FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	A trip planning application for Android
Student:	Martin Hudymáč
Supervisor:	Ing. Tadeáš Sosín
Study program:	Informatics
Branch / specialization:	Software Engineering 2021
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

Planning trips can be a tedious and time-consuming process, requiring users to meticulously research destinations, organize transportation, and coordinate schedules. This thesis aims to design and prototype an application that will simplify and automate this process.

Do the following steps:
1) Analyse the market competition and user needs.
2) Based on the results, analyze and define the application requirements.
3) Design and implement the application prototype for Android.
4) Test the app and implement analytics.
5) Propose possible future improvements.

Electronically approved by Ing. Michal Valenta, Ph.D. on 21 December 2023 in Prague.

Bachelor’s thesis

TRIP PLANNING APPLICATION FOR ANDROID

Martin Hudymáč

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Tadeáš Sosín
May 16, 2024

Contents

List of Figures

List of Tables

List of Code Listings

I would like to express my sincere gratitude to my supervisor, Ing. Tadeáš Sosín, for his professional guidance, friendly and patient approach, and valuable advice while introducing me to multiplatform development. I am also grateful to my family for their constant support, not only in my studies. Last but not least, I want to thank my girlfriend for always keeping me motivated and focused on my goals.

Declaration

In Prague on May 16, 2024

Abstract

With the increasing popularity of travel due to more affordable transportation options, efficiency has become a priority. This bachelor's thesis, therefore, focuses on analyzing, designing, implementing, and testing a mobile application for trip planning with automatic itinerary optimization. Existing solutions and potential users were analyzed to create a set of requirements. These were then translated into a design the application could be built on. This thesis output is an Android application that simplifies the process of creating, managing, optimizing, and keeping a record of trips. It is designed using Kotlin Multiplatform Mobile based on Clean Architecture for future scalability and expandability to more platforms. Google Maps Platform APIs are used to get accurate and up-to-date data. This application contributes to society by empowering people to travel more and worry less.

Keywords trip, itinerary, optimization, mobile, Android, KMM, Clean Architecture, Jetpack Compose

Abstrakt

S narůstající popularitou cestování se díky dostupnějším možnostem dopravy dostává do popředí především efektivita. Tato bakalářská práce se proto zaměřuje na analýzu, návrh, implementaci a testování mobilní aplikace pro plánování cest s automatickou optimalizací itineráře. Za účelem vytvoření souboru požadavků se provedla analýza již existujících řešení a potenciálních uživatelů. Ty pak byly převedeny do návrhu, dle kterého by bylo možné aplikaci vytvořit. Výsledkem této bakalářské práce je aplikace pro systém Android, která slouží ke zjednodušení procesu vytváření, spravování, optimalizace a evidence výletů. Aplikace je navržena s využitím jazyka Kotlin Multiplatform Mobile založeného na Clean Architecture pro možnost budoucího škálování a rozšíření na více platforem. Pro získávání přesných a aktuálních údajů jsou využívány API platformy Google Maps. Aplikace představuje významný přínos pro společnost neboť umožňuje lidem více cestovat a méně se o vše starat.

Klíčová slova výlet, itinerář, optimalizace, mobil, Android, KMM, Clean Architecture, Jetpack Compose

List of Abbreviations

API	Application Programming Interface
F	Functional Requirement
IDE	Integrated Development Environment
JVM	Java Virtual Machine
KMM	Kotlin Multiplatform Mobile
N	Non-functional Requirement
SDK	Software Development Kit
SQL	Structured Query Language
UC	Use Case
UI	User Interface
UML	Unified Modeling Language
UX	User Experience

Introduction

The world has shrunk dramatically in recent times. With rapid technological development and affordable transportation options, we are connected even more than was imaginable only a few decades ago. Information about distant destinations is readily available from any of our smart devices. Airlines, budget carriers, and ride-sharing services have made travel more accessible. Posts with picturesque views and different cultures from social media motivate us to experience the world ourselves.

However, despite this ease of access, planning such a trip can still be a big undertaking. People must go through many suggested itineraries, starting in different parts of the city. These itineraries usually include places the general public likes, which may only suit some. Hence, people search for places they want to see and somehow navigate themselves. This whole process can be meticulous and challenging.

My mother visited Prague not long ago, and since I have been studying here for a while now, I am familiar with this city. Therefore, I wanted to make an itinerary that would be easy to follow and lead her to all the places I think are must-see. I tried to make a long route on Google Maps but was unsuccessful because Google allows only a few locations in one route. I tried searching for an alternative but could not find one. In the end, I wrote down all the places with their corresponding map links next to them in an order I thought was efficient. This experience made me question how to handle this process better. Thus, I decided to make this my bachelor's thesis objective.

The main goal is to create a mobile application that makes this process less overwhelming so people can look forward to the trip instead of stressing out. The app will allow simple trip creation by adding places to a list. The locations can then be ordered automatically or at the user's will. After creation, the trips are saved and wait for their date. Users will see the itinerary on the home screen on the day of the journey, and the app will track them as they go according to the itinerary. Upon arrival at a place from the itinerary, the app will prompt users to take photos. These pictures will be saved so the user can remember the trip later by looking in the app's gallery.

Goals

This thesis aims to design and prototype an Android application for easy trip planning and management with intelligent itinerary creation.

The first goal is to define a potential user and analyze other solutions already on the market. Identify the competition's strengths and weaknesses, and, from potential users, find what the app's main functionalities should be.

Based on this analysis, the second goal is to determine the specific functional and non-functional requirements.

