]. These schemas define the expected structure and data types, which ensures consistency and reliability throughout the application. They eliminate the need for manual checks, while at the same time improving the code maintainability as well as the error reporting.

Following is an example schema of a listing. Other schemas build on this base schema to add or modify fields depending on the specific use case, such as creating, updating or returning detailed responses.

```python
class ListingBase(BaseModel):
    title: str
    description: Optional[str] = None
    price: Decimal
    condition_id: int
    category_id: int
```

Figure 8.10: Listing Base Schema

Models

Using SQLAlchemy, models which represent the structure of the database tables, are defined [24]. Each model maps to a table and contains definitions regarding its columns, relationships and possible constraints. SQLAlchemy's ORM is used to query and manipulate the data.

Incoming Pydantic schemas are converted into model instances when creating or updating data. The model instances are converted back into response schemas when returning data to clients.

Dependencies

The backend application relies on several SDKs and libraries to provide essential functionality. Below are the most important ones:

boto3: Is the AWS SDK for Python and used to interact with AWS services such as S3 to upload listing images [25].

meilisearch-python: To index new listings and make them available via the search engine, the Meilisearch Python API client is used [26].

firebase-admin-python: Push notifications need to be triggered in the backend in certain cases, for this the Firebase Admin Python SDK is required [27].

swiss-qr-bill: To provide the QR-bill generation feature, this library is used to generate Swiss QR-bill payment slips as SVGs [28].

8.2.2 Endpoints

The backend REST API offers about 30 endpoints for clients to interact with. The full documentation of all the endpoints, including the request and response formats, response codes and more, can be found on this Notion page. When running the app locally, all routes can also be inspected by accessing the Swagger UI served at `/docs`. The following is an example of such an endpoint.

Endpoint: `POST /listings`

Content-Type: `application/json`
Request Body:

```json
{
  "title": "MX Master 3",
  "description": "Working and in good condition.",
  "price": "29.99",
  "condition_id": 2,
  "category_id": 1
}
```

Listing 8.4: Request Body

Response Body (201 Created):

```json
{
  "id": 42,
  "title": "MX Master 3",
  "description": "Working and in good condition.",
  "price": "29.99",
  "view_counter": 0,
  "created_at": "2025-06-09 12:11:05.291563",
  "status_id": 1,
  "condition_id": 2,
  "category_id": 1,
  "owner": {
    "id": 1,
    "name": "John Doe",
    "rating": 2.5
  },
  "thumbnail": null
}
```

Listing 8.5: Response Body

Response Codes

- `201 Created`
- `400 Bad Request`
- `404 Not Found`

8.2.3 Integration Testing

The current implementation relies fully on integration tests. These tests cover the full request and response flow through the different layers. An endpoint is called, after which the response is verified and the repository is queried to ensure correct data access. The integration tests are conducted using the pytest framework [29].

```python
def test_create_listing(
    test_client, db_session, persisted_user, create_listing_request
):
    response = test_client.post(
        "/listings", json=create_listing_request.model_dump(mode="json")
    )
    assert response.status_code == status.HTTP_201_CREATED

    json_data = response.json()
    assert json_data.get("id") == 1
    # ... (additional response field checks)

    listing = db_session.query(Listing).filter(Listing.id == 1).one_or_none()
    assert listing is not None
    assert listing.id == 1
    # ... (additional DB field checks)
```

Listing 8.6: Integration Test Example

8.3 AWS

The cloud infrastructure is set up as planned in the architecture chapter. All cloud components are configured via IaC using OpenTofu, which can be found in the according repository [30].

8.3.1 VPC Resource Map

The following diagram shows the resulting VPC resource map after the infrastructure has been set up and configured. It shows the relationships between the resources in the VPC and the traffic flow. Included are public and private subnets which are linked to the route tables. There is also a default one which can be ignored. Public subnets connect to an Internet Gateway, private subnets use a NAT Gateway and an S3 VPC Endpoint provides S3 access.

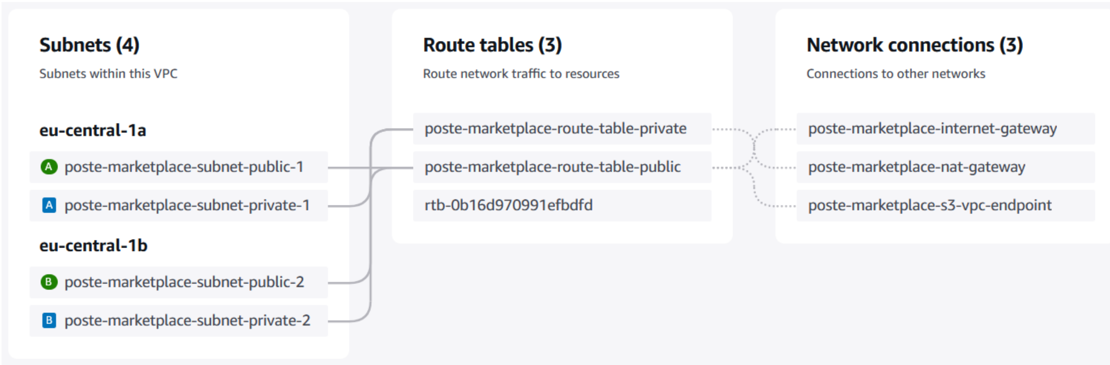


Figure 8.11: VPC Resource Map

8.3.2 Security

The principle of least privilege is followed to ensure secure communication between the services. Security Groups are used to control and restrict inbound and outbound traffic. For example, the database only allows traffic on port 5432 from the ECS service. All other traffic is blocked completely. This setup limits access to and from trusted internal resources within the environment, significantly reducing the attack surface.

```
resource "aws_vpc_security_group_ingress_rule" "security-group-rds-ingress" {
  security_group_id            = aws_security_group.security-group-rds.id
  from_port                    = 5432
  to_port                      = 5432
  ip_protocol                  = "tcp"
  referenced_security_group_id = aws_security_group.security-group-ecs.id
}
```

Listing 8.7: Security Group Example

8.3.3 Remote State

As this is a shared environment, the state of the infrastructure needs to be saved at a remote location and locked if it is in use. For this, a remote backend is configured where the state is saved in an S3 bucket and the locks are handled via DynamoDB.

```
terraform {
  backend "s3" {
    bucket         = "poste-marketplace-terraform-state"
    key            = "terraform.tfstate"
    profile        = "poste"
    region         = "eu-central-1"
    dynamodb_table = "poste-marketplace-terraform-state-lock"
  }
}
```

Listing 8.8: Remote State

Chapter 9

Results

This chapter highlights the produced results of the marketplace project.

9.1 NFR Validation

The NFRs defined at the start of the project are validated two times throughout the course of the project. The first time is right after the beta release and the second time after the final release, the MVP.

9.1.1 Beta Validation

Below are the results of the first NFR validation.

ID	Measured	Accepted
NFR1	The fetching and displaying of the search results, including the thumbnails, takes around 100-300ms which puts it in the outstanding landing zone.	✓
NFR2	Multiple stress tests show that 150 search requests per minute are no problem for the backend and can be processed easily.	✓
NFR3	After inserting 120'000 listings into the database, just shy of 30MB of the available 5GB is used. The database can thus handle way more than the required amount of entries.	✓
NFR4	Using the AWS ALB as an entry point, the incoming requests are load-balanced between all active backend nodes. Healthchecks happen periodically, and if a node is unavailable, the operability is maintained by only sending traffic to healthy nodes while a replacement is provisioned.	✓
NFR5	A daily backup is made of the database. These backups are kept for 35 days and a restore can be made at any time.	✓
NFR6	The app did not crash due to user input during all usability tests.	✓
NFR7	Pre-commit hooks and GitLab pipelines enforce the defined code guidelines. No changes can be made unless they succeed.	✓
NFR8	The entrypoint of the backend, the ALB, only accepts and enforces HTTPS. JWT authentication ensures only requests from the app are processed. Connections to external systems are secured as well. All the data in the database is automatically encrypted at rest.	✓
NFR9	This is verified by the whole team on an ad hoc basis. No issues have been found.	✓
NFR10	If any errors occur, a snackbar describing the error is shown to the user.	✓
NFR11	As seen in the usability test protocols, the general feedback is positive.	✓
NFR12	The beta release is tested on Android and iOS. The usability tests are conducted on both operating systems to gather as much information as possible.	✓

Table 9.1: NFR Validation - Beta

9.1.2 MVP Validation

Below is the protocol from the MVP validation of all NFRs.

ID	Measured	Accepted
NFR1	Revalidated.	✓
NFR2	Revalidated.	✓
NFR4	Revalidated.	✓
NFR5	Revalidated.	✓
NFR6	Revalidated.	✓
NFR7	Revalidated.	✓
NFR8	Revalidated.	✓
NFR9	Revalidated.	✓
NFR10	Revalidated.	✓
NFR11	Revalidated.	✓
NFR12	Revalidated.	✓

Table 9.2: NFR Validation - MVP

9.2 Final Product

The final product implements all use cases defined for the MVP, specifically UC1 through UC15. All NFRs, as seen in the table above, are validated and accepted as well. In addition to the defined use cases, several additional, unplanned features are implemented. These features originate from feedback during meetings or usability testing sessions and were subsequently refined, estimated and prioritised through an iterative development process.

ID	Description
EC1	On the home page, the user can view popular listings (called "Hot Listings"). These are randomly selected listings whose view count is over a certain threshold.
EC2	All screens that require fetching of externally hosted data display loading indicators until the data is retrieved, processed and ready to show. This includes data from the backend, such as user data and listings, and data regarding chats from PubNub.
EC3	The home page includes a horizontally scrollable overview of all available categories.
EC4	The app implements a dark mode which can be enabled and disabled in the app settings.

Table 9.3: Extended Cases

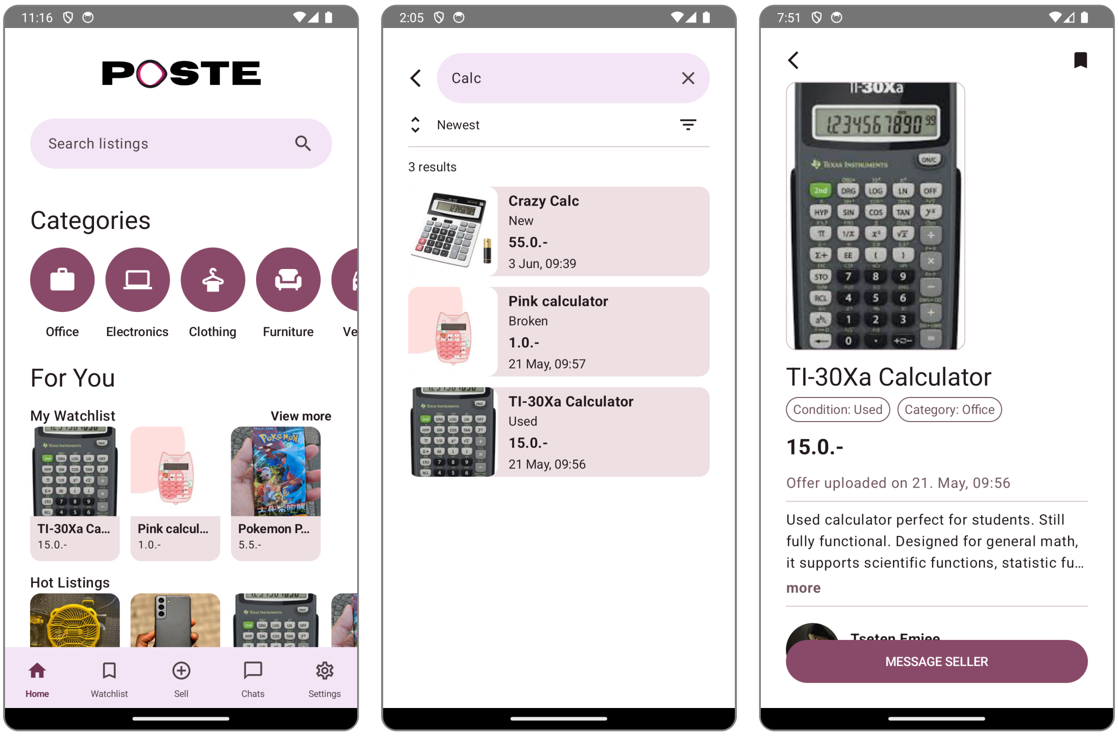


Figure 9.1: MVP App - Home Screen, Search and Listing

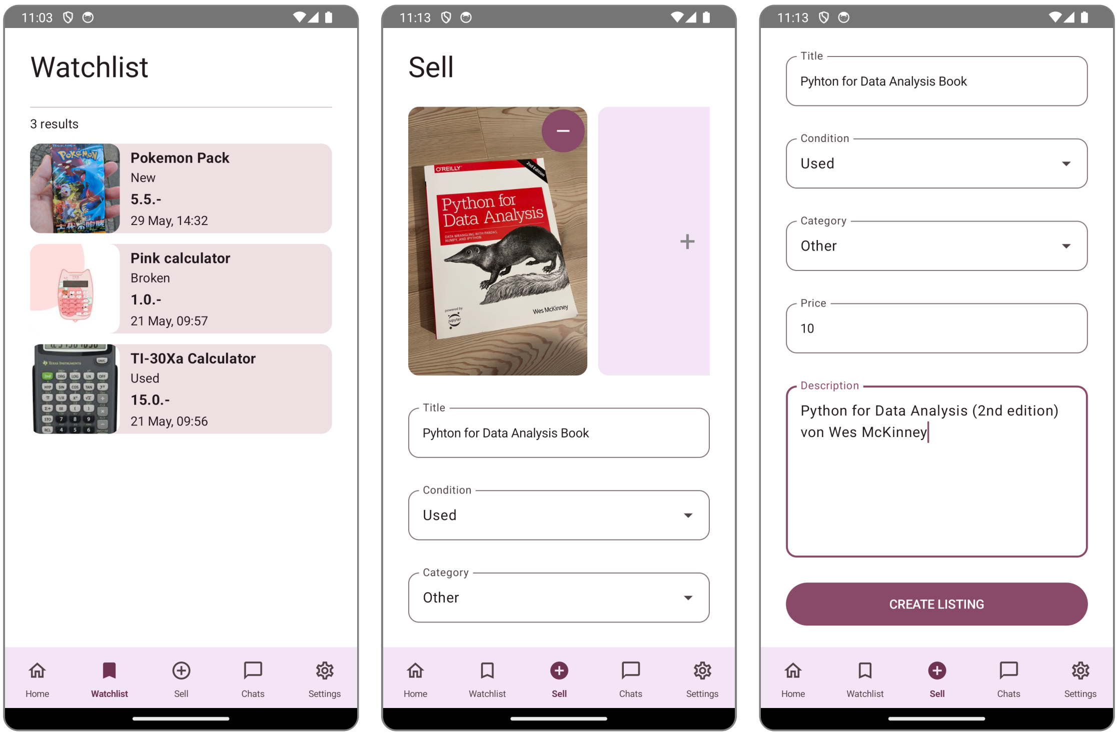


Figure 9.2: MVP App - Watchlist, Sell, Sell (Scrolled)

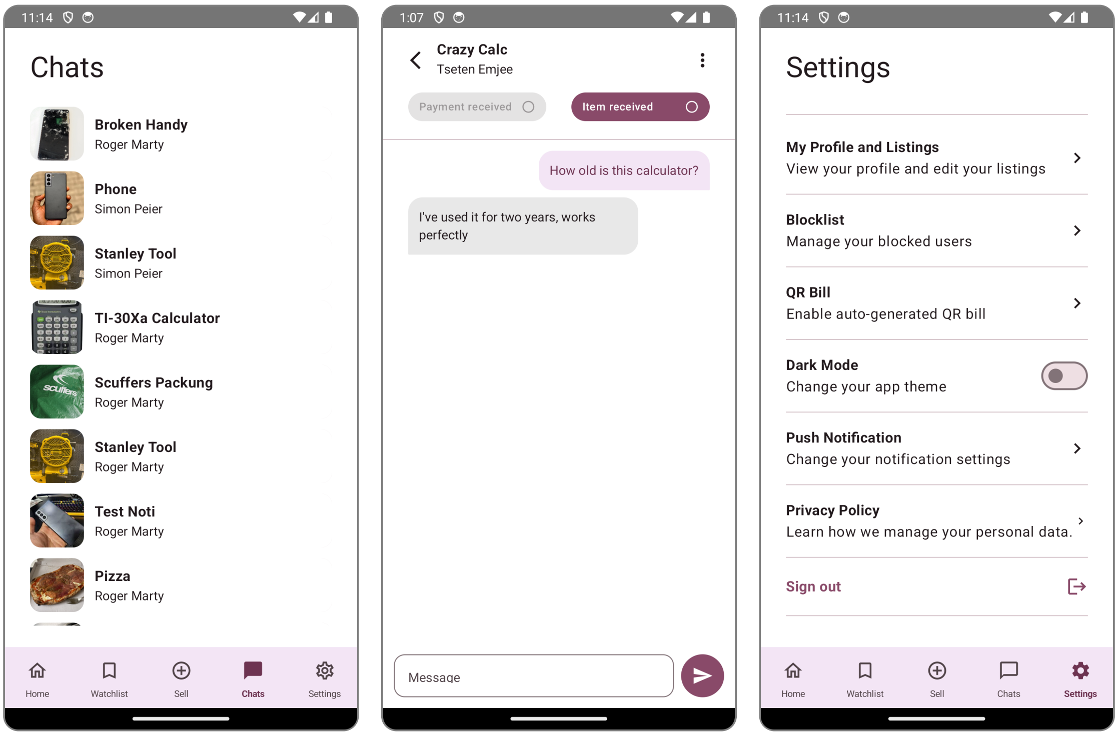


Figure 9.3: MVP App - Chats overview, Chat, Settings

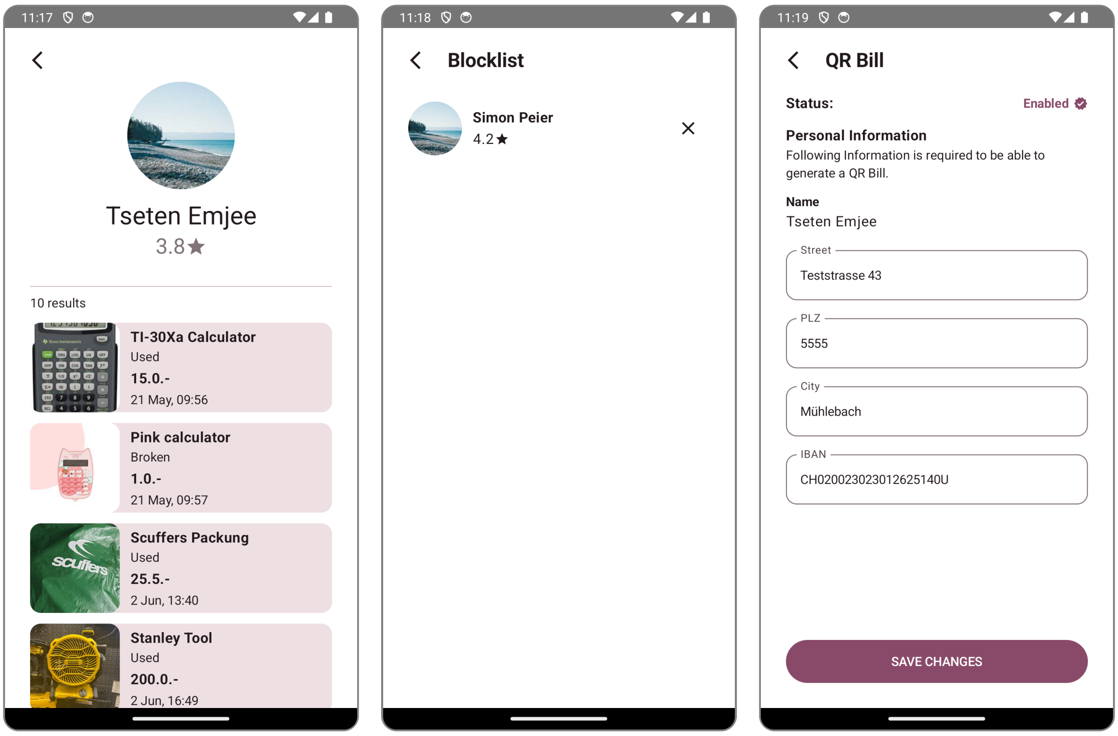


Figure 9.4: MVP App - My Profile and Listings, Blocklist, QR Bill

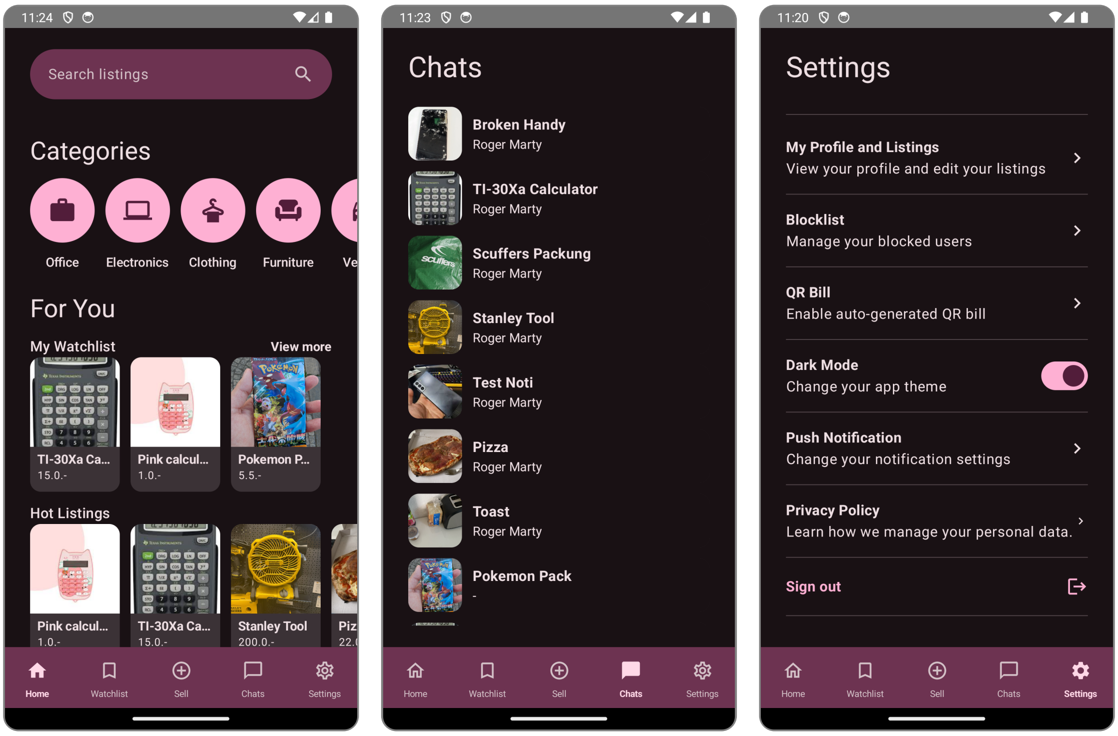


Figure 9.5: MVP App - Selection of screens in dark mode

Part VI

Research Documentation

Chapter 10

Kotlin Multiplatform

This chapter provides an introduction, as well as a deep-dive into the concepts of the Kotlin Multiplatform (KMP) technology.

10.1 Overview

Kotlin Multiplatform is a cross-platform technology developed and open-sourced by JetBrains. This technology aims to allow easy sharing and reuse of Kotlin code between different platforms, while still retaining the advantages of native programming [31].

Today, KMP supports a total of eight different platforms, allowing them all to share business logic together. Compose Multiplatform (CMP), the sub-branch of KMP, additionally allows the sharing of UI code based on the Jetpack Compose UI framework and is supported by four of the eight platforms [32].

Platform	**Stability Level March 2025**
Android	Stable
iOS	Stable
Desktop (JVM)	Stable
Server-Side (JVM)	Stable
Web (based on Kotlin/WASM)	Alpha
Web (based on Kotlin/JS)	Stable
watchOS	Best effort
tvOS	Best effort

Table 10.1: Kotlin Multiplatform Stability Levels

As seen in the figure below, there are three main ways to share code using KMP.

The technology allows developers to share only some specific logic or module with their applications to isolate critical functions. This aids in keeping big codebases maintainable.

One can also write the entire business and data handling logic using KMP and then share it with the native UIs. This approach enables products with demanding performance and strict native UI requirements.

If development speed and a smaller, more central codebase is required, developers may also "share up to 100%" of their code using the CMP framework. UI code is then written only once for the supported platforms and adding native integrations is possible when needed.

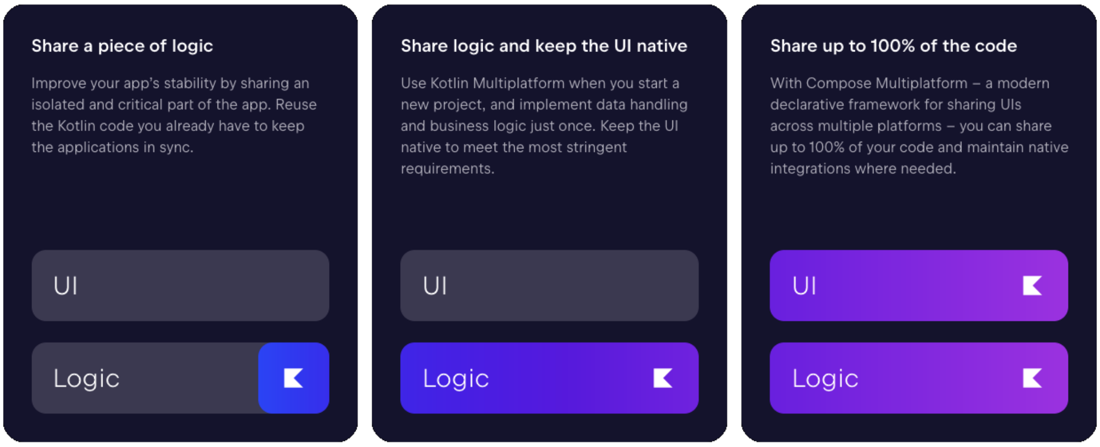


Figure 10.1: Ways to share code in KMP [33]

10.2 Compose Multiplatform

To be able to share UI code across supported platforms, the Compose Multiplatform (CMP) framework needs to be used. CMP is a declarative and reactive framework, also created by JetBrains. It extends Google's Jetpack Compose framework, which allows the creation of declarative UI's for Android, by supporting additional platforms including iOS, Desktop (Linux, MacOS, Windows) and Web. As of the time of writing, the stability of the supported platforms is as follows:

Platform	**Stability Level March 2025**
Android	Stable
iOS	Beta[3]
Desktop (JVM)	Stable
Web (based on Kotlin/WASM)	Alpha

Table 10.2: Compose Multiplatform Stability Levels

To check the current stability levels, consult the official documentation [21].

10.2.1 Features & Constraints

For developers already familiar with Jetpack Compose, creating UI's is self explanatory, as CMP leverages its API. However, certain components and functionalities are only available on Android. This is either because they are Android-specific or because they have not yet been ported to other platforms. A list of Android-only components can be found in the official documentation [34]. If such a component is to be used, a developer can use the Android-only component for Android and implement the same functionality for the other platforms by using their respective native libraries.

10.2.2 Interrelation with Jetpack Compose

As previously mentioned, CMP and Jetpack Compose are highly similar as they use the same core concepts and APIs. However, CMP has certain restrictions regarding platform-specific features and differences compared to Jetpack Compose. The following list summarizes this, but is not exhaustive:

- Android-only components are not available for the common CMP code.
- Platform-specific APIs are only available on their respective platforms.
- To access resources such as images, fonts and strings, CMP uses the `Res` class from the Compose Multiplatform resources library. Jetpack compose on the other hand uses the `R` class from the Android resource system.
- The Maps Compose library [35] is not available for CMP as it is Android-specific.

One major difference is how dependencies are handled in the background when building an application. In CMP, when a build is done for Android, it uses the same dependencies as Jetpack Compose would. For example, if a project has `compose.material3` as a dependency, it uses `androidx.compose.material3:material3`. But if it is built for another target such as iOS, it uses `org.jetbrains.compose.material3:material3`. This does not require any configuration and is done automatically, utilizing Gradle Module Metadata within the multiplatform artifacts.

10.3 History

JetBrains' first steps into cross-platform development started with the KotlinConf 2017, where the "Kotlin Multiplatform Project" (KMP) was announced, introducing an experimental project that supported JVM, Android, iOS and JS targets.

On August 31, 2020, the first public alpha of "Kotlin Multiplatform Mobile" (KMM) was released, focusing only on mobile development, specifically Android and iOS. It

[3] As of May 6, 2025, the iOS platform has been marked Stable [5].

enabled the sharing of business logic between the two native platforms. This is possible due to JetBrains' other technology called Kotlin Native, which allows compilation of Kotlin code into native libraries [36].

This split between KMP and KMM led to naming confusion in the community, complicating content discovery and leading to a misunderstanding that Kotlin Multiplatform was primarily mobile-focused. Hence, on July 31, 2023, the decision was made to deprecate the "Kotlin Multiplatform Mobile" (KMM) naming and cover everything under "Kotlin Multiplatform" (KMP), regardless of the combination of targets [37].

Around the same time, the alpha version of Compose Multiplatform for iOS was released. Announced at KotlinConf 2023, Compose Multiplatform for iOS now enables iOS and Android to have shared UI code in addition to the business logic [38].

Since then many improvements have been made for KMP and CMP, even supporting the sharing of UI code with the web platform [39].

10.4 Concepts

This section introduces the core concepts that define Kotlin Multiplatform [40].

10.4.1 Common Code

As the name suggests, common code is the Kotlin code that is shared between the platforms. Typically located in the `commonMain` directory, the compiler takes the source code and produces platform-specific binaries.

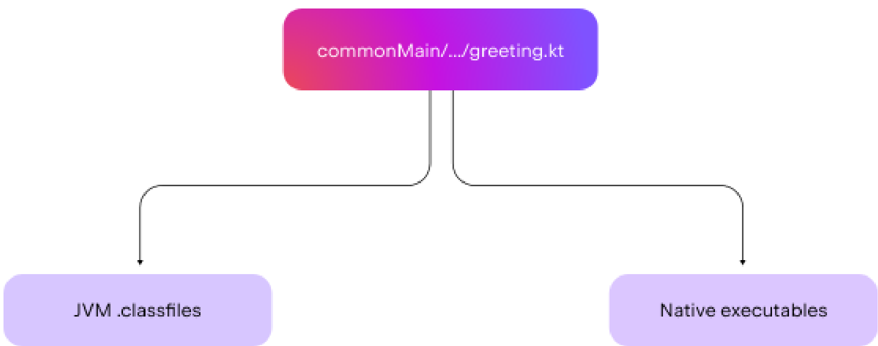


Figure 10.2: Common Code Compilation

Of course, not every Kotlin code can be compiled like this. Platform-specific code found in the `commonMain` directory will not compile. For example using the `java.util.UUID` package to generate a UUID would not be allowed, since it is a part of the JDK. Therefore, the native code compiled for other targets would not have access to those JDK classes.

Instead, Kotlin Multiplatform-specific libraries that support all targets, such as `kotlinx.coroutines`, can be used.

10.4.2 Targets

The concept of targets define which platforms the common code gets compiled to. In the example of Figure 10.2, the targets are JVM and native executables like iOS. The targets are declared in the `build.gradle.kts` file inside the directory containing the source sets, commonly named `shared` or `composeApp` if CMP is used, of your KMP project.

```
// shared/build.gradle.kts
kotlin{
    jvm()                   // Declares JVM target
    iosArm64()              // Declares 64-bit iPhone target
}
```

Listing 10.1: Target Declaration

These targets can be seen as labels that tell Kotlin how to compile the code, what kind of binaries to produce and which language constructions and dependencies are allowed.

10.4.3 Source Sets

A source set is a set of Kotlin source files that has its own dependencies, targets and compiler options. Source sets have the following properties:

- A unique name in the project.
- A set of Kotlin source files and resources, typically stored in the directory with the name of the source set.
- A set of compilation targets that determine which language constructions and dependencies are available in this source set.
- Defines its own dependencies and compiler options.

Kotlin provides some predefined source sets, like the `commonMain` source set for example. Typically the folder structure of the project would look like this:

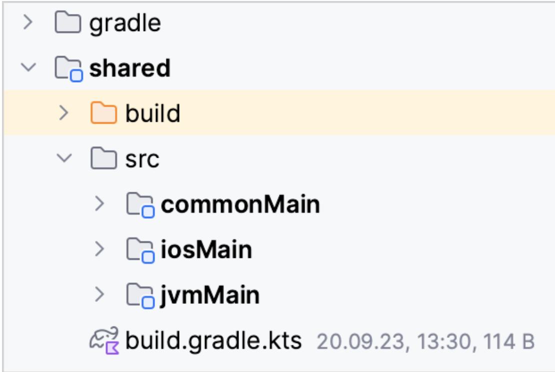


Figure 10.3: Folder Structure

Figure 10.3 shows the source sets `commonMain`, `iosMain` and `jvmMain`. The source sets can be configured in the same location as the target declaration in Listing 10.1.

```
// shared/build.gradle.kts
kotlin{
    jvm()                    // Declares JVM target
    iosArm64()               // Declares 64-bit iPhone target

    // Source Set declaration
    sourceSets{
        commonMain{
            // Configure this source set
        }
        ...
    }
}
```

Listing 10.2: Source Set Configuration

While `commonMain` handles all declared targets, every other source set is either platform-specific or an intermediary source set.

Platform-specific Source Set

As mentioned previously, common code or code inside the `commonMain` source set cannot contain any platform-specific API. If the project requires such APIs then they need to be put into platforms-specific source sets, also called platform source sets. Each target has a corresponding platform source set, which is responsible only for that target. Let's say the project targets Android as well as iOS and requires the generation of UUIDs. The listing below shows that the Android specific source set allows the usage of the JDK library, since it is only responsible for the Android target and compilation.

```kotlin
// commonMain/kotlin/common.kt
// Doesn't compile in common code because iOS does not support JDK
fun uuid() {
    val uuid = java.util.UUID.randomUUID()
}

// androidMain/kotlin/android.kt
// Works because code is in Android source set
// and Android target supports JDK libraries
fun androidUuid() {
    val uuid = java.util.UUID.randomUUID()
}
```

Listing 10.3: Platform-specific API Usage

Kotlin Multiplatform allows for compilation of a singular target, even with multiple source sets.

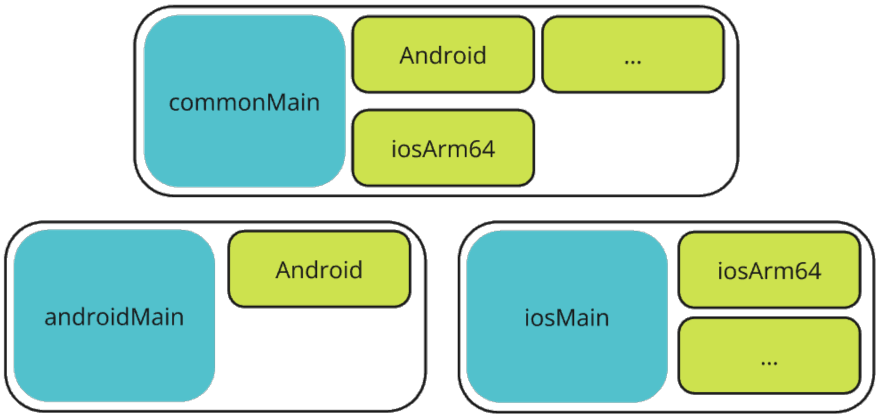


Figure 10.4: Source Sets and Targets

With a setup as depicted in Figure 10.4, the compilation target can be set to Android only. In that case `commonMain` and `androidMain` would be the only source files affected and they would be compiled together into an Android native binary. This binary would contain the declarations of both `commonMain` and `androidMain`.

Important to note is that platform-specific source sets can use the `commonMain` source set declarations but never the other way.

Intermediate Source Sets

In some cases, having only common code and platform-specific source sets is not enough or rather too inefficient. For example, if the project requires multiple Apple targets, there would be code duplications across the source sets such as `iosArm64`, `macosArm64` and `tvosArm64`. Using intermediate source sets, also called hierarchical source sets or simply hierarchies, solves this issue. This is because they enable the compilation of a selection of targets.

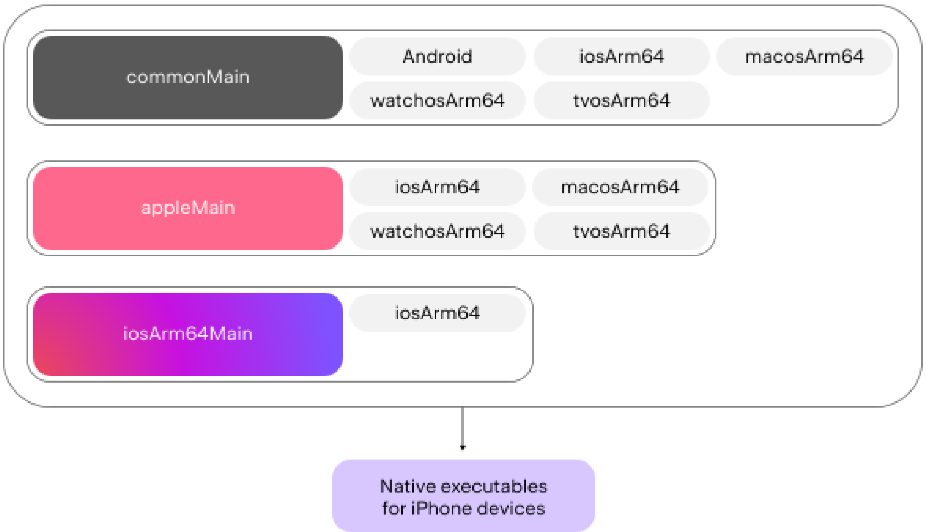


Figure 10.5: `appleMain` Source Set

As Figure 10.5 shows, the `appleMain` intermediate source set is used to cover all the Apple targets. This effectively reduces code duplications since the developer needs to write Apple-specific code only once inside `appleMain`. Then the target is set to `iosArm64` only and a native executable for iPhones is compiled, containing the declarations of `commomMain`, `appleMain` and `iosArm64Main`. Kotlin creates some commonly used intermediate source sets per default, like the `appleMain` for example. These predefined intermediate source sets are part of the default hierachy template:

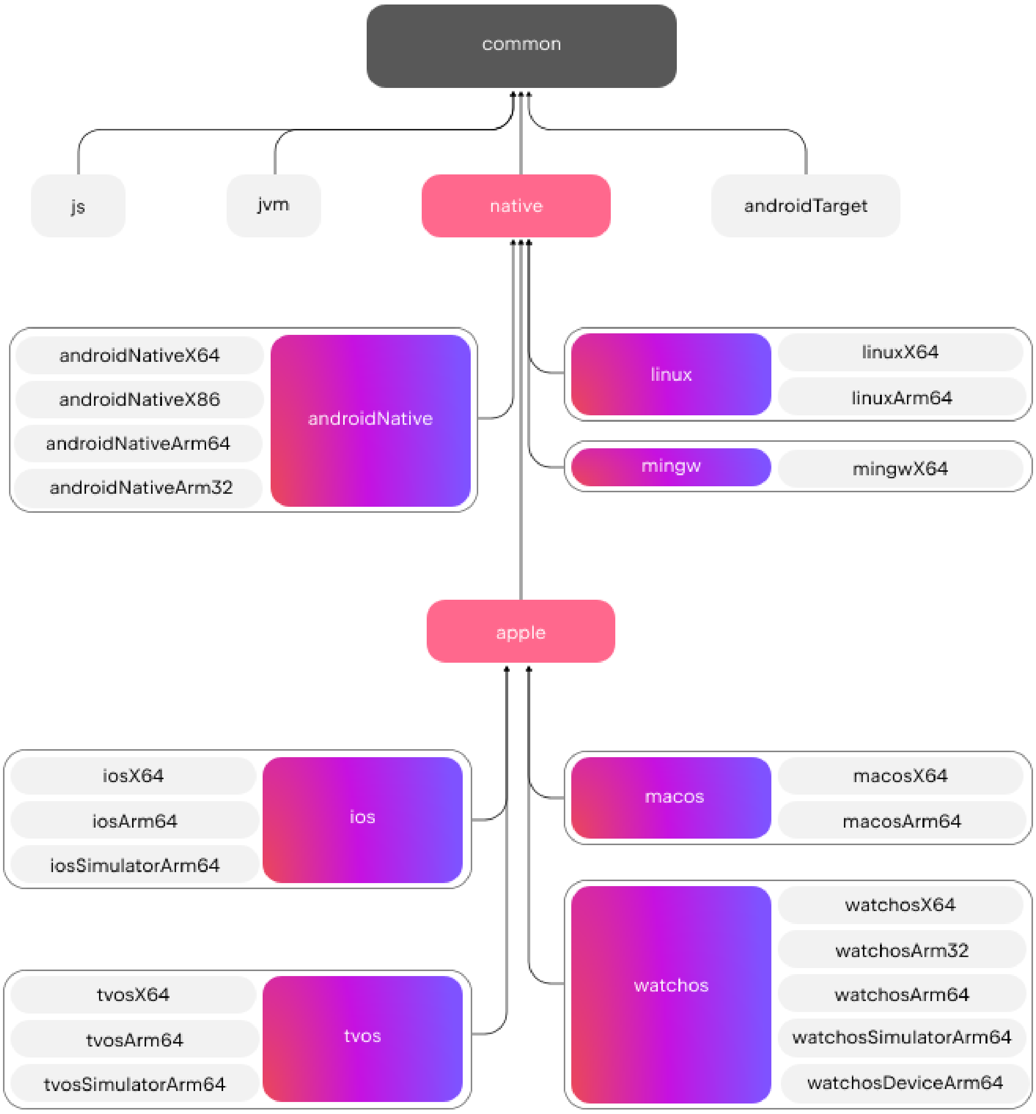


Figure 10.6: Full Hierarchy Template

It is important to note that the difference between `androidTarget` and `androidNative` is that `androidTarget` gets compiled into JVM bytecode, while `androidNative`, using the Kotlin/Native technology, is compiled without a virtual machine. Essentially, `androidTarget` is for the normal Android mobile apps, while `androidNative` is for writing Android NDK components.