Once the requirements are established, the next step is to design a low-fidelity prototype, which multiple people with different backgrounds will test. Based on their insights, the design should be iterated upon. Next, suitable third-party tools will be chosen to execute the solution and its internal architecture. With the right tools, the Android prototype is to be implemented.

The final goals are implementing Google Analytics to test the app's UX and UI and, based on the tests, proposing future improvements.

Chapter 1
Analysis

This chapter creates the base for the entire project. It is a necessary step toward understanding the intended purpose of this application. This phase focuses on discovering the potential user base, analyzing existing solutions, and defining the functional and non-functional requirements based on their strengths and weaknesses.

1.1 Potential User

The analysis begins by identifying the potential users for the future application. Analyzing relevant statistics can provide insights into the size and demographics of the target audience. This helps to realize the application's perspective early on, justifying its existence.

According to statistics from Our World in Data, international travel is rising thanks to affordable aviation [1]. Before the COVID-19 pandemic, the number of international visits had more than doubled since the year 2000, with Europe experiencing the highest increase of all. Moreover, domestic travel was also on the rise, with the average European taking around two trips per year [1].

Navigating unfamiliar cities on foot can be daunting for many travelers, especially first-time visitors or those with limited time for planning. Without proper assistance, these individuals may struggle to create efficient itineraries, potentially leading to missed opportunities, wasted time due to inefficient route planning, or overlooked hidden gems that could enhance their travel experience.

The above statistics underscore the vast and diverse potential user base for a solution that addresses these challenges. A common thread among this diverse group is smartphone ownership. Data reveals that over 4.8 billion individuals worldwide possess a smartphone, with this number continuing to rise [2]. These statistics position the smartphone platform as an ideal candidate to aid people with their travels.

1.2 Existing Solutions

Since this problem concerns a significant part of society, people have come up with many solutions. In particular, solutions for Android will be taken into consideration. After searching the prompt 'Trip planner' in Google Play, a long list of possible applications was displayed, ordered by relevancy. A few of the first ones will be analyzed based on my preferences and published information. All the apps chosen must be actively worked on with an update in the last half a year as of 1/4/2024.

Each analysis consists of the following five parts: general information, first power on and design, itinerary creation, itinerary progression, and viewing past trips.

1.2.1 Wanderlog

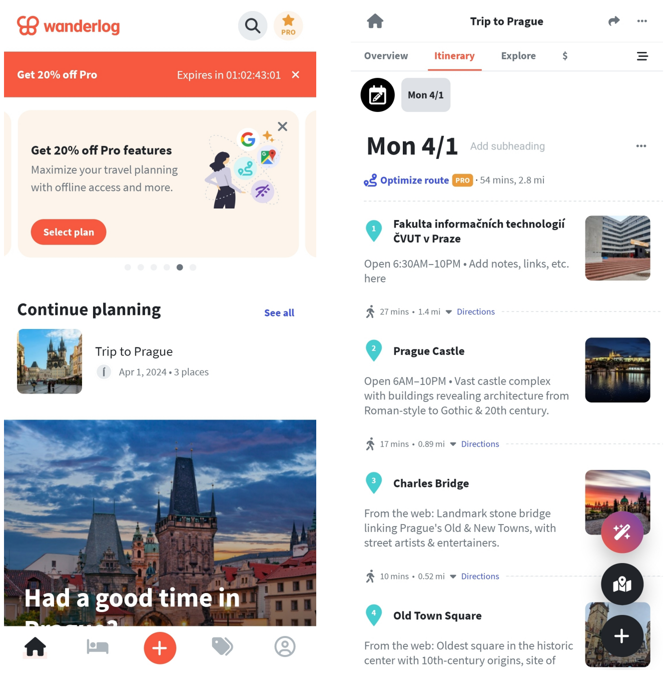
Wanderlog is the first recommendation that Google Play offers. It is an app from an American company based in California, United States, intending to be the only platform needed to manage everything related to travel. The app has over a million downloads and an average rating of 4.7 stars out of 5. It offers a free limited and paid premium version, which costs around 90 euros annually.[3]

Upon opening the app, the user is prompted to create an account or log in. Without an account, the app cannot be accessed. After creating an account, a user-friendly interface is shown 1.1. It consists mainly of orange, red, and yellow colors, and dark mode is not implemented. The app is straightforward to navigate thanks to an easy-to-use bottom bar and clearly labeled tabs. The only problem with the design is the overwhelming amount of ads for their pro version, which makes the UI[1] chaotic.

Creating an itinerary is quite simple. Search works reliably, and the places can be deleted or moved quickly. There are three buttons on the right. The first represents an AI[2] assistant, accessible after paying for the pro version. The second shows a map of all the locations in the itinerary and the user's location. The last one offers additional features, including sharing the trip with a different person, adding plane or train tickets, hotels, notes, and more.

When the trip is active, nothing in the UI changes. However, it is helpful that upon clicking the place, it opens in Google Maps or Waze to find directions.

After finishing the itinerary, nothing changes yet again. The trip looks as if nothing ever happened.



Get 20% off Pro features
Maximize your travel planning with offline access and more.

Trip to Prague
Apr 1, 2024 • 3 places

Mon 4/1
· 54 mins, 2.8 mi

1 Fakulta informačních technologií ČVUT v Praze
Open 6:30AM–10PM • Add notes, links, etc. here

27 mins • 1.4 mi

2 Prague Castle
Open 6AM–10PM • Vast castle complex with buildings revealing architecture from Roman-style to Gothic & 20th century.

17 mins • 0.89 mi

3 Charles Bridge
From the web: Landmark stone bridge linking Prague's Old & New Towns, with street artists & entertainers.

10 mins • 0.52 mi

4 Old Town Square