10.4.4 Tests

For every `Main` source set, there is a `Test` counterpart. These can be common code tests inside `commonTest` or platform-specific tests inside `iosTest`, using the `XCTest` library for example. By default, a multiplatform test library called `kotlin.test` is provided, giving access to methods like `assertEquals`.

10.5 Sharing Code

To showcase how code is shared in practice, the base KMP project generated by the official KMP Wizard will be used as a template. The project targets Android and iOS with UI code sharing selected, meaning CMP is also employed. After opening the project, the directory structure looks like this:

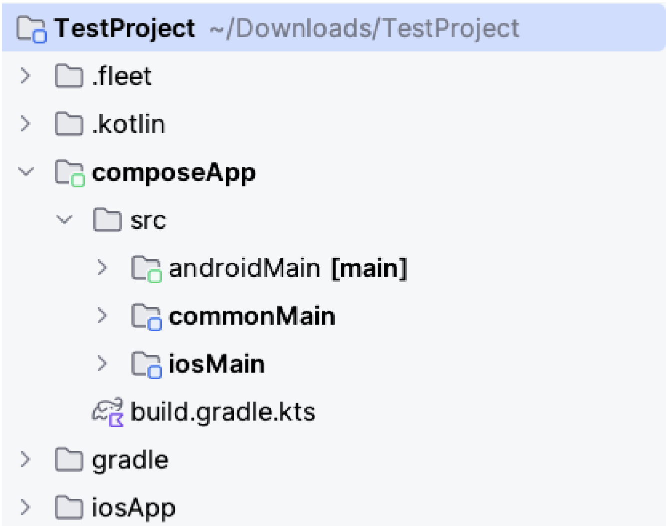


Figure 10.7: Test Project Structure

The project has a `composeApp` folder acting as the base directory for shared code. Inside there are three source sets: `androidMain`, `commonMain` and `iosMain`. Taking a look inside the `commonMain` folder, the file Platform.kt can be found:

```kotlin
// commonMain/kotlin/Platform.kt
interface Platform {
    val name: String
}

expect fun getPlatform(): Platform
```

Listing 10.4: Platform.kt

This file declares a common interface that can be used and implemented by all platform-specific source sets. It also declares a function with the `expect` keyword. The implementation of this function should then return an instance of type `Platform`.

expect/actual mechanism The `expect` keyword is part of the KMP expect/actual mechanism, allowing access to platform-specifc APIs. This mechanism can be used with functions, properties, classes, interfaces, enumerations, or annotations. The ”expected” construct can be used in common code without any actual implementation, as shown in the example. The implementation is then provided by the platform-specific source sets, marking them with the `actual` keyword. For this mechanism to work, the compiler checks that: [41]

- Every `expect` declaration in the common source set has a matching `actual` declaration in every platform-specific source set.
- Expected declarations do not contain any implementation.
- Every `actual` declaration shares the same package as the corresponding `expect` declaration, such as `org.example.project`.

The `actual` implementations can be found in the `androidMain` and `iosMain` source sets:

```kotlin
// iosMain/kotlin/Platform.ios.kt
import platform.UIKit.UIDevice

class IOSPlatform: Platform {
    override val name: String = UIDevice.currentDevice.systemName() + " " + UIDevice.currentDevice.systemVersion
}
actual fun getPlatform(): Platform = IOSPlatform()

// androidMain/kotlin/Platform.android.kt
import android.os.Build

class AndroidPlatform : Platform {
    override val name: String = "Android ${Build.VERSION.SDK_INT}"
}
actual fun getPlatform(): Platform = AndroidPlatform()
```

Listing 10.5: Platform-specific Implementations

As the code above shows, the expect/actual mechanism allows the usage of platform-specific APIs.

Sharing UI Code

Since the example project is using CMP to share UI code, a simple composable function `App()` can be found in the common code inside `commonMain/kotlin/App.kt`. This function builds the UI declaratively using CMP. It supports sharing composable UI code with iOS out-of-the-box, but the starting point must still be configured per platform.

```kotlin
// androidMain/kotlin/MainActivity.kt
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContent {
            App()   // Main composable function
        }
    }
}

//iosMain/kotlin/MainViewController.kt
fun MainViewController() = ComposeUIViewController {
    App() // Main composable function
}
```

Listing 10.6: Platform-specific UI Start

For the example project targeting iOS and Android, this means setting the composable `App()` function as the content for the Android `MainActivity`, and for iOS, setting the content for the `MainViewController`.

10.6 iOS Integration

This section explains the different approaches of iOS integrations in a KMP app, as well as the mechanics of the Kotlin-Swift interoperability.

As previously mentioned, source sets are a collection of Kotlin source files. The platform-specifc source sets, `androidMain` and `iosMain`, both only contain Kotlin code. The `androidMain` also contains the actual native Android application, specifically the `MainActivity.kt`, `AndroidManifest.xml` and `res` folder.

But the `iosMain` source set does not contain the native Swift application, but instead is responsible for providing easier access to Apple SDKs that have been translated to Kotlin. Examples of this are the prebuilt libraries used in Listing 10.5 through the expect/actual mechanism. KMP provides prebuilt libraries for basic Apple SDKs such as Foundation or Core Bluetooth [42].

The actual iOS application is inside the folder called `iosApp`, which resides on the same level as the `composeApp` folder that contains the source sets.

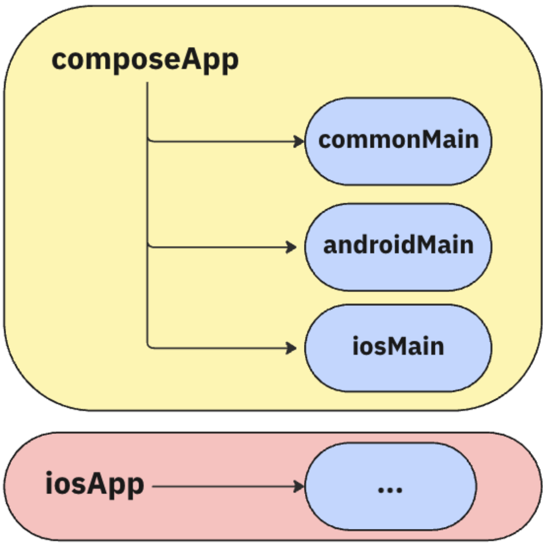


Figure 10.8: iOS App Location

As seen in Listing 10.6, in Android one can use the common code declarations akin to regular dependencies. In iOS the usage is the same. The `MainViewController` function defined in the Listing 10.6 is used in the actual iOS app Swift code as follows:

```swift
// iosApp/ContentView.swift
import ...
import ComposeApp // the shared KMP code gets imported here

struct ComposeView: UIViewControllerRepresentable {
    func makeUIViewController(
        context: Context
    ) -> UIViewController {
        // the MainViewController defined in iosMain
        MainViewControllerKt.MainViewController()
    }

    func updateUIViewController(
        _ uiViewController: UIViewController,
         context: Context
    ) {}
}

struct ContentView: View {
    var body: some View {
        ComposeView()
                .ignoresSafeArea(.keyboard)
    }
}
```

Listing 10.7: iOS App Setup

While at first glance it might seem as simple as the Android counterpart, with just an `import` statement for the ComposeApp, the situation is more complex for iOS. Swift code can call a Kotlin library because the Kotlin shared module is integrated into the iOS app as a native dependency, using one of several methods.

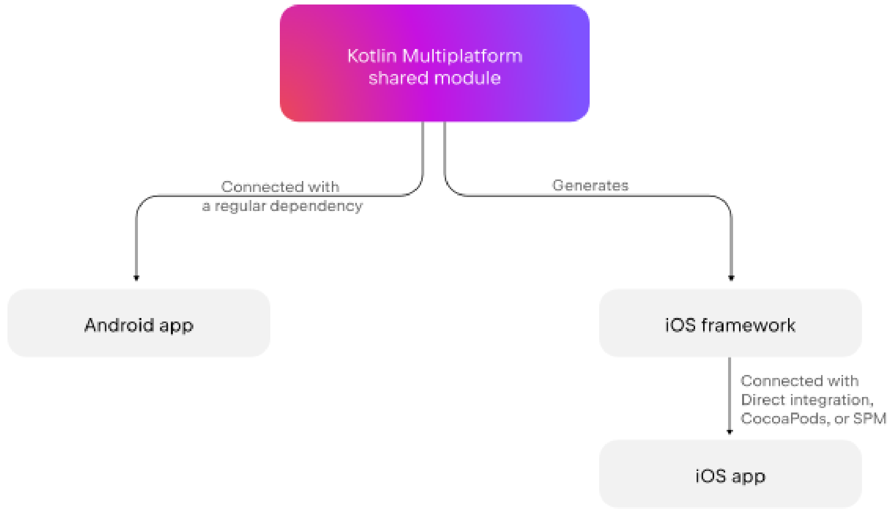


Figure 10.9: iOS Integration Scheme [43]

The above figure shows that the KMP shared module first gets generated into an iOS framework. This can then be connected to the iOS application, either through Direct Integration, CocoaPods or Swift Package Manager (SPM).

This dependency can be consumed locally or remotely. Local meaning that the developer will have full control over the code base and get instant updates for the native application when the common code changes. With remote integration, a developer can explicitly separate the code base of the native app and the common code, treating the dependency like other third-party libraries. Local integration is possible using Direct Integration, Cocoapod and SPM, while remote integration can be achieved only with SPM or CocoaPods. This paper will focus on local integration with Direct Integration and CocoaPods. Further details on remote and iOS integration are available in the following source: [43].

10.6.1 Direct Integration

This integration method is suitable for KMP projects with following characteristics:

- KMP project targeting iOS has already been set up
- KMP project does not have CocoaPods dependencies

With Direct Integration, the iOS framework gets generated using a script that is configured to run before the *Compile Sources* build step in Xcode. The script uses the Gradle task `embedAndSignAppleFrameworkForXcode` to generate the framework. The task does the following things:

- Copies the compiled Kotlin framework into the correct directory within the iOS project structure.
- Handles the code signing process of the embedded framework.
- Ensures that code changes in the Kotlin framework are reflected in the iOS app in Xcode.

Further information regarding Direct Integration, including a setup guide, can be found here: [44].

10.6.2 CocoaPods Integration

This integration method is suitable for KMP projects with following characteristics:

- Mono repository setup with an iOS project that uses CocoaPods.
- KMP project has CocoaPods dependencies.

Choosing to integrate using CocoaPods allows the iOS app to add dependencies to Pod libraries and integrates the KMP shared module as a Pod as well.

CocoaPods integration requires the development machine to have a local CocoaPods installation which in turn also requires Ruby to be installed as well. The biggest configuration change is the inclusion of CocoaPods inside the `build.gradle.kts` of the `composeApp` directory.

```
// composeApp/build.gradle.kts
plugins {
    ...
    kotlin("native.cocoapods") version "2.1.21"
    ...
}
kotlin {
    cocoapods {
        // Required properties
        // Specify the required Pod version here
        // Otherwise, the Gradle project version is used
        version = "1.0"
        summary = "Some description for a Kotlin/Native module"
        homepage = "Link to a Kotlin/Native module homepage"
        ...
    }
}
```

Listing 10.8: CocoaPods Configuration

The CocoaPod plugin creates the Gradle task `podspec` which generates a Podspec file for the KMP shared module. This Podspec file includes the script phases that build the iOS framework during the build process of an Xcode project. Additional information about CocoaPods integration can be found in following source: [45].

10.6.3 Kotlin-Swift/Objective-C Interoperability

The next point to address is how the Kotlin code gets translated into an iOS framework.

KMP was built on top of the Kotlin/Native technology, its purpose being to compile Kotlin code into native libraries [46]. Kotlin/Native supports Apple targets, for which Kotlin was made bidirectionaly interoperable with Objective-C [47]. Swift was not considered at the time, but can still be bridged if its API is exported to Objective-C with the `@objc` tag [42].

This technology is what enables KMP to be used together with iOS and is the backbone of the integrations.

10.7 iOS Dependencies

The described integration methods showcase how KMP code can be called in the native iOS application and how to use the prebuilt Apple SDKs in the `iosMain` source set. If the project requires native capabilites beyond the prebuilt libraries, there are several ways to make Swift/Objective-C code available to be used inside the Kotlin `iosMain` source set.

10.7.1 Cinterop

Cinterop is a tool that enables developers to generate Kotlin bindings for Objective-C or Swift libraries and frameworks. The tool leverages the Kotlin-Swift/Objective-C interoperability mentioned in subsection 10.6.3. While there are some differences between libraries and frameworks, in general these bindings can be generated through following steps [42]:

1. Download your dependency.
2. Build it to get its binaries.
3. Create a special .def definition file that describes this dependency to Cinterop.
4. Adjust your build script to generate bindings during the build.

10.7.2 CocoaPods

If Cocopods is the chosen iOS integration method, then iOS dependecies can be added through Pods [42].

```kotlin
// composeApp/build.gradle.kts
kotlin {
    cocoapods {
        version = "2.0"
        //..
        pod("SDWebImage") {
            version = "5.20.0"
        }
    }
}

// composeApp/iosMain/Example.kt
import cocoapods.SDWebImage.*
```

Listing 10.9: Pod Dependency

Developers may import Pods from various sources [48]:

- From the CocoaPods repository
- On a locally stored library
- From a custom Git repository
- From a custom Podspec repository

10.7.3 Dependency Injection

Another way to use iOS dependencies in Kotlin code is through dependency injection with frameworks like Koin. This allows for pure native implementations of interfaces defined in the common code, as seen in the example below by Pavel Puchkov.

```kotlin
// shared/src/commonMain/.../services/Analytic.kt
interface Analytic {
    fun logEvent(event: String)
}

// androidApp/src/main/.../services/AnalyticImpl.kt
class AnalyticImpl(private val logger: Logger): Analytic {
    override fun logEvent(event: String) {
        logger.log("...")
    }
}

// iosApp/iosApp/Services/AnalyticImpl.swift
class AnalyticImpl: Analytic {
    private let logger: Logger

    init(logger: Logger) {
        self.logger = logger
    }


```

```
        func logEvent(event: String) {
            logger.log(text: "...")
        }
    }
```

Listing 10.10: Dependency Injection Native Services

These implementations can then be provided to the dependency injection container and used in the common code.

Chapter 11

Comparison

This chapter compares Kotlin Multiplatform with native Android development and MAUI cross-platform development.

11.1 Native Android

Modern native Android development is Kotlin-first, as recommended by Google [49]. This naturally leads to overlaps with Kotlin Multiplatform due to having the same syntax. Although new concepts need to be applied, as explained in the previous chapter.

Regarding the limitations of native Android development in KMP, it actually does not limit it at all, since KMP supports all native libraries in the platform-specific source set with the expect/actual mechanism. However, using this mechanism means there are as many `actual` implementations as there are targets in the KMP project.

To avoid this issue, the code would need to be written in the common source set, but that is where stability problems can occur. For example the, in Android development popular, `ViewModel` package is still experimental when used in combination with Compose Multiplatform [50]. [4]

11.2 MAUI

MAUI, formerly known as Xamarin, is the cross-platform technology developed by Microsoft. Both KMP and MAUI try to solve the same problem, as do all cross-platform technologies: to reduce development time, reuse code and reach a wider audience [51].

Following is a comparison between these technologies:

[4] As of May 6, 2025, the ViewModel package has been marked Stable [5]

	KMP	**MAUI**
Released	August 2020 (KMM)	June 2004 (Mono) [52]
Programming Language	Kotlin	C#
Supported Platforms	• Android • iOS • macOS • Windows • Web • watchOS • tvOS	• Android • iOS • macOS • Windows [53]
Native Libraries Supported	Yes	Yes [54]
Shared UI Supported	• Android (Stable) • iOS (Beta)[4] • macOS (Stable) • Windows (Stable) • Web (Alpha)	• All stable

Table 11.1: KMP vs. MAUI

As the above table shows, MAUI is the more mature technology. MAUI started with the Mono open-source project that Xamarin was based on. The biggest difference is the programming language, with KMP using Kotlin and MAUI using C#. But since they're both object-oriented, the concepts remain largely the same.

In terms of supported platforms, KMP has the edge, being able to target Web, watchOS and tvOS as well. Native libraries or writing platform-specifc code is sup-

[4] As of May 6, 2025, the iOS platform has been marked Stable [5].

ported by both. In KMP, there are platform-specifc source sets with the expect/actual mechanism and in MAUI a similar approach with partial classes and methods exists [54].

Regarding shared UI code, KMP is lagging behind with its young and unstable Compose Multiplatform implementation, while on MAUI's side, the sharing of UI code has been deemed stable since the Xamarin days.

Chapter 12

In Practice

This chapter discusses the primary use cases for KMP and how companies are already adopting it. Furthermore, the community is analyzed and compared with other communities. Lastly, insights into KMP usage in the POSTE application is discussed.

12.1 Use Cases

KMP presents an approach to cross-platform development that enables code sharing while preserving platform-specific optimizations, or even allows complete native implementation of the UI. This section examines the primary use cases for implementing KMP.

12.1.1 Cross-Platform Mobile Application Development

The most prominent use case for KMP is mobile application development. KMP allows developers to share business logic, data handling and networking code between Android and iOS. Developers have two options for developing the UI. Either they use KMP in combination with Compose Multiplatform, while still allowing parts of the UI to be written natively. The other option is to completely implement the UI natively for all required platforms.

This approach allows teams to reduce development time and maintenance cost without compromising the user experience. Examples of companies that have already streamlined their mobile development processes with KMP can be found in section 12.2.

12.1.2 Unifying Business Logic

Organizations with established platform-specific codebases might want to utilize KMP to unify their codebase. KMP allows developers to choose the level of code-sharing. This would allow for incremental migration to a shared common codebase. A possible approach could look like this:

- Selective extraction of common components without full-scale rewriting

- Incremental adoption of cross-platform architecture
- Risk mitigation through phased implementation
- Preservation of platform-specific optimizations during transition

This approach enables organizations to realize cross-platform benefits while avoiding disruptive and extensive rewrites of functional applications.

12.2 Industry Adoption

Despite being fairly new in the market, KMP has already been adopted by several major companies. In other words, KMP is already built well enough that multi-billion dollar companies deem it worthy to invest large amounts of resources into building products with it.

12.2.1 Forbes

Forbes for example, decided to build their latest mobile app with KMP, with the aim to reduce development costs and time to market. They decided to build the UI natively for iOS and Android, using Kotlin for everything else. By doing so, they are able to deliver features faster and reduce cost. Furthermore, they use open-source libraries built by the community to further optimize costs. Although KMP offers lots of advantages, Forbes faced and identified some challenges. They find that [55]:

- most of the cross-platform libraries are not as mature as their native iOS/Android counterparts
- there are still some performance tradeoffs when compared to optimized native code
- the decision of when to use native or shared approaches has to be made yourself and can be tricky

12.2.2 McDonald's

To reduce the amount of code in their Global Mobile App codebases, McDonalds decided to use KMP (or then called KMM). They selected KMP because most of the other options are web-based solutions. These need another translation layer, which results in worse performance and thus mediocre experience for the user. In McDonald's opinion, KMP is best suited for projects that are designed around dependency injection and/or clean architecture.

McDonald's serves locales worldwide, with lots of them having unique requirements and different menus. Using KMP they now have all the business, parsing and storage logic in the same place for both platforms.

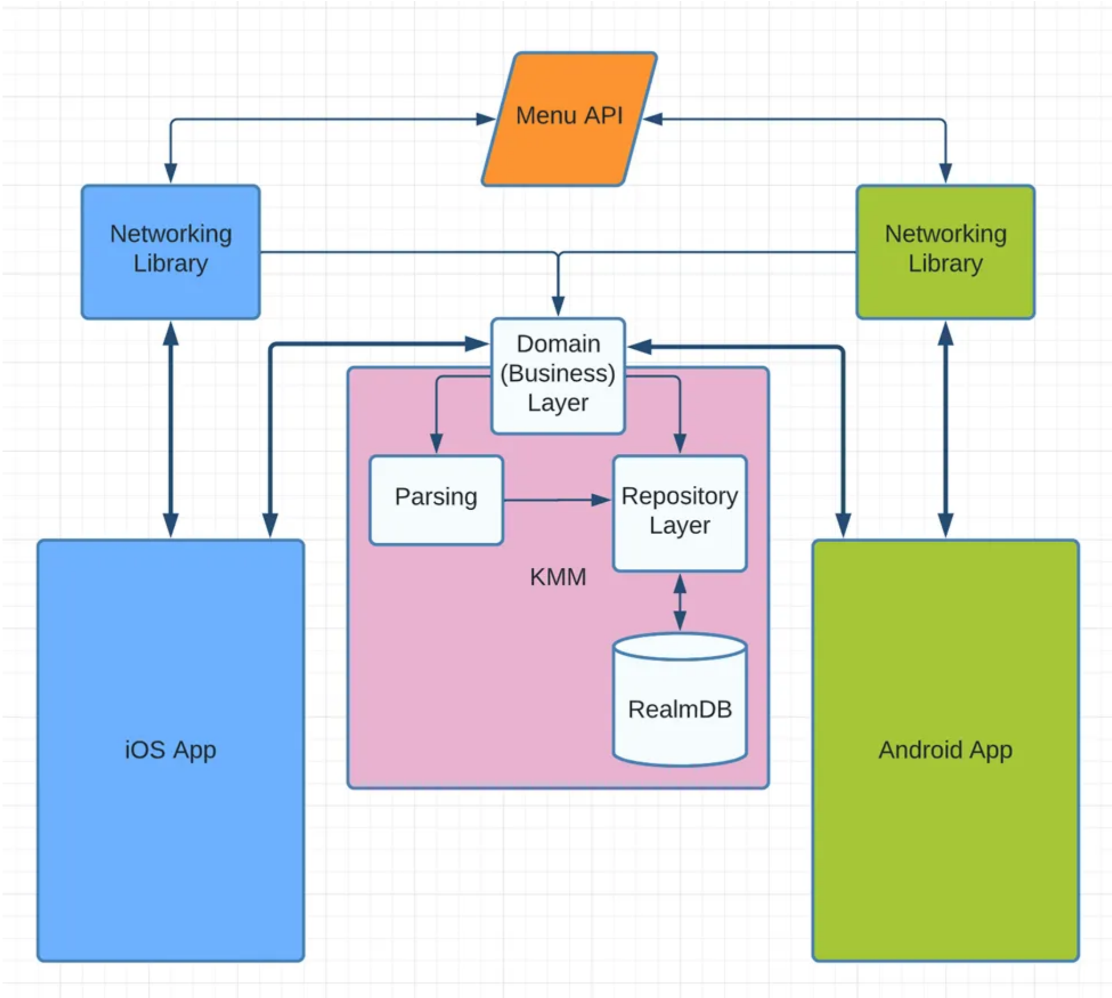


Figure 12.1: McDonald's Architecture Diagram [56]

With this architecture, as shown in Figure 12.1 they were able to:

- reduce the time needed to develop a feature
- minimize the time needed by a developer to understand and integrate the required logic
- reduce the test burden for developers as unit tests are handled for both platforms simultaneously
- run full end-to-end testing on the business logic and database without having to build the entire application

In contrast to this, McDonald's also ran into some challenges. The biggest one is, that not all libraries have direct Kotlin equivalents. This can be solved using Kotlin's expect/actual paradigm for platform-specific code [56].

12.3 Community

Kotlin Multiplatform has been growing since 2017 and is becoming increasingly popular. As a result, a community of developers, organizations and open-source contributors has cultivated around it, fostering its growth and adoption.

While precise metrics regarding community size remain difficult to find, analysis of GitHub repositories, conference attendance and forum activity indicate a steady growth, particularly following Google's endorsement of Kotlin for Android. Nevertheless, the community is still smaller than those of established cross-platform frameworks such as Flutter or .NET MAUI, but it is experiencing increased adoption among enterprises and independent developers. The Kotlinlang Slack workspace with over 60'000 members has active discussions in dedicated KMP channels [57]. In addition, KMP-related discussions and knowledge sharing also take place on platforms such as GitHub, Stack Overflow, Reddit and Kotlin's official forums.

The KMP community actively contributes to the ecosystem by developing open-source libraries and tools. A curated list of libraries and tools can be found on the *Awesome Kotlin Multiplatform* repository [58]. Furthermore, the Kotlin web team has created a website containing all libraries compatible with KMP [59].

12.4 Kotlin Multiplatform in POSTE

This section presents how KMP is used to build the cross-platform mobile app POSTE, the student marketplace app, and which libraries are employed to complete the product.

The POSTE application shares 100% of its code and is targeting Android and iOS platforms. This means the UI code, as well as business logic and data handling are all written in Kotlin inside the common code.

12.4.1 Integration Method

KMP integration with the iOS application is handled through CocoaPods. This required setting up CocoaPods on the development machines and writing specific scripts in the Codemagic build pipeline. The reason for this choice was the ability to use Pod dependencies when required.

12.4.2 UI Code

Since the code is fully shared in POSTE, the Compose Multiplatform framework is used to write the UI Code. Supporting this decision is the Voyager library, providing multiplatform navigation and an alternative to, at the time of construction, experimental `ViewModel` and `Navigation` API of CMP.

Additionally, Coil and FileKit are also added to handle async web images and local file handling for both platforms.

12.4.3 Data Handling

To facilitate calls to the backend and the internet in general, JetBrains' own Ktor packages are added. Ktor provides a multiplatform HTTP client, enabling POSTE to send and receive requests easily.

12.4.4 Persistency

The app requires the storing of local key-value pairs, for which the DataStore library is used. The, in Android development popular, package supports multiplatform usage, assisted by the expect/actual mechanism [60].

12.4.5 Dependency Injection

The application is split into a Data, DI and UI module with a uni-directional dataflow. The dependency injection framework Koin is leveraged to provide instances of Data classes to the UI. Especially significant is the use of dependency injection to enable native SDK usage in POSTE. Specifically the libraries MSAL for authentication and PubNub for chat functionality. Following is the simplified code for MSAL in POSTE.

```
// commonMain/.../di/msal/MsalCommon.kt
interface MsalCommon {
    suspend fun signIn()
    suspend fun signOut()
    ...
}

// androidMain/.../di/MsalAndroid.kt
import com.microsoft.identity.client.*

class MsalAndroid(
    private val activity: Activity,
) : MsalCommon {
    ...
    override suspend fun signIn() {...}
    override suspend fun signOut() {...}
    ...
}

// iosApp/MsalImpl.swift
import ComposeApp
import MSAL
import UIKit

class MsalImpl: MsalCommon {
    ...
    func signIn(){...}
    func signOut(){...}
    ...
}
```

Listing 12.1: MSAL Implementation

The code below demonstrates how the implementations of `MsalCommon` and `PubNubCommon` are then injected into the dependency container.

```
// commonMain/.../di/KoinHelper.kt
class KoinHelper {
    ...
    fun initKoin(
        pubnub: PubNubCommon,
        msal: MsalCommon,
    ) {...}
    ...
}

// Android, onCreate
KoinHelper().initKoin(
    PubNubAndroid(),
    MsalAndroid(this)
)

// iOS, init of iOSApp
KoinHelper().initKoin(
    PubNubImpl(),
    MsalImpl(parentViewController: rootVC)
)
```

Listing 12.2: Native Service with DI

12.4.6 Local Pod

The POSTE app has unique requirements regarding native keyboard behaviour. To meet those requirements a local CocoaPods Pod is developed and used as a dependency using the CocoaPods integration.

```
// iosHelperPod/src
@objc public class KeyboardObserver: NSObject {
    @objc public func observeKeyboardState(
        onKeyboardStateChanged: @escaping (Bool) -> Void
        ) { ...}
    ...
}

//composeApp/build.gradle.kts
cocoapods {
    ...
    pod("ios_helper") {
        version = "1.0"
        source = path(project.file("../iosHelperPod"))
        extraOpts += listOf("-compiler-option", "-fmodules")
    }
    ...
}


```

```
//iosApp/Podfile
target 'iosApp' do
    use_frameworks!
    platform :ios, '16.0'
    pod 'composeApp', :path => '../composeApp'
    pod 'ios_helper', :path => '../iosHelperPod'
end
```

Listing 12.3: Local Pod `ios_helper`

This pod can then easily be used in Kotlin code through an `import` statement. The CocoaPods integration together with Kotlin/Native interoperability automatically generates Kotlin bindings, since the Swift code is marked with `@objc`.

12.4.7 UI/Integration Testing

Additionally POSTE has a requirement for E2E testing, which is met through UI testing in the `commonTest` source set. UI testing is possible through the same test libraries as Jetpack Compose.

Chapter 13

Evaluation and Outlook

This chapter evaluates the Kotlin Multiplatform technologies and looks ahead into the future of KMP.

13.1 Features

From a theoretical standpoint, Kotlin Multiplatform offers all the features one needs to be able to develop cross-platform mobile applications.

- The ability to share UI code through CMP
- Easy access to native SDKs through the intuitive expect/actual mechanism or dependency injection if bigger services are needed
- The flexibility regarding iOS integration with CocoaPods or SPM depending the requirements of the app
- The ability to only share specific logic

Especially the last point makes KMP attractive for companies meaning to adapt the technology. The reason being that they can integrate it step-by-step instead of a complete rewrite or migration of the current product.

13.2 Industry

The research of the industry adoption of KMP revealed satisfactory results. With big international companies such as McDonald's and Forbes already using the technology in production, providing millions with their services, the argument can be made that KMP is already ripe enough. Important to highlight is that both companies use KMP for business logic while their UIs remain native.

13.3 POSTE Experience

The POSTE app has been developed as a measure of how viable a productive, fully shared KMP app can be.

Early hurdles were experienced due to, the then still experimental, `ViewModel` and `Navigation` APIs. This lead to the third-party Voyager library being used as a replacement, which fulfilled its purpose as a multiplatform navigation library. The choice of CocoaPods integration was also met with a fair share of troubles, mainly caused by the inexperience of the development team regarding CocoaPods and iOS development in general. But the CocoaPods integration setup also could have been simpler. A bug was also encountered while writing Compose UI tests, which broke iOS tests completely. Luckily, a fellow KMP developer had encountered this bug before and opened an issue in JetBrains' YouTrack before and provided a workaround. The POSTE development team left a comment to confirm this bug. Another small caveat for the contemporary developer is that due to its young age, the AI chat agents such as ChatGPT by OpenAI and LeChat by Mistral simply lack the dataset to provide consistent responses for questions regarding KMP.

Despite such problems, as a development team experienced in Jetpack Compose, writing CMP UI code for POSTE was fast and went without any problems. The developer experience between regular Compose and CMP is practically identical, even to the point where it was possible to port UI code from a previous native Android app directly into POSTE. The use of native SDKs in POSTE was made easy by using Koin and dependency injection to provide native implementations. To be able to use local pods for small pieces of Swift code, such as native keyboard handling, has been a positive experience. For most other Apple SDKs the prebuilt libraries in combination with the expect/actual mechanism was also simple to use.

13.4 Final Assessment

The industry already has adopted KMP as a productive technology in use cases where business logic must be shared. Regarding 100% shared KMP apps, while the development of POSTE was not without any problems, most of the issues did not come from KMP itself, but the team's inexperience with Swift, CocoaPods and iOS development. The UI code was written quickly and there are many ways to use native SDKs. Most issues that are encountered can be googled and there are plenty of articles with code examples. The YouTrack can also be a valuable source of information for niche bugs.

As of May 6, 2025 KMP version 1.8.0 was released and with it the iOS platform and various APIs such as `ViewModel` and `Navigation` have been marked as stable. This means that cross-platform mobile development with KMP is now fully stable. This showcases the continued support of JetBrains for KMP.

With the new release and the aforementioned points, the development team deems the Kotlin Multiplatform technology viable for use in production.

13.5 Future

With the iOS being stable now, JetBrains will look to towards their other platforms that still remain in Alpha and Beta. This includes Web (Kotlin/Wasm), watchOS and tvOS. For CMP, Web (Kotlin/Wasm) is still in Alpha. In the future, once all platforms are stable, it will probably be possible to write the application only once for Android, iOS, Desktop and Web.

The 1.8.0 release also had an impact on the community. With a stable iOS platform more independent developers will be convinced to write third-party libraries, further improving and expanding the KMP ecosystem.

All in all, the new release has been a huge confidence boost for people still on the fence about this technology and its future as a whole stays promising.

Part VII

Project Documentation

Chapter 14

Project Plan

This chapter addresses the planning aspect of the project and the tools used.

14.1 Planning

This section focuses on short- and long-term planning, as well as methodologies, responsibilities and the identified risks.

14.1.1 Methodology

The chosen methodology for this project is Scrum+. This OST-original methodology combines the flexibility of Scrum with the stability of RUP. Scrum is used for short-term planning and RUP for long-term planning. With a sprint length of two weeks, the project will have eight sprints in total. These sprints are planned and allocated into the four RUP phases.

14.1.2 Roles and Responsibility

Following are the role assignments and the respective responsibilities that come with them. While the responsibilities are clear, the work can be shared and assistance is provided when needed.

Scrum Master Simon Peier
Responsibility: Leading Scrum meetings, documenting short-term plans

Project Manager: Roger Marty
Responsibility: Writing meeting minutes

Product Owner: Tseten Emjee
Responsibility: Leading refinement meetings and sanitation of the backlog

DevOps: Roger Marty
Responsibility: Overview and general responsibility over pipelines and infrastructure

Frontend: Simon Peier
Responsibility: Overview and general responsibility over the mobile app

Backend: Roger Marty
Responsibility: Overview and general responsibility over the backend including external systems

Testing: Tseten Emjee
Responsibility: Overview and general responsibility over testing

Architecture: Tseten Emjee
Responsibility: Overview and general responsibility over the architecture

14.1.3 Meetings

These are the regularly held meetings of the project and their timeslots:

- **Sprint Planning / Review**: Every two weeks on Monday 10h00 - 11h00
- **Refinement Meeting**: Every second Monday in Sprint 10h00 - 11h00
- **Weekly Sync with Advisor**: Every Monday 11h00 - 12h00

14.1.4 Long-Term Plan

The following graphic shows the long-term plan with the planned phases, durations and milestones. The defined durations and work segments are estimations and may change throughout the project's duration.

Figure 14.1: Long-Term Plan

14.1.5 Milestones

These are the five milestones that mark the key deliverables and completion of phases:

Milestone 01: Requirements 17.03.2025

The first milestone is to establish all requirements for the product. This includes mandatory requirements for the MVP as well as optional ones.

Planned Deliverables:

- Define functional requirements
- Define non-functional requirements
- Create a domain model consistent with the defined requirements

Milestone 02: Prototype 31.03.2025

This milestone focuses on completing the prototype to ensure that the chosen technologies are able to work together.

Planned Deliverables:

- Prototype App
  - Specification: An Android and iOS app that integrates the cloud-based backend and external systems like PubNub and Firebase.

Milestone 03: Beta 19.05.2025

With the Beta milestone, the construction phase is nearing its end and the focus lies on bug-fixes and implementing final changes.

Planned Deliverables:

- Beta App
  - Specification: Near feature-complete with bugs and few features remaining.

Milestone 04: MVP 02.06.2025

This milestone signifies the completion of the MVP and the end of the construction phase.

Planned Deliverables:

- MVP App
  - Specifications: Feature-complete with almost no bugs remaining.

Milestone 05: Final Submission 13.06.2025
The final milestone ensures everything is ready for submission.

Planned Deliverables:

- Abstract for brochure
- Poster for bachelor thesis exhibition
- Final Product
  - Specifications: MVP with no bugs and maybe optional features included.
- Final Documentation
  - Specifications: Complete and submission-ready documentation, proof-read by all team members.

14.1.6 Short-Term Plans

The short-term plans are the separate sprints of the project. They are planned and estimated following Scrum methodology and planning poker. Following is a history of all sprints and short descriptions of the work done.

Sprint 1

Product	Documentation
- Setup various tools	- Complete project planning (long-term plan, milestones, risks,...)
	- Prepare quality measures

Table 14.1: Short-Term Plan Sprint 1

Sprint 2

Product	Documentation
- Create low-fidelity mockups	- Create C4 diagrams, domain model, ERD and other diagrams
- Make technology decisions	- Define FRs/NFRs
	- Research KMP, cloud, authentication

Table 14.2: Short-Term Plan Sprint 2

Sprint 3

Product	Documentation
- Design high-fidelity mockups	- Create high-level sequence diagrams
- Create app logo	
- Setup DEV environments	
- Configure CI/CD pipelines	
- Make working prototype	

Table 14.3: Short-Term Plan Sprint 3

Sprint 4

Product	Documentation
- Create AWS infrastructure	- Define API endpoints
- Composables created	- Review risks
- Setup navigation	- Make interim presentation
- MS Entra ID integration	

Table 14.4: Short-Term Plan Sprint 4

Sprint 5

Product	Documentation
- Implement user, listing and image endpoints	- Adjust certain endpoints
- Create login page	
- Integrate material theme	
- Create various screen models and UIs	

Table 14.5: Short-Term Plan Sprint 5

Sprint 6

Product	**Documentation**
- Implement all remaining endpoints	- Various small adjustments
- Implement create/detail listing page	
- Implement overview page	
- Implement watchlist page	
- Implement search	
- Implement settings page	

Table 14.6: Short-Term Plan Sprint 6

Sprint 7

Product	**Documentation**
- Implement chat functionality	- Design payment integration concept
- Implement guest login for Apple reviews	- Write abstract
- Set up instrumented and integration tests	- Write management summary
- Conduct usability tests	
- Make adjustments based on testing feedback	

Table 14.7: Short-Term Plan Sprint 7

Sprint 8

Product	Documentation
- Conduct usability tests	- Rework and improve KMP research chapter
- Implement test feedback improvements	- Write implementation chapter
- Improve chat and notification functionality	- Document results and review chapter
- Prepare production data	- Finish remaining sections
- Submit App Store release	- Review and improve documentation

Table 14.8: Short-Term Plan Sprint 8

14.1.7 Risk Management

This section focuses on the project risks that exist and categorizes them according to their probability and severity. Prevention and mitigation strategies are outlined as well. The aim is to bring all risks below the acceptance line, i.e. into the green and yellow zone of the risk matrix.

Changes to risks during the course of the project are recorded in the changelog.

Identified Risks

ID	Description
Technical Risks	
R1	Loss of project data / code
R2	Unauthorized use of interfaces
R3	Unauthorized access to user data
R4	Performance issues
R5	Inadequate code quality
R6	Incompatible technologies
R7	Kotlin Multiplatform is inadequate
Project Risks	
R8	Misjudgment of the time schedule
R9	Absence of team member
R10	Lack of team communication
R11	Insufficient knowledge of technologies
R12	Outage of project relevant tools

Table 14.9: Identified Risks

Risk Matrix

Probabilities	Severity			
	Negligible	Marginal	Critical	Catastrophic
High				
Likely				
Possible		R8		
Unlikely		R1, R10	R9	
Rare	R5		R2, R4, R7, R11	R3, R6, R12

Table 14.10: Risk Matrix

Risk Handling

ID	Prevention	Mitigation
Technical Risks		
R1	Commit often, local backups	Restore from local backup
R2	Do not expose any secrets, keys and ports	Refresh / deactivate keys and secrets
R3	Implement authentication/authorization, encrypt data at rest and in transit	Deactivate user, rotate tokens
R4	Clean coding, well planned architecture	Review code and architecture
R5	Coding guidelines, linters, pipelines and pre-commit hooks	Refactoring
R6	Initial research of technologies	Rapid change of technologies
R7	Evaluate current status of Kotlin Multiplatform	Choose an alternative
Project Risks		
R8	Time buffer in planning, review progress of sprints	Adjust scope, prioritize core-functions
R9	Eat healthy, stay fit	Smooth transfer of knowledge and tasks
R10	Sync often, work together	Extraordinary meetings
R11	Sufficient time for research	Check documentation, ask in forums or discuss with advisor
R12	Check for alternatives	Quickly switch to alternatives

Table 14.11: Risk Handling

Risk Changelog

Date	Risk	Change	Cause
01.04.2025	R2	Critical/Unlikely to Critical/Rare	Azure Authentication implemented with JWT
01.04.2025	R6	Catastrophic/Possible to Catastrophic/Unlikely	Prototype completion
01.04.2025	R7	Catastrophic/Unlikely to Critical/Unlikely	Prototype completion
01.04.2025	R8	Critical/Possible to Marginal/Possible	Passed critical project phase
01.04.2025	R11	Critical/Possible to Critical/Unlikely	Knowledge gained through prototype completion
20.05.2025	R6	Catastrophic/Unlikely to Catastrophic/Rare	Proven compatibility through beta release
20.05.2025	R7	Critical/Unlikely to Critical/Rare	CMP 1.8.0 release marks iOS as stable
04.06.2025	R11	Critical/Unlikely to Critical/Rare	Experience gained with PubNub on chat feature

Table 14.12: Risk Changelog

14.2 Tooling

This section highlights the tools that are used in this project.

14.2.1 Documentation

The documentation is built with LaTeX and versioned with Git. The remote repository is on GitLab where the configured pipeline builds the output PDF on every commit. This enables version control and allows for rolling back to a previous version if needed.

To ensure compliance with the OST guidelines and requirements, the LaTeX template from the module SE Project is used.

14.2.2 Code

The codebase is divided into frontend, backend and cloud-related code/configuration. Each of these components has its own repository in the GitLab group. This clear separation ensures better organization and maintainability, with each repository having its own pipeline for automatic checks, builds and tests.

14.2.3 Tracking

Jira is used to track sprints, epics, issues and bugs:
(Jira Board)

Clockify is used to track the working hours of each member. Clockify is a time tracking software which integrates with Jira and is able to generate time reports:
(Clockify Tracker)

14.2.4 Workflow

The steps that each issue goes through are illustrated in the workflow diagram below. Initially, every issue has the status "To Do". As soon as a team member starts working on an issue, it is moved to "In Progress" and remains there until the implementation is complete. Upon completion, the issue transitions to "In Review" where another team member reviews the work. When the implementation is acceptable and the review is successfully completed, the issue can be marked as "Done". In case that the implementation is not acceptable, the issue is moved back to "In Progress".

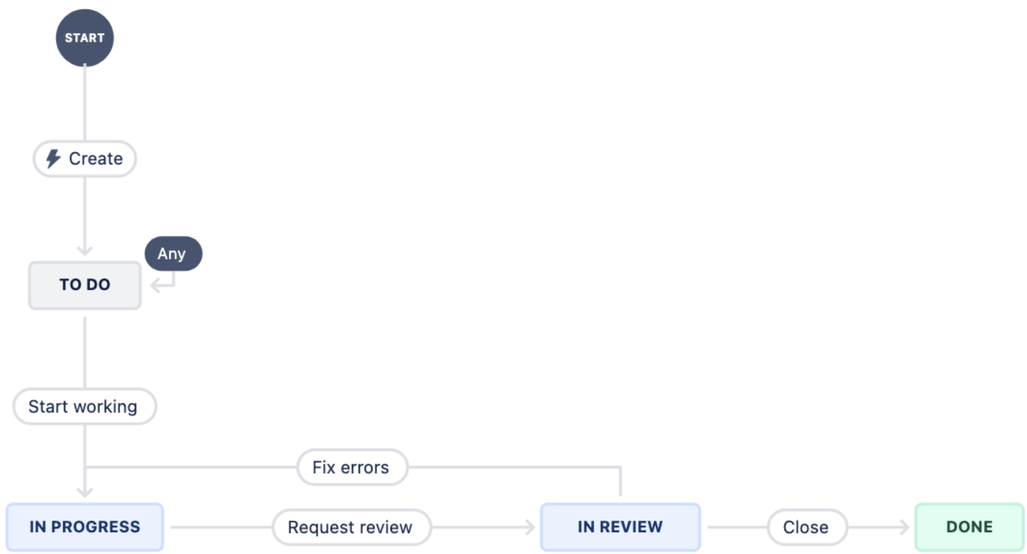


Figure 14.2: Jira Workflow

14.2.5 Tool and Resource Directory

The following table lists the tools and resources used.

Task Area	Tools
Literature Research and Management	Google, LLM[5], IEEEtran
Idea Generation	LLM[5], Miro
Translation	DeepL, Google Translate
Designs	Figma, Balsamiq, Miro, Canva
Coding	Visual Studio Code, Android Studio, Xcode, LLM[5]
Text Creation, Editing, Spelling and Grammar Checking	Latex, DeepL, LLM[5]
Collaboration and Project Management	MS Teams, Jira, Clockify, Miro, Outlook, GitLab, Notion, WhatsApp
DevOps	GitLab, Codemagic, AWS, OpenTofu
Version Control and Code Collaboration	GitLab

Table 14.13: Tool and Resource Directory

Use of AI Tools

The usage of AI is acknowledged in this thesis. These tools are used to gather ideas, refine grammar and sentence structure and assist in solving certain coding problems due to their efficiency and widespread application in the IT domain.

All AI-generated content undergoes a critical review before being used in this project due to the potential for errors.

[5]Depending on the case different LLMs were used including ChatGPT, Claude and Le Chat.

Chapter 15

Quality Measures

This chapter covers all methods and tools used to guarantee good quality of the produced increments and the final product.

15.1 Code

Every project containing code uses linters for static analysis of the code. This reveals programming errors, bugs and stylistic problems. Code formatters are configured for all projects to automatically format the code according to guidelines and alert if something is not up to standard.

To enforce these guidelines as well as good code quality, the linters and formatters run locally as pre-commit hooks. The linters run again in the repository pipelines for each commit.

	Frontend (KMP)	**Backend (Python)**
Linter	ktlint [61]	ruff [62]
Formatter	ktlint [61]	ruff [62]

Table 15.1: Code Quality Tools

15.2 Gitflow

To simplify the scheduled releases during this thesis, the Gitflow strategy is used across all repositories. Feature branches are created from the develop branch and when finished merged back into the develop branch. For a release, the develop branch is merged back into the main branch.

Each merge request requires a review and approval of another developer as well as successful pipelines before they can be confirmed.

Branch	Description
Main Branch	Always contains the latest stable release of the software which is used in production environments.
Develop Branch	Branched from the main branch in the beginning. If a new release approaches and the develop branch is stable and pipelines are successful, it gets merged into the main branch.
Feature Branch	Every issue that gets handled receives a separate feature branch, branched off from the latest develop branch. Naming convention: feature/[issueID]-[issueTitle]
Bugfix Branch	Every bug that needs to be fixed requires a separate bugfix branch either from the latest develop branch or directly from the main branch depending on the severity. Naming convention: bugfix/[issueID]-[issueTitle]

Table 15.2: Gitflow Branch Description

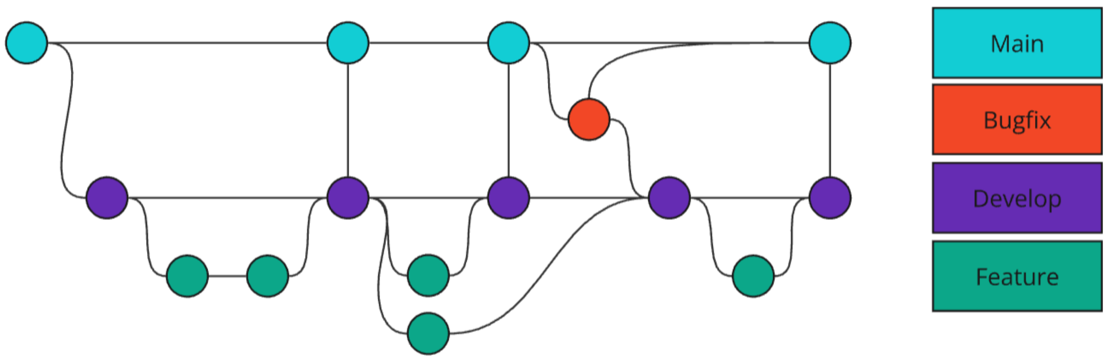


Figure 15.1: Gitflow Workflow

15.3 DoR / DoD

This section covers the Definition of Ready (DoR) and Definition of Done (DoD) related to issues in the Scrum environment.

Definition of Ready	Definition of Done
• Common understanding of the issue • Acceptance criteria are defined • Estimated effort is determined • Can be completed in one sprint	• Acceptance criteria are implemented • Code adheres to standards and guidelines • New code is tested and passing • Pipelines are successful • Work is reviewed by another developer • Documentation is updated if necessary

Table 15.3: DoR / DoD

15.4 Metrics

Code metrics help with providing insight into the code that is written and can even improve its quality. However, having too many metrics can be distracting and require too much effort in this time-limited project. This is why the test coverage is the only code metric used across the various codebases.

Tests ensure that the code functions as expected and continues to do so after changes. For each commit, the test coverage is calculated in the CI/CD process and shown to the developers in the merge requests. They are able to see the change in coverage compared to the last merge and also its history. Through this, the developers are reminded to write tests. No specific coverage target percentage is set.

The actual results can be found in subsection 16.3.1.

15.5 Testing

This section focuses on the testing methods that are used in the frontend and backend.

15.5.1 Frontend

Multiple testing variants are used to validate the mobile app.

Unit Tests

Testing begins with unit tests, where specific code sections are tested in isolation to ensure they function correctly. Unit test were not the focus of this thesis and as such only sample tests to confirm KMP functionality are used.

Instrumented Tests

Instrumented tests verifiy the behavior of the UI on emulated devices. Connecting these tests to a local backend instance facilitates the end-to-end validation as required by the task description.

Usability Tests

Usability tests are conducted to gather real-world feedback about the app. After the beta and MVP release, each member of the group selects a student as a testee, as they are the target group of POSTE. The tester and testee then work through the usability testing protocol in a one on one session. The testee has to go through all scenarios and think out loud, while the tester makes notes. At the end there are also general feedback questions. The protocol and the results can be found in Appendix B.

All findings are collected and discussed in a meeting where each possible change is prioritized. To conduct a usability test, following elements are required:

- Team member
- Test participant, must be a student
- Test smartphone containing the latest stable release of the app connected to the production system
- Prepared and populated production environment
- Private area
- Internet

15.5.2 Backend

The backend utilizes integration tests to ensure that all layers and components work together correctly. Unit tests were considered initially but were removed, as their main benefits would have been limited to the service layer. Integration tests were deemed more appropriate, as they also include the router and repository/client logic.

15.6 Pipelines

Each repository in the project GitLab group has its own pipeline. These pipelines run on every commit to the repository and before/after each merge request. All repositories containing code run at least a linter, tests and calculate the metrics. Depending on the repository there are additional stages for deployment or security.

The exact implementation of the pipelines is described in chapter 6.

Chapter 16

Project Monitoring

This chapter focuses on the tracking and evaluation of the project progress.

16.1 Time Tracking Reports

Jira is used to keep track of tasks in the form of issues and combined with Clockify to track the time spent on them per project member. These tools enable the export of following detailed time reports:

- **Total Time** (Link)
- **Sprint 1** (Link)
- **Sprint 2** (Link)
- **Sprint 3** (Link)
- **Sprint 4** (Link)
- **Sprint 5** (Link)
- **Sprint 6** (Link)
- **Sprint 7** (Link)
- **Sprint 8** (Link)

16.2 Time Tracking Statistics

This section shows statistics and reports about the logged working hours. The total available time is 1'080 hours, meaning 360 hours per team member. The spent hours must not exceed 1'296 hours.

16.2.1 Work Distribution

The below pie chart shows the total hours each team member has accumulated throughout the thesis.

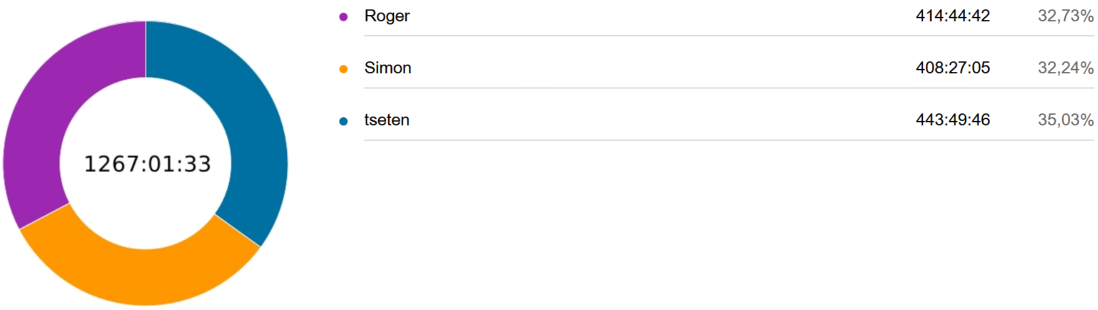


Figure 16.1: Work Distribution

16.2.2 Work History

The following graph shows the progression of the hours logged.

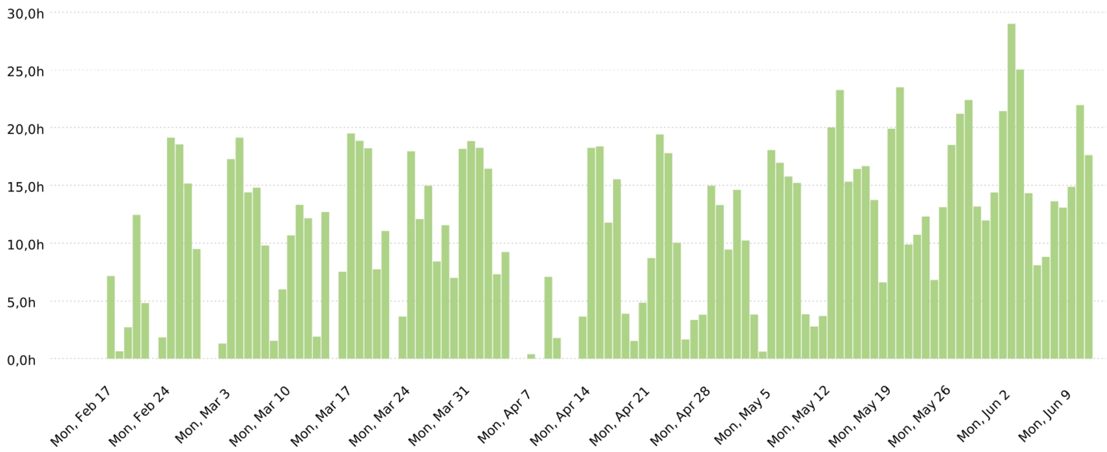


Figure 16.2: Work History

16.2.3 Overview Epics

At the beginning of the thesis, epics were defined, to which all issues are assigned accordingly. The following illustration shows the amount of assigned issues each epic has.

Even though this does not represent the workload of an epic, it still shows where most of the work was done.

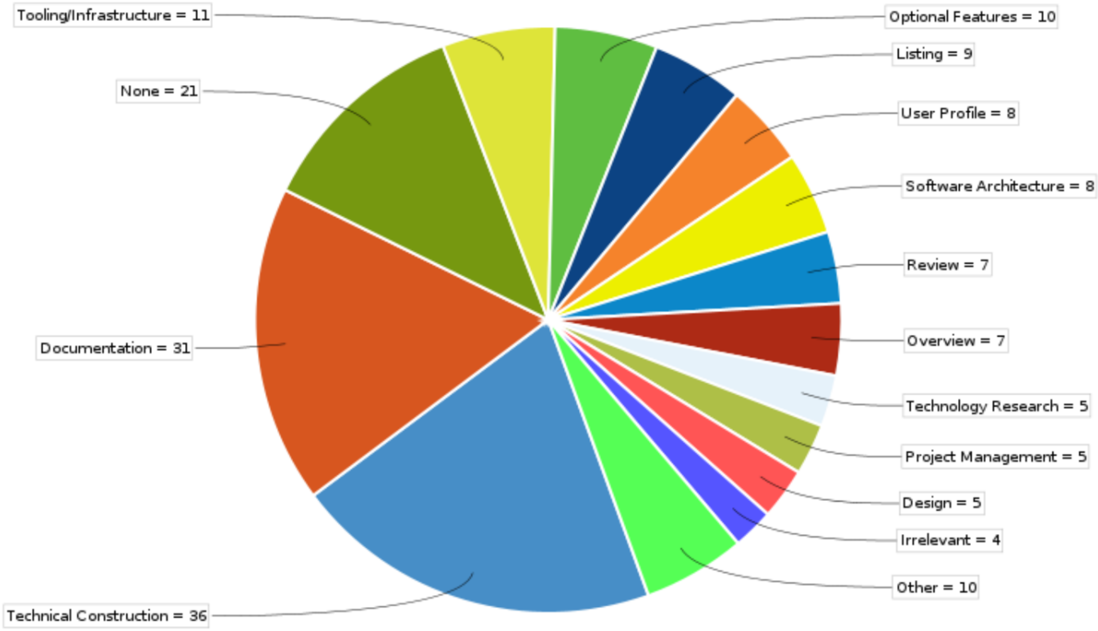


Figure 16.3: Overview Epics

16.2.4 Project Timeline

Due to assigning all issues to their respective epic, it is possible to track when an epic was worked on, as seen below. This can then be compared to the long term plan which shows the similarities between the planned and actual progress.

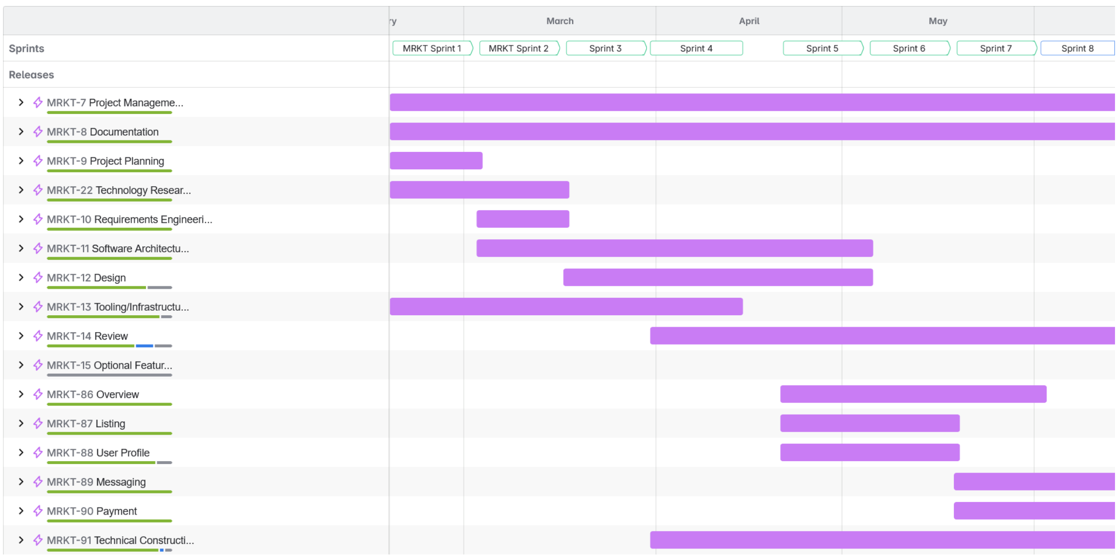


Figure 16.4: Project Timeline

16.2.5 Milestone Fullfilment

At the beginning of the thesis five milestones were defined. They represent important steps in the project, such as the MVP release for example. It can be said that all milestones were achieved on time and no delays occured.

16.3 Repository Analytics

The following sections display important and interesting statistics regarding the project repositories.

16.3.1 Test Coverage

The test coverage is the only metric in this project that is measured and monitored. No specific target percentage is defined, but the information is still shown to the developers.

The frontend test coverage does not include instrumented tests.

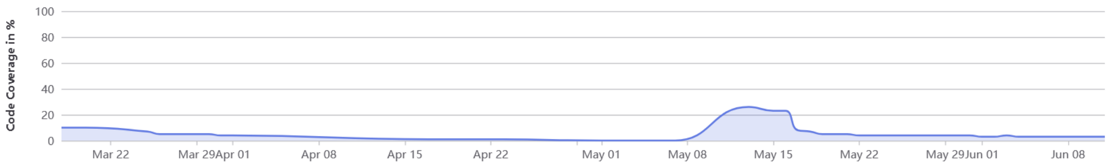


Figure 16.5: Frontend Test Coverage

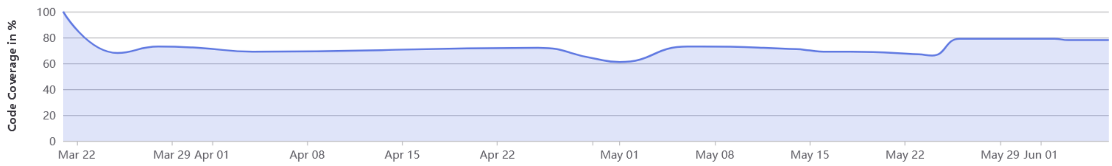


Figure 16.6: Backend Test Coverage

16.3.2 Commits

Following figures show the commit history of the various repositories. The two main repositories, frontend and backend, have all commits spread out throughout the construction phase. The infrastructure commits mainly happened in the beginning and stopped once the infrastructure was stable.

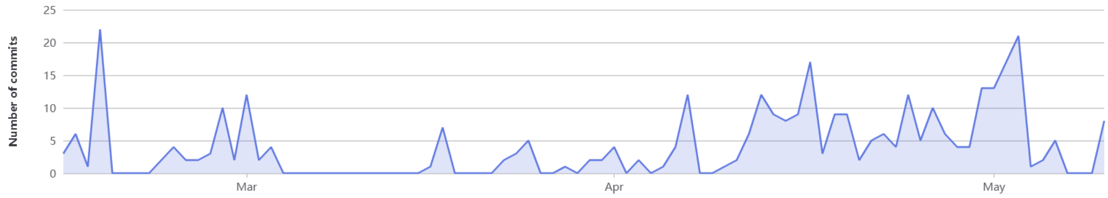


Figure 16.7: Frontend Commit History

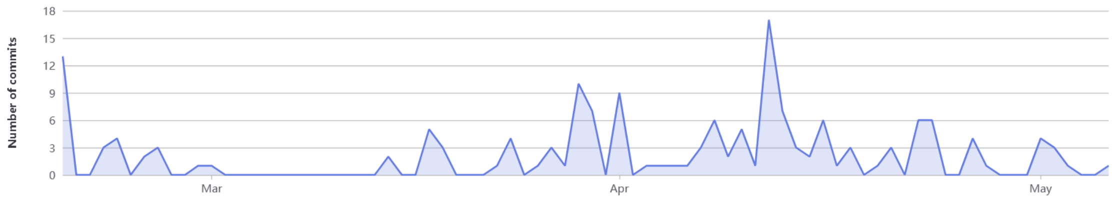


Figure 16.8: Backend Commit History

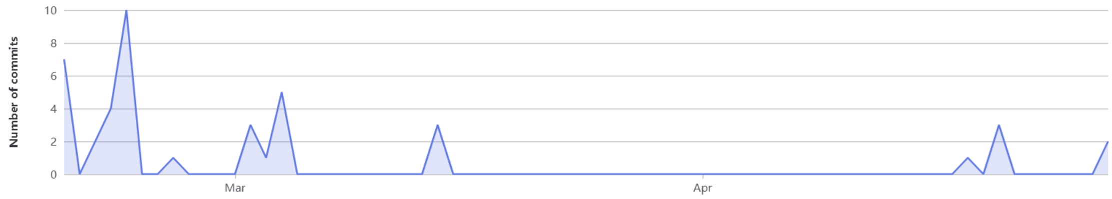


Figure 16.9: Infrastructure Commit History

Part VIII

Closing Thoughts

Chapter 17

Conclusion

The conclusion first discusses our opinion and experience on using KMP and PubNub. Then the success of this thesis is evaluated and finally possible progressions of the application are discussed.

17.1 KMP

This thesis allowed us to evaluate the maturity of KMP and CMP and gain deep insights, firstly by researching and secondly hands-on when implementing the POSTE app.

The development of POSTE with KMP and CMP was not without problems, but most of the issues were due to the team's inexperience with iOS development. Although certain APIs and libraries of KMP were still in beta while developing, we were always able to solve the problems. Either through the use of alternative libraries or by searching the internet for solutions.

On May 6th 2025, KMP version 1.8.0 was released, which marked the iOS platform and various APIs, such as `ViewModel` and `Navigation`, as stable.

We believe that the future of KMP looks promising. JetBrains' approach to let developers choose the extent of code sharing also enables companies to gradually transition to KMP. As more and more companies start to use it, more features and libraries will be available and still existing platform incompatibilities will be rectified. Otherwise, still existing incompatibilities can always be bypassed by implementing affected parts natively.

17.2 Chat Implementation with PubNub

Implementing the chat functionality with PubNub turned out to be more challenging than initially expected. PubNub offers two different SDKs: the standard PubNub SDK and the PubNub Chat SDK. Since the Chat SDK includes prebuilt functionality tailored for chat applications, we chose it to streamline development.

However, we soon encountered limitations. We could have used the standard SDK, which is highly flexible and allows full customization, but this requires developers to build all chat features from scratch. With the limited scope of this project, this was not feasible.

Another challenge was the lack of a multiplatform Chat SDK. As a result, we had to implement the chat features natively using the Swift Chat SDK for iOS and the Kotlin Chat SDK for Android. Although both SDKs offer the same core functionality, they differ significantly in how they are used. The Kotlin SDK relies on a callback-based approach, which led to deeply nested and difficult-to-maintain code. In contrast, Swift's use of async/await resulted in a much cleaner and more readable implementation, which we found to be easier to understand.

We also noticed that the documentation is, in some parts, incomplete and we had to find solutions elsewhere. Worth mentioning is the PubNub AI, which consistently provided usable answers to our questions.

On a last note, the chats feel slower than expected. While it is possible that the issue lies within our implementation, we largely followed the recommended guidelines and best practices, making us believe the performance issue might not be on our end.

For future projects, we would opt for a better alternative.

17.3 Evaluating Success

The objective of this project was to create a solution that provides students and employees of the OST with a marketplace app to buy and sell goods which are not used anymore. Furthermore, KMP was to be used to provide users with an Android and iOS app and to evaluate the maturity of the technology.

Based on this goal, 18 use cases and one precondition were defined, alongside 12 NFRs to ensure quality. For the MVP, use cases UC1 to UC15 were mandatory, as well as the precondition. The NFRs all had to be validated and accepted. The MVP was defined, so that upon meeting all requirements, the goal and aim of this project would be achieved.

Furthermore, extensive research into KMP was conducted before starting development, to ensure that it is sufficient for our requirements.

Reviewing the documented results, the project was successful. All MVP requirements have been met and some additional features have also been implemented to provide users with a better product.

17.4 Future and Outlook

With completing the necessary use cases and some additional features, one could think the application is complete. This is not the case. As with everything, there always is room for improvement. The optional use cases "UC16 Report User", "UC 17 Bidding Process" and "UC18 Set Listing Alerts" remain open. In addition, several ideas on how to improve the application even further came up during the course of the project. The list below provides an overview of said ideas:

- UC16: Allow users to report other users to POSTE. This would allow for moderation of misbehaving users.
- UC17: Implement functionality to allow a bidding process on listings.
- UC18: Allow the user to save searches and receive alerts for new listings that match them.
- Allow users to delete their account via settings.
- Add camera functionality on listing creation, meaning that users could directly take pictures of their article when creating a listing.
- Divide the chat overview into two sections, one for buying items and one for selling.

Chapter 18

Personal Reports

Below are the personal reports of each team member.

18.1 Roger Marty

Looking back, this bachelor thesis was a success in my opinion. The collaboration and communication between all group members was solid and no problems occurred in the group, as was the case in previous projects.

What I really liked was the assignment of the various tasks according to our strengths and interests. As I like working in the backend, as well as with cloud infrastructure and DevOps related things, I was able to take on all those responsibilities. Meanwhile Tseten and Simon could fully focus on the mobile app part.

But still there are certain areas that could be improved. The main one would be to define the DoD more strictly and in detail. Some implementations and issues were complex and consisted of a lot of logic. As a result, sometimes only the main component was built, after which the issue was marked as completed. Later, it turned out that certain minor details were overlooked. These then had to be fixed, which cost extra time.

In the middle of the construction phase there was also a problem with Meilisearch. Their AMI was used to set up and configure the search engine. However, suddenly the AMI was not available anymore. I opened a thread in their Discord and an issue on GitHub was also created. After a few days they were able to fix the problem and the AMI was available again. But sadly there were some problems with the image and it could not be used. Luckily, we still had an instance running and just had to make sure that it does not get deprovisioned by IaC changes.

In the early implementation phases it sometimes happened that the frontend had to wait for the backend endpoints. This is not something unusual when developing the frontend and backend in parallel, but I quickly saw that we have to do something about that. To combat this, I started to define all API contracts as early as possible, so the frontend could mock and already implement them.

Personally, I feel like the time estimation, which was an area to improve in the last project, got quite a bit better. Many issues were completed in the estimated time frame.

Even though my main focus was not on the frontend, I still participated in it and it was really interesting to work with a relatively young technology such as KMP. It revealed unexpected challenges, that may not be apparent in the beginning.

Of course another highlight was to work so much with AWS and it was an important learning experience for me.

18.2 Simon Peier

This thesis marks my first experience using cross-platform technology to develop mobile applications. Having previous experience in developing Android apps using Jetpack Compose, I found the transition to KMP to be relatively straightforward.

In the course of the project, we encountered several challenges. Most of these challenges were related to getting code to work on iOS as well. For example, when I implemented the chat using PubNub, I had to ensure that the Swift class properly aligned with the Kotlin interface it inherited from. I am proud to say that, despite these hurdles, we were able to successfully overcome all technical challenges.

As we already have successfully worked on previous projects together, communication and collaboration went smoothly. We were able to plan tasks according to personal strengths and interests, with Roger working on the backend and cloud infrastructure, while Tseten and me concentrated on the frontend development. Being able to focus on the frontend provided an excellent opportunity for me to improve my skills and broaden my skillset, particularly since most of my previous development experience was in backend technologies.

My personal highlight was implementing the search page with its filter and sorting capabilities. I was positively surprised how fast the search, using Meilisearch in the background, returned results.

Overall, I consider this bachelor thesis highly successful and the final product remarkably impressive, especially considering our time constraints. The project provided extensive learning opportunities, both theoretical and practical, and contributed significantly to my professional development.

18.3 Tseten Emjee

This bachelor thesis has personally been quite a challenge. I was confident in my mobile development abilites, having experience in native Android development and cross-platform development with Xamarin/MAUI as well. But learning KMP, a new technology, on the go while developing a decently sized app at the same time really pushed my limits.

There have been many fails and oversights during development, especially while writing the Swift-only iOS implementations for MSAL and PubNub. But while it may sound strange, I like to fail. That is where I can improve the most. Some of the big lessons include the realisation that we were developing a multiplatform app, meaning that of course the app needs to work on both platforms. This may seem obvious but in the mids of battle it is easy to just keep coding while confirming that the code works only on one platform and then getting surprised that it breaks on the other. There is a balance to be striked between coding and testing on both platforms.

Regarding project and time management, I have realised that I did not really consider the time it took to integrate a completed feature into the rest of the application, resulting in issues being judged too optimistically.

Overall, I really enjoyed working on this bachelor thesis. Having chosen the topic ourselves was also a big motivation factor. I am happy to have learned another way to develop cross-platform mobile apps and deepen my understanding of iOS, Swift and CocoaPods.

Chapter 19

Note of Thanks

We would like to express our deepest gratitude to our advisor, Martin Seelhofer, for his time and effort. In addition to the term project, we were also fortunate enough to carry out this bachelor thesis with him. Together, we were able to develop and refine the idea behind this work.

Weekly meetings were conducted which helped us when challenges or questions arised. But even outside of meetings we always had the opportunity to contact him regarding urgent matters and received quick responses.

We would like to extend our thanks to our co-examiner Paul Sevinç and proofreader Mirko Stocker. During the interim presentations they provided us with insightful questions and constructive feedback.

Part IX

Lists

Bibliography

[1] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[2] ISO25000, "ISO/IEC 25010," https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?linkId=100000045879485&utm_pview=8, 2022, [Online; accessed 04-March-2025].

[3] ADR, "About MADR," https://adr.github.io/madr/#example, 2025, [Online; accessed 08-March-2025].

[4] JetBrains, "Create a multiplatform app using Ktor and SQLDelight," https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-ktor-sqldelight.html#create-a-project, 2025, [Online; accessed 10-June-2025].

[5] E. Petrova, "Compose Multiplatform 1.8.0 Released: Compose Multiplatform for iOS Is Stable and Production-Ready," https://blog.jetbrains.com/kotlin/2025/05/compose-multiplatform-1-8-0-released-compose-multiplatform-for-ios-is-stable-and-production-ready/, 2025, [Online; accessed 07-June-2025].

[6] Google, "Guide to app architecture," https://developer.android.com/topic/architecture/, 2025, [Online; accessed 28-May-2025].

[7] I. Carrión, "Exploring App Architectures in Kotlin," https://carrion.dev/en/posts/app-architecture/, 2025, [Online; accessed 04-June-2025].

[8] V. Tarasov, "KMP: Modular Architecture," https://medium.com/@vptarasov/kmp-modular-architecture-faabbdf58197, 2024, [Online; accessed 04-June-2025].

[9] Y. Zhanymkanov, "FastAPI Best Practices," https://github.com/zhanymkanov/fastapi-best-practices, 2025, [Online; accessed 04-June-2025].

[10] Netflix, "dispatch," https://github.com/Netflix/dispatch, 2025, [Online; accessed 04-June-2025].

[11] K. Włodarczyk, "Fast API — Repository Pattern and Service Layer," https://medium.com/@kacperwlodarczyk/fast-api-repository-pattern-and-service-layer-dad43354f07a, 2025, [Online; accessed 04-June-2025].

[12] A. W. Services, "Amazon ECS networking best practices," https://docs.aws.amazon.com/AmazonECS/latest/developerguide/networking-best-practices.html, 2025, [Online; accessed 12-June-2025].

[13] Stripe, "Stripe Connect," https://stripe.com/en-ch/connect, 2025, [Online; accessed 28-May-2025].

[14] ——, "Stripe Connect Documentation," https://docs.stripe.com/connect, 2025, [Online; accessed 28-May-2025].

[15] Adyen, "Adyen for Platforms," https://www.adyen.com/platform-payments, 2025, [Online; accessed 28-May-2025].

[16] ——, "Adyen for Platforms Documentation," https://docs.adyen.com/adyen-for-platforms-model/, 2025, [Online; accessed 28-May-2025].

[17] PayPal, "PayPal for Marketplaces," https://www.paypal.com/us/enterprise/industry-solutions/platforms-and-marketplaces, 2025, [Online; accessed 28-May-2025].

[18] ——, "PayPal for Marketplaces Documentation," https://developer.paypal.com/docs/multiparty/, 2025, [Online; accessed 28-May-2025].

[19] Figma, "Figma," https://www.figma.com/, 2025, [Online; accessed 16-April-2025].

[20] "Material Theme Builder Plugin," https://www.figma.com/community/plugin/1034969338659738588/material-theme-builder, 2025, [Online; accessed 16-April-2025].

[21] Jetbrains, "Stability of supported platforms," https://www.jetbrains.com/help/kotlin-multiplatform-dev/supported-platforms.html#compose-multiplatform-ui-framework-stability-levels, 2024, [Online; accessed 04-March-2025].

[22] JetBrains and K. Foundation, "Testing Compose Multiplatform UI," https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-test.html, 2025, [Online; accessed 05-June-2025].

[23] Pydantic, "Pydantic," https://docs.pydantic.dev/latest/, 2025, [Online; accessed 05-June-2025].

[24] M. Bayer, "The Python SQL Toolkit and Object Relational Mapper," https://www.sqlalchemy.org/, 2025, [Online; accessed 07-June-2025].

[25] A. W. Services, "Boto3 - The AWS SDK for Python," https://github.com/boto/boto3, 2025, [Online; accessed 05-June-2025].

[26] Meilisearch, "Meilisearch Python," https://github.com/meilisearch/meilisearch-python, 2025, [Online; accessed 05-June-2025].

[27] Firebase, "Firebase Admin Python SDK," https://github.com/firebase/firebase-admin-python, 2025, [Online; accessed 05-June-2025].

[28] C. Paroz, "Python library to generate Swiss QR-bills," https://github.com/claudep/swiss-qr-bill, 2025, [Online; accessed 05-June-2025].

[29] holger krekel and pytest-dev team, "pytest: helps you write better programs," https://docs.pytest.org/en/stable/, 2025, [Online; accessed 04-June-2025].

[30] L. Projects, "Open-Source Infrastructure as Code," https://opentofu.org/, 2025, [Online; accessed 06-June-2025].

[31] JetBrains, "Kotlin Multiplatform Development," https://www.jetbrains.com/help/kotlin-multiplatform-dev/get-started.html, 2025, [Online; accessed 04-March-2025].

[32] ——, "Stability of supported platforms," https://www.jetbrains.com/help/kotlin-multiplatform-dev/supported-platforms.html, 2024, [Online; accessed 04-March-2025].

[33] ——, "Kotlin multiplatform," https://www.jetbrains.com/kotlin-multiplatform/, 2025, [Online; accessed 04-March-2025].

[34] Jetbrains, "Android-only components," https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-android-only-components.html, 2024, [Online; accessed 04-March-2025].

[35] Google, "Maps Compse Library," https://developers.google.com/maps/documentation/android-sdk/maps-compose, 2025, [Online; accessed 05-March-2025].

[36] E. Petrova, "Kotlin Multiplatform Mobile Goes Alpha," https://blog.jetbrains.com/kotlin/2020/08/kotlin-multiplatform-mobile-goes-alpha/, 2020, [Online; accessed 04-March-2025].

[37] ——, "Update on the Name of Kotlin Multiplatform," https://blog.jetbrains.com/kotlin/2023/07/update-on-the-name-of-kotlin-multiplatform/, 2023, [Online; accessed 04-March-2025].

[38] S. Agner, "Compose Multiplatform for iOS Is in Alpha," https://blog.jetbrains.com/kotlin/2023/05/compose-multiplatform-for-ios-is-in-alpha/, 2023, [Online; accessed 04-March-2025].

[39] A. Zamulla, "Compose Multiplatform 1.6.10 - iOS Beta, Web Alpha, Lifecycle, Navigation, and More," https://blog.jetbrains.com/kotlin/2024/05/compose-multiplatform-1-6-10-ios-beta/, 2024, [Online; accessed 04-March-2025].

[40] JetBrains, "The basics of Kotlin Multiplatform project structure," https://kotlinlang.org/docs/multiplatform-discover-project.html, 2025, [Online; accessed 05-March-2025].

[41] ——, "Expected and actual declarations," https://kotlinlang.org/docs/multiplatform-expect-actual.html, 2024, [Online; accessed 06-March-2025].

[42] ——, "Adding iOS dependencies," https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-ios-dependencies.html, 2025, [Online; accessed 07-June-2025].

[43] ——, "iOS integration methods," https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-ios-integration-overview.html, 2025, [Online; accessed 08-June-2025].

[44] ——, "Direct integration," https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-direct-integration.html, 2025, [Online; accessed 08-June-2025].

[45] ——, "CocoaPods overview and setup," https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-cocoapods-overview.html, 2025, [Online; accessed 08-June-2025].

[46] ——, "Kotlin/Native," https://kotlinlang.org/docs/native-overview.html, 2025, [Online; accessed 08-June-2025].

[47] ——, "Kotlin-Swift interopedia," https://github.com/kotlin-hands-on/kotlin-swift-interopedia, 2025, [Online; accessed 08-June-2025].

[48] ——, "Add dependencies on a Pod library," https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-cocoapods-libraries.html, 2025, [Online; accessed 08-June-2025].

[49] Google, "Android's Kotlin-first approach," https://developer.android.com/kotlin/first, 2025, [Online; accessed 12-June-2025].

[50] JetBrains, "Common ViewModel," https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-viewmodel.html, 2024, [Online; accessed 09-March-2025].

[51] ——, "What is cross-platform mobile development?" https://www.jetbrains.com/help/kotlin-multiplatform-dev/cross-platform-mobile-development.html#is-cross-platform-mobile-development-right-for-you, 2025, [Online; accessed 09-March-2025].

[52] J. O. Castro, "OSS .NET implementation Mono 1.0 released ," https://arstechnica.com/uncategorized/2004/06/3949-2/, 2004, [Online; accessed 09-March-2025].

[53] Microsoft, "What is .NET MAUI?" https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-9.0, 2025, [Online; accessed 09-March-2025].

[54] ——, “Invoke platform code,” https://learn.microsoft.com/en-us/dotnet/maui/platform-integration/invoke-platform-code?view=net-maui-9.0, 2022, [Online; accessed 09-March-2025].

[55] C. King, “Forbes Mobile App Shifts To Kotlin Multiplatform,” https://www.forbes.com/sites/forbes-engineering/2023/11/13/forbes-mobile-app-shifts-to-kotlin-multiplatform/, 2023, [Online; accessed 09-March-2025].

[56] McDonalds, “Mobile multiplatform development at McDonald’s,” https://medium.com/mcdonalds-technical-blog/mobile-multiplatform-development-at-mcdonalds-3b72c8d44ebc, 2023, [Online; accessed 19-March-2025].

[57] Slack, “Kotlin Lang Slack,” https://kotlinlang.slack.com/, 2025, [Online; accessed 10-March-2025].

[58] J. Konstantin Tskhovrebov, “Awesome Kotlin Multiplatform,” https://github.com/terrakok/kmp-awesome, 2025, [Online; accessed 11-March-2025].

[59] Jetbrains, “Jetbrains Klibs.io,” https://klibs.io/, 2025, [Online; accessed 11-March-2025].

[60] M. erabi, “Local Preferences in Kotlin Multiplatform Using DataStore,” https://medium.com/@mohaberabi98/local-preferences-in-kotlin-multiplatform-using-datastore-c23ec677a35f, 2025, [Online; accessed 07-June-2025].

[61] J. Leitschuh, “ktlint-gradle,” https://github.com/JLLeitschuh/ktlint-gradle, 2025, [Online; accessed 23-February-2025].

[62] Astral, “ruff,” https://github.com/astral-sh/ruff, 2025, [Online; accessed 02-June-2025].

List of Figures

List of Tables

Listings

Part X

Appendix

Appendix A

Task Description

Following is the original task description as it was received.

OST Marketplace App für Android oder iOS

Bachelorarbeit Informatik FS25

1. Betreuer

Diese Bachelorarbeit wird von Martin Seelhofer [redacted] betreut. Gegenleser ist Mirko Stocker [redacted]

2. Studierende

Diese Arbeit wird als Bachelorarbeit an der Abteilung Informatik durchgeführt von:

- Tseten Emjee[redacted]
- Roger Marty[redacted]
- Simon Peier,[redacted]

3. Beschreibung

Im Rahmen dieser Bachelorarbeit soll eine X-Plattform App entwickelt werden, die den Studierenden und Mitarbeitenden der OST erlaubt, gebrauchte und nicht mehr benötigte Gegenstände wie Taschenrechner, Tablets, Laptops oder auch ganz alltägliche Dinge wie Kleider, Handys oder Velos auf einfache und unkomplizierte Weise innerhalb des abgeschlossenen OST-Universums auszutauschen und dazu auch Geldbeträge transferieren zu können.

4. Aufgabenstellung

Folgende Aktivitäten sollen im Rahmen der App Entwicklung mittels einer agilen Vorgehensweise durchgeführt werden:

- Anforderungsanalyse: Definition der Anforderungen und Spezifikationen der App.
- Technologie-Recherche: Die App soll mit Kotlin Multiplatform entwickelt werden. Dazu soll diese Technologie umfassend recherchiert und deren Reifegrad beurteilt werden.
- Cloud-Development: Entwicklung eines Cloud Backends mit Anbindung an eine Zahlungsschnittstelle (mind. Konzept).
- App-Development: Implementierung als X-Plattform-App für iOS und Android mit moderner UX.
- App-Release (optional): Veröffentlichung der App im Google Play und/oder Apple App Store.
- Test und Validierung: Erstellen eines Testkonzepts und End-zu-End-Validierung der App.
- Dokumentation: Ausführliche Dokumentation des Entwicklungsprozesses, der eingesetzten Technologien und der erzielten Ergebnisse.

Der genaue Funktionsumfang der App wird in Absprache mit dem Betreuer festgelegt, bzw. ergibt sich aus dem Projektfortschritt.

5. Zur Durchführung

In der Bachelorarbeit geht es darum, das in den verschiedenen Modulen der HSR/OST gelernte Wissen in einem Projekt anzuwenden. Insbesondere Ihre Software Engineering Fähigkeiten werden dabei gefordert sein. Es wird erwartet, dass Sie dieses Wissen anwenden und Methoden wie zum Beispiel Unit Testing, Clean Code und Continuous Integration, wenn immer möglich anwenden. Vorkenntnisse in den Bereichen KI, App-Entwicklung und UX-Design werden Ihnen dabei helfen, die Umsetzungsrisiken tief zu halten.

Während der Bearbeitung findet normalerweise einmal wöchentlich eine Besprechung mit dem Betreuer statt. Zusätzliche Besprechungen können nach Bedarf der Studierenden individuell veranlasst werden.

Alle Besprechungen (ausser Kick-off) sind von den Studierenden mit einer **Traktandenliste** (Was wurde gemacht? Was wurde erreicht? Was nicht? Welche Fragen haben wir?) vorzubereiten und zu leiten. Im Regelfall sollte eine Besprechung maximal eine Stunde dauern. Die Ergebnisse der Besprechungen (also die getroffenen Entscheidungen) sind durch die Studierenden zu protokollieren und an den Betreuer anschliessend zuzustellen oder an einem definierten Ort (z.B. Wiki) abzulegen.

6. Werkzeuge

Sofern nicht in der Aufgabenstellung vorgegeben sind die Studierenden für die Auswahl Ihrer Werkzeuge, Libraries, Frameworks, SaaS-Angebote, etc. selbst verantwortlich. Auf Wunsch kann eine an der OST gehostete virtuelle Maschine zur Verfügung gestellt werden.

7. Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen (siehe Sharepoint, hier finden Sie auch weitere Reglemente und Anleitungen). Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren. Es ist eine Zeiterfassung zu führen und im Bericht auszuwerten.

8. Termine

Eine Übersicht der Termine finden Sie auf Sharepoint.

17.02.2025	Beginn der Bachelorarbeit, Ausgabe der Aufgabenstellung durch den Betreuer.
März/April/Mai	Regelmässige Abstimmungsmeetings zum Projektfortschritt
09.06.2025	Die Studierenden erfassen den Broschüren-Abstract im Online Tool http://abstract.rj.ost.ch/ und geben den Broschüren-Abstract zur Kontrolle an ihren Betreuer frei.
11.06.2025	Der Betreuer gibt das Dokument mit dem korrekten und vollständigen Broschüren-Abstract zur Weiterverarbeitung an das Studiengangsekretariat frei.
13.06.2025	Abgabe des Berichts an den Betreuer und hochladen aller Dokumente auf https://avt.i.ost.ch/ bis 17 Uhr
13.06.2025	Ab 16 Uhr Präsentation und Ausstellung der Bachelorarbeiten.
Bis 30.08.2025	Mündliche BA-Prüfung
25.09.2025 16:00	Bachelorfeier

9. Beurteilung

Eine erfolgreiche Bachelorarbeit zählt 12 ECTS Punkte pro Studierendem. Für 1 ECTS Punkt soll mit einer Arbeitsleistung von ~30 Stunden gerechnet werden. Für die Beurteilung sind die Betreuungsperson und die zugeteilten Experten zuständig.

Gesichtspunkt	Gewicht
1. Organisation und Durchführung	10%
2. Formale Qualität des Berichts	10%
3. Analyse, Entwurf und Auswertung	20%
4. Technische Umsetzung	40%
5. Bachelorprüfung	20%

Die Note wird den Studierenden über unterricht.ost.ch bekannt gegeben. Die Betreuungsperson kann/darf vor diesem Zeitpunkt angeben, ob die Arbeit als bestanden/nicht bestanden gilt, er darf auch vorwarnen, wenn das Risiko auf eine ungenügende Note besteht. In diesen Fällen wird auch die Studiengangleitung informiert.

Im Übrigen gelten die Bestimmungen der Abteilung Informatik für Bachelorarbeiten gemäss «Leitfaden für Bachelor- und Bachelorarbeiten» in Version 1.2, gültig ab HS 23.

St. Gallen, den 18. Februar 2025

[redacted]

Martin Seelhofer

Dozent für Informatik
OST – Ostschweizer Fachhochschule

Appendix B

Usability Tests

This chapter shows the usability tests that were conducted for the beta and MVP release. The tests contain multiple scenarios a testee has to go through, with notes taken at each step. At the end there are general questions about the experience.

POSTE Usability Test - Beta	Testee 1	Testee 2	Testee 3	Improvements
Name	Chimey	Florian	Pascal	
Age	23	23	29	
Scenario 1				
You've just bought a new smartphone and now want to sell your old Samsung Galaxy S21. Do it!	kommt nicht aus keyboard raus	Worked like a charm, Bug when trying to exit keyboard (already fixed)	Create Listing Button fast nicht zugänglich. Mühe Tastatur zu verstecken.	Move Create Button with Keyboard (sticky, like chat, content in background)
Scenario 2				
You realise that you listed your old phone with a price that is too high. Try to correct it.	gut	Worked like a charm	siehe oben	
Scenario 3				
You need a replacement for your broken calculator to pass the upcoming analysis exam. How many used ones are there? Which one is the cheapest?	gut, filter gut, sort price geht nicht	2 used, Casio is cheapest	Sortierung funktioniert nicht.	Check with Meilisearch POST if Backend returns wrong
Scenario 4				
Before committing to buying the calculator you want to know if the seller is trustworthy. Try to figure it out and also check what other things he is selling.	dark mode text in listing nicht gut, zurück vom search mit back swipe probiert (iOS)	Has bad rating, lots of other calculators. Not able to view reviews	Reihenfolge?	ListingScreen DarkMode needs fixing; UserProfileScreen Listings new to old;
Scenario 5				
A friend just told you that he can lend you his calculator. Because of that you decide not to buy the calculator. Since you might still need to buy one later, make sure you can find the listing again quickly.		Used watchlist, no problem	Reihenfolge?	Use watchlist entry creation date for sorting
Scenario 6				
You're not looking for anything specific, but still want to browse for popular listings.	good, no / bad bottom padding	Used hot listings on overview	Weiter nach unten scrollen können bzw. Abstand benötigt.	Bigger padding for iPhones in Content
Scenario 7				
You decide to give your old Samsung to a friend instead of selling it. Remove it from the marketplace.	gut	Worked like a charm	i.O.	
Scenario 8				
Even though you love the POSTE marketplace, you don't like to receive push notifications. Disable them.	gut	No problemo	i.O.	
Follow-up Questions				
What went well?	alles, create easy, suche easy	Creating a new listing	Alle Aufgaben waren einfach zu lösen. Klarer Aufbau und man findet Dinge schnell.	
What was unclear for you?	text box leave mit swipe/back	Not really (only that there is no rubiks cube)	Tastatur verschwinden lassen.	
What did you like?	einfach und simple gehalten, direkt bei wichtigen funktionen	-	Einfache Bedinung, Schnelligkeit.	
Where would you have expected something different?	nichts	He expected a back navigation from other tabs back to the home tab	-	
On a scale of 1-10, how intuitive was the app?	9.5	8	9	
What is missing for you on the app?	ProfilScreen, user reviews, last sold / sell history, active since when, wenig infos zum trustworthiness checken	Nothing in particular (maybe bidding). And this small amount of categories is not very useful, there should be more	-	
What else would you like to tell us?	Create naming vielleicht ändern	-	Schwarzer Balken am oberen Rand entfernen. Beim Listing ein bisschen klarere Unterteilung (Offer uploaded Text und Description ist alles ein bisschen eng aufeinander, siehe Mockups). Mehr Padding bei Detail Listing (Message Seller).	Rename Create Tab to Sell, Padding for iPhone for ListingScreen so not inside Message Seller Button, Datum Farbe etc bei Detail Listing anpassen
Additional Questions / Comments				

Figure B.1: Usability Tests - Beta

POSTE Usability Test - MVP	Testee 1	Testee 2	Testee 3	Improvements
Name	Chimey	Florian	Pascal	
Age	23	23	29	
Scenario 1				
You've just bought a new smartphone and now want to sell your old Samsung Galaxy S21. Do it!	all ok	Search bar, add placeholder text, Creating a listing successful, Snackbar not shown when keyboard is open	i.O.	Add placeholder text to search
Scenario 2				
You realise that you listed your old phone with a price that is too high. Try to correct it.	all ok	Edit functionality successfully, Button hidden when keyboard is open	i.O.	Move button out of scaffold to make it reachable by scrolling
			Eigener Kommentar: Reihenfolge falsch, wieso "Save Changes" Button nicht gleiches Verhalten?	
Scenario 3				
You need a replacement for your broken calculator to pass the upcoming analysis exam. How many used ones are there? Which one is the cheapest?	all ok	When opening search over category, in the filter "categories" the "other" field is not selectable	i.O.	Make other field selectable
Scenario 4				
Before committing to buying the calculator you want to know if the seller is trustworthy. Try to figure it out and also check what other things he is selling.	all ok	User profile successfully found, rating was a bit unclear, one star was interpreted as a bad rating	Reihenfolge falsch herum	Fix order
				Indicate in the star rating, that the max number of stars is 5
Scenario 5				
A friend just told you that he can lend you his calculator. Because of that you decide not to buy the calculator. Since you might still need to buy one later, make sure you can find the listing again quickly.	all ok	add label to clickable icons, item was successfully added to watchlist, but it took some time		Add label to clickable icons
Scenario 6				
You're not looking for anything specific, but still want to browse for popular listings.	all ok	was found	i.O.	
Scenario 7				
You decide to give your old Samsung to a friend instead of selling it. Remove it from the marketplace.	all ok	was found, but took some time because icon does not have a label	i.O.	
Scenario 8				
Even though you love the POSTE marketplace, you don't like to receive push notifications. Disable them.	all ok	Notification successully disabled	i.O.	
Scenario 9				
You suddenly have the urge to buy a rubik's cube. Find a good offer and communicate your interest.	duplicated chat message	Start a conversation is not clickable, successfully sent chat afterwards	Nachicht zwei mal da wenn Chat nochmals geöffnet (passiert wenn man Chat das erste Mal öffnet), Chat Feld immer eine Zeile gross	Change text to "Start a conversation below"
				Fix that no message is shown twice
				Make text field expandable
Scenario 10				
You have an active listing for an old notebook with an interested buyer. Send him a QR-code bill.	all ok	QR Bill successfully generated	Öffnete zuerst Chat und probierte Buttons oben. QR Bill konnte nicht gesendet werden.	QR-Bill on iOS not working consistently
Scenario 11				
Another buyer messaged you regarding the same listing. His messages are rude and suspicious. Make sure that person can't contact you again.	all ok	User was blocked	i.O.	
Scenario 12				
You have just received payment for your notebook. Inform the buyer about it.	all ok	When user is blocked, chats of blocked user do not get deleted	i.O.	Information if one is alone in chat
Scenario 13				
The buyer marked the notebook as received which completes the transaction. Because the buyer was friendly and everything went well, leave him a good rating.	works for first user, but second bugs	First to confirm can rate, but the other user cannot, channel is not available anymore	Fehler wenn Negotiation fertig und man zurück auf Chats geht. Fertige Negotiation/Chat wird nicht angezeigt (trotz Backend Fix)	Don't filter for open listings, ignore if no thumbnail
Follow-up Questions				
What went well?	alles ok	Watchlist, QR Code	Fast alles	
What was unclear for you?	which chat am i selling and which am i buying	Search was difficult to find	QR-Bill aktivieren/senden (vielleicht Button in Chat sichtbar aber ausgeblendet oder Tooltip)	Make button visible if QR bill not enabled but show hint, Create toggle between buying/selling chats
What did you like?	design is nice	Deleting items is easy	Verhandlungen über Chat	
Where would you have expected something different?	-	No indication that a filter is selected	-	Add pill with current category in search
On a scale of 1-10, how intuitive was the app?	8	6	9	
What is missing for you on the app?	ProfilScreen, user reviews, last sold / sell history, active since when, wenig infos zum trustworthiness checken	Bin should be in navigation, review functionality is missing	-	
What else would you like to tell us?	Confirm step for clicking package received, Chat are loading long	-	Chat stabiler machen, Loading Indicator auch bei App Start	Also add loading indicator during PubNub initialize, confirm for package received
Additional Questions				

Figure B.2: Usability Tests - MVP

Appendix C

Designs

This chapter contains all mockups of the app that were created.

C.1 Low Fidelity

Following are the low fidelity mockups that were made to get a feeling of what the app should look like.

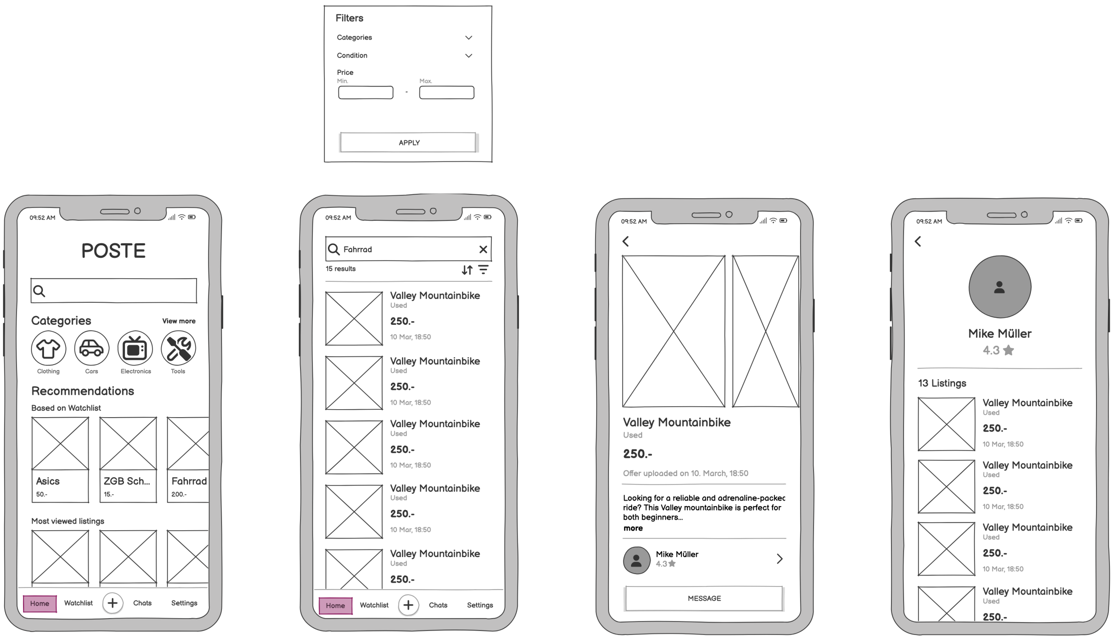


Figure C.1: Low-Fidelity Designs - Search, Listings, Profile

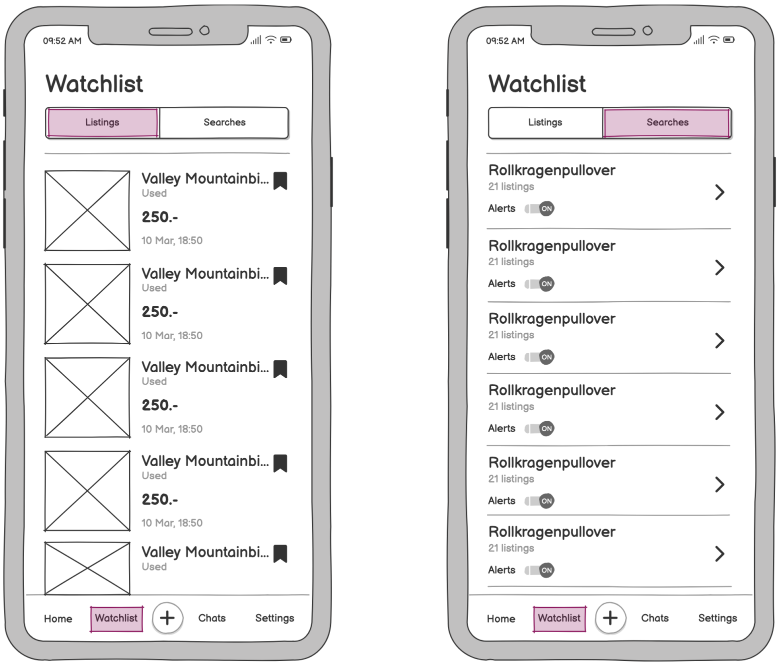


Figure C.2: Low-Fidelity Designs - Watchlist

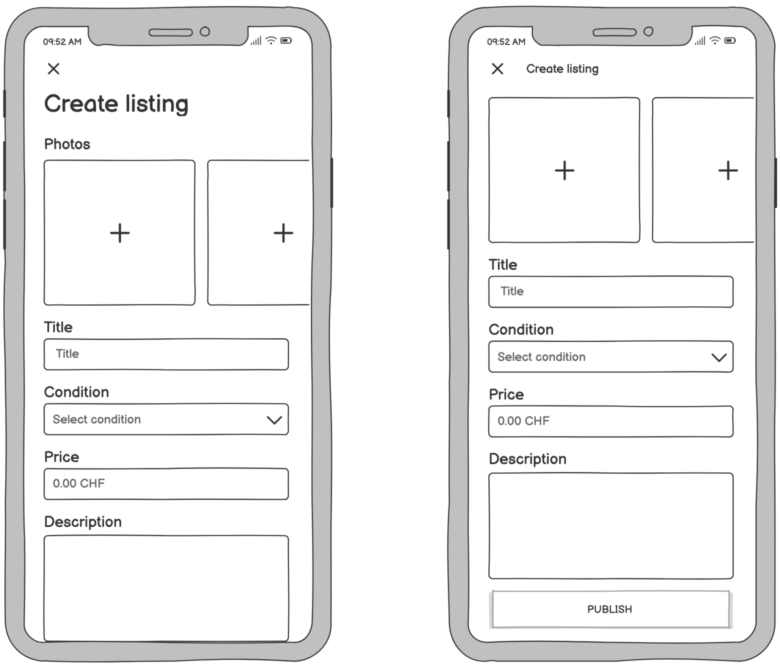


Figure C.3: Low-Fidelity Designs - Sell

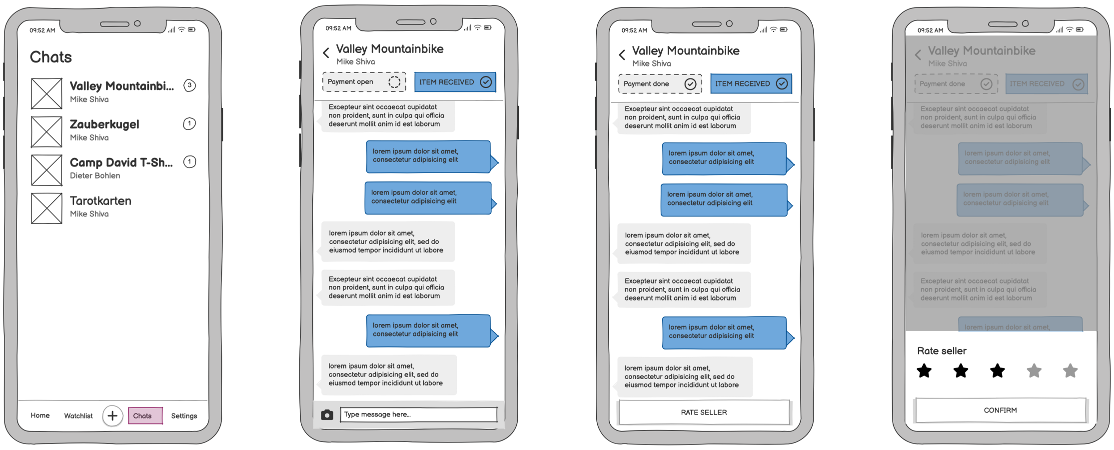


Figure C.4: Low-Fidelity Designs - Chats

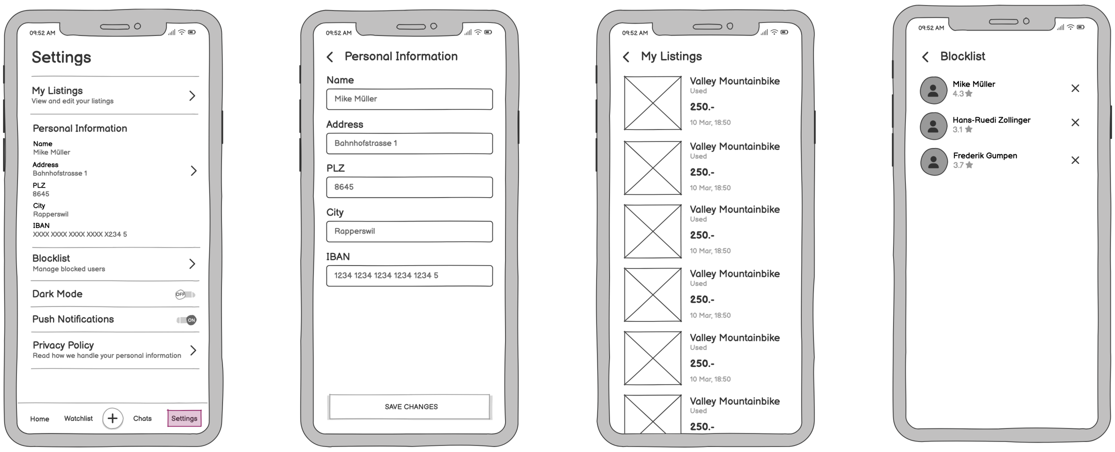


Figure C.5: Low-Fidelity Designs - Settings

C.2 High Fidelity

After the low fidelity designs, more accurate ones were made using Figma, which resemble how the final product should look like.

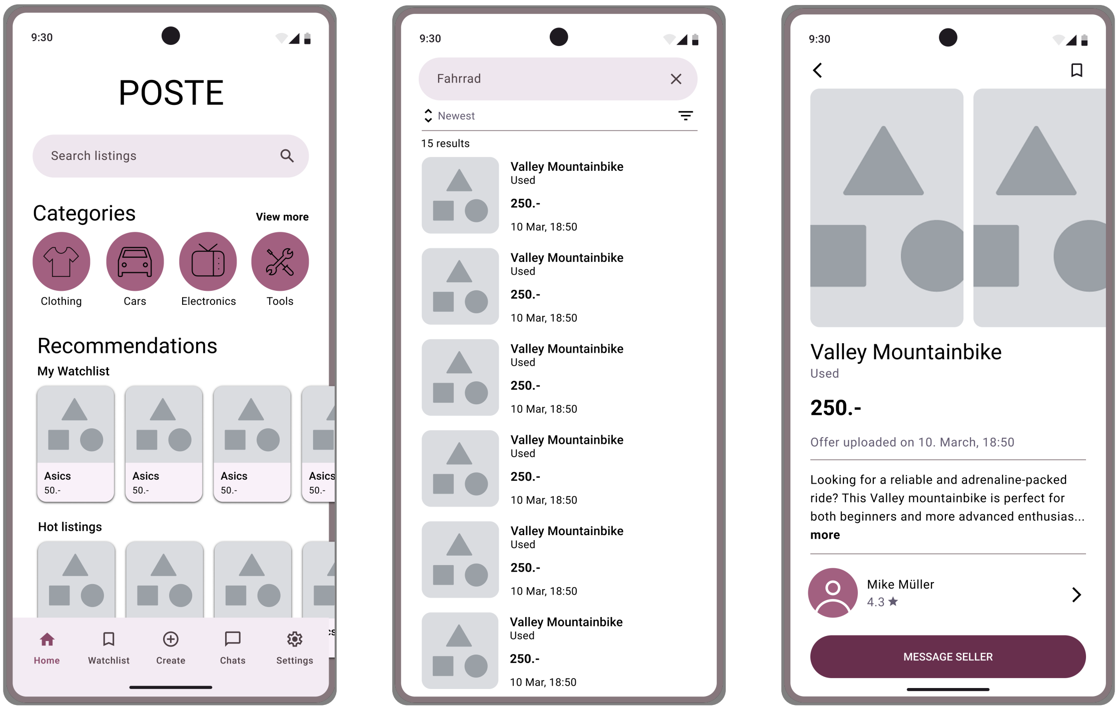


Figure C.6: High-Fidelity Designs 1

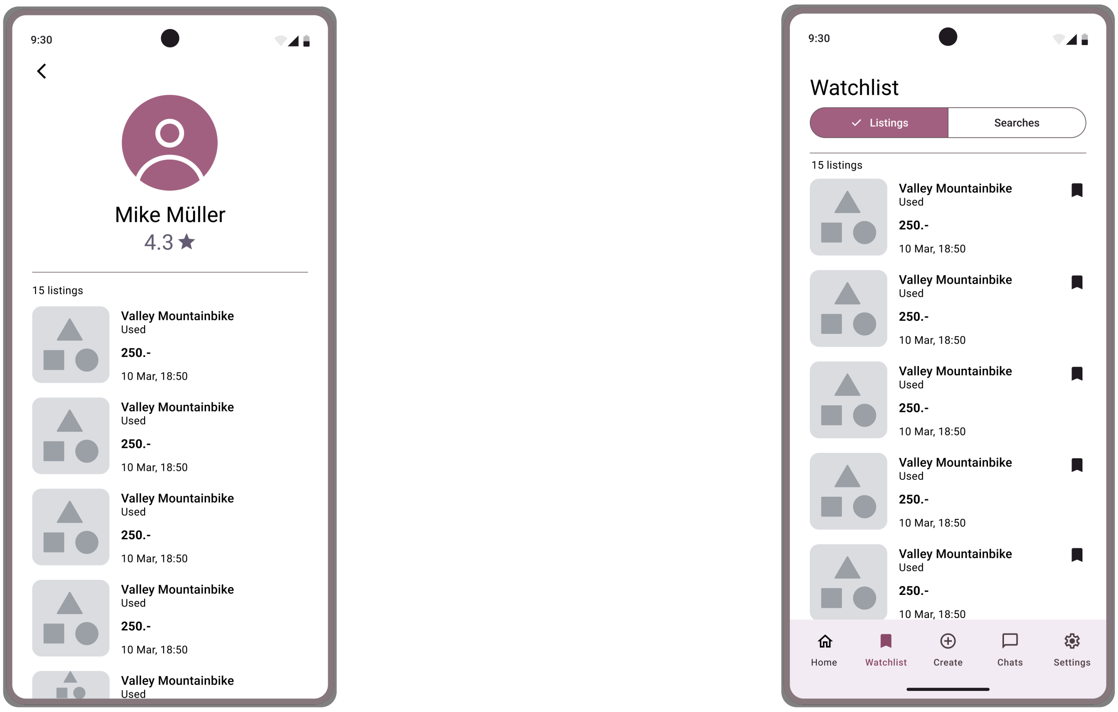


Figure C.7: High-Fidelity Designs 2

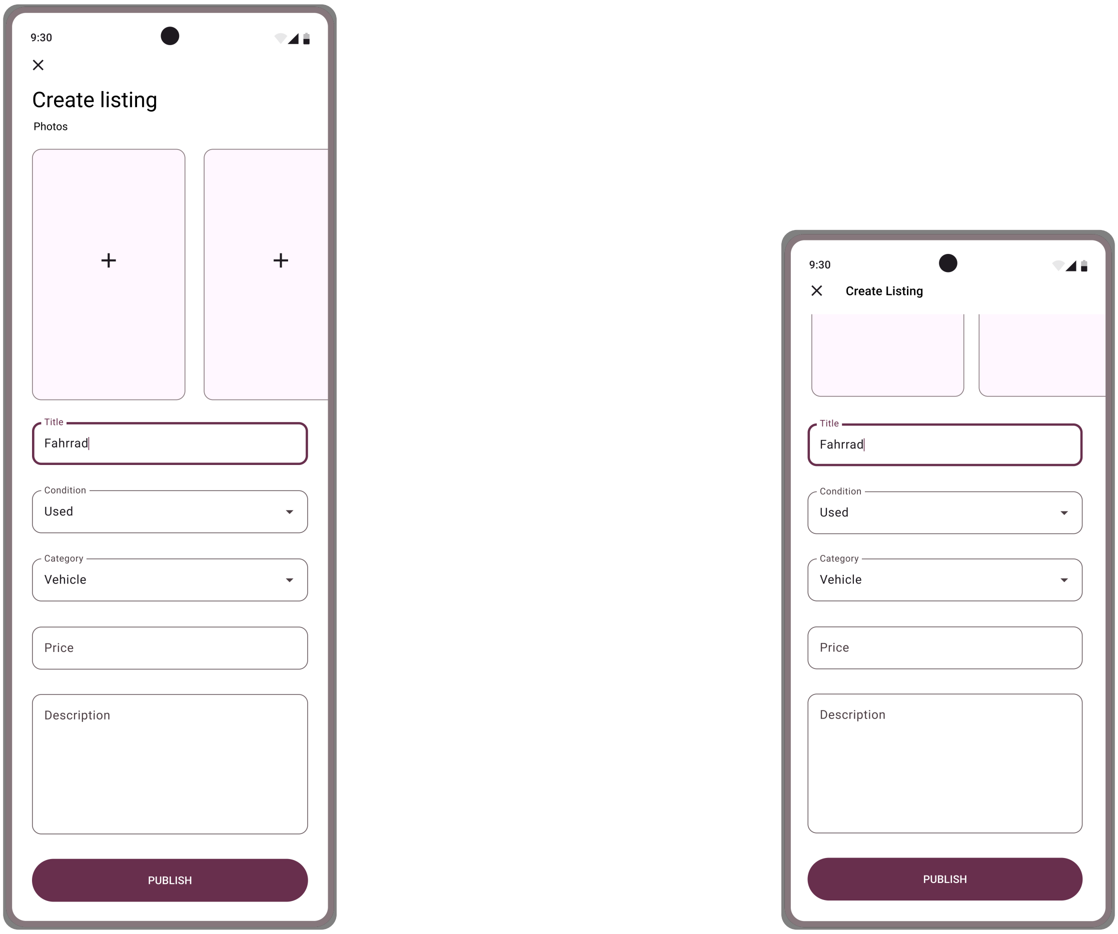


Figure C.8: High-Fidelity Designs 3

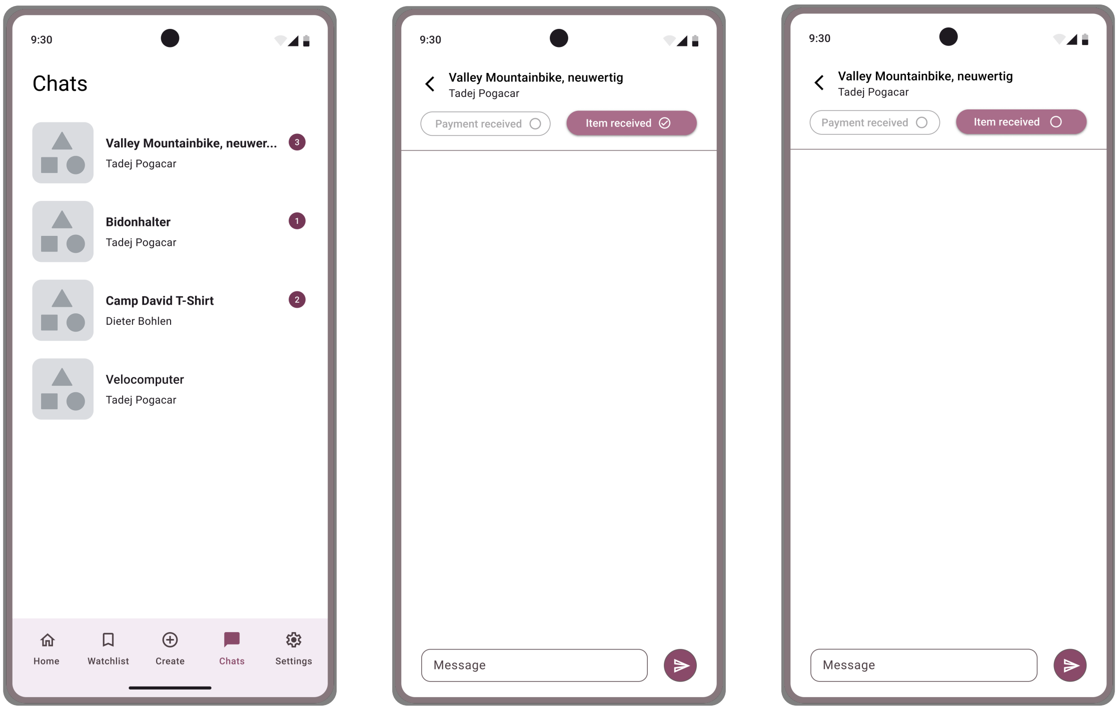


Figure C.9: High-Fidelity Designs 4

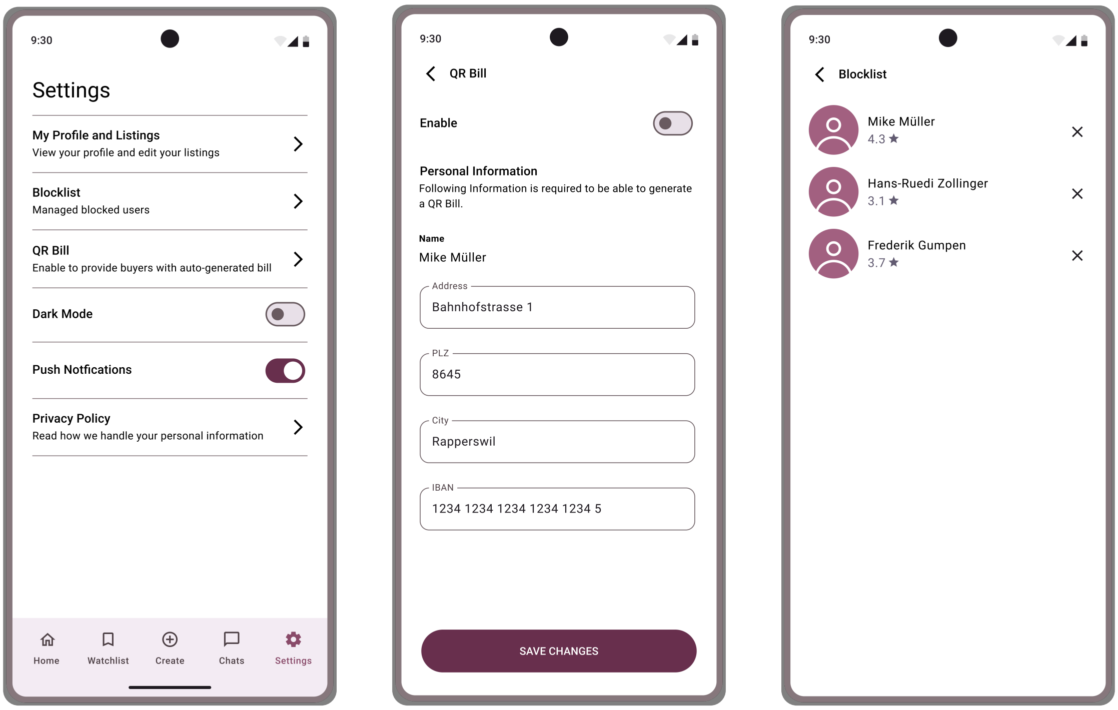


Figure C.10: High-Fidelity Designs 5

Appendix D

Meeting Minutes

This chapter contains the summarized form of all notes taken during the weekly meetings together with the advisor, Martin Seelhofer. The full version is located on the Miro board used for this project.

19.02.2025

Kick-Off Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier, Tseten Emjee
Location: RJ — 8.125

Agenda:

- First meeting regarding bachelor thesis
- Introduction of all project members
- Discussion of task definition and initial questions
- Feedback and possible improvements based on the semester thesis

Decisions:

- Justify decisions, especially those regarding technologies, more scientifically
- Better usage of footnotes

24.02.2025

1. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Share progress of inception phase
- Discuss project planning

Decisions:

- Do not act as middle-man regarding payments but instead use QR code bill solution
- Separate chapter for KMP research in documentation as well as a separate document
- Check if Switch edu-ID or OST Microsoft login is feasible

03.03.2025

2. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Share progress of inception phase
- Discuss questions and make decisions

Decisions:

- No market analysis needed for this thesis
- Integration of a native library is an interesting aspect of KMP

10.03.2025

3. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier
Location: MS Teams Meeting

Agenda:

- Discuss questions that were being clarified
- Show progress of documentation with focus on research chapter and FRs/NFRs

Decisions:

- Specify certain NFRs more precisely
- Interim presentation to be held on the 14th April
- Continue with AWS, even though OST would cover some Azure costs
- Decision regarding Firebase Authentication or OST Microsoft integration remains open, Switch edu-ID is off the table

17.03.2025

4. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Progress update after completion of first elaboration sprint
- Discuss questions and make decisions

Decisions:

- OST Microsoft account integration is realisable and ticket is opened
- Own payment process is fine, though concept/research about payment providers is required in separate section which covers changes required for the integration of such a provider and major differences between popular payment providers

24.03.2025

5. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Present and talk about low/high fidelity designs
- Discuss questions and make decisions

Decisions:

- Instrumented tests can be done manually or locally before release due to CI limiations
- Check if PubNub supports rich text features

31.03.2025

6. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Present prototype
- Discuss emerged problems with KMP, especially CocoaPods
- Discuss questions and make decisions

Decisions:

- OST logo can be used freely because it is an internal project
- Meeting held regurarly on 14th April, interim presentation postponed due to scheduling conflicts
- Date for bachelor exam provisionally set on July 2nd or 3rd

14.04.2025

7. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Show progress of first construction week
- Discuss requirements and expectations for interim presentation

Decisions:

- Next weekly meeting will be skipped due to public holiday
- Date for bachelor exam will be discussed at interim presentation

22.04.2025

Interim Presentation

Attendees: Martin Seelhofer, Mirko Stocker, Paul Sevinç, Roger Marty, Simon Peier, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Conduct interim presentation
- Questions and feedback

Decisions:

- Bachelor exam will be held on June 25th
- Presentation feedback
  - Justify why KMP was chosen in the end and what alternatives exist
  - Explain why backend is written in a different language and why Python
  - Mention SSO earlier in presentation
  - Statistic about actual use of common code vs. platform-specific code

28.04.2025

8. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Discuss interim presentation feedback
- Revise the E2E testing criteria
- Show progress

Decisions:

- Create some instrumented tests of user journeys using production backend
- Implement on iOS and Android but not in pipeline
- Add testing insights to KMP research

05.05.2025

9. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Show progress of second construction sprint
- Discuss authentication flow

Decisions:

- -

12.05.2025

10. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Show construction progress
- Discuss instrumented test problem
- Ask questions regarding submission and poster

Decisions:

- Find source or open bugfix regarding instrumented test issues
- Open ticket to request test user for OST MS Entra ID

19.05.2025

11. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Show progress of beta
- Showcase the features
- Code review/overview

Decisions:

- Mention backend technology decision in documentation and that it wasn't the focus and not that important

26.05.2025

12. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Present and demonstrate beta release
- Show general progress and what is left to do

Decisions:

- OST MS Entra ID cannot be used for Apple review process because all accounts require 2FA
- Instead implement a hidden button where password is required, if correct it bypasses backend Authentication
- Check if and how good the app works on tablets and if it should be supported or not

02.06.2025

13. Weekly Meeting

Attendees: Martin Seelhofer, Roger Marty, Simon Peier, Tseten Emjee
Location: MS Teams Meeting

Agenda:

- Show construction progress towards MVP
- Demonstrate chat feature
- Discuss questions

Decisions:

- Mention Stripe Connect in payment integration concept
- More detailed demonstration after MVP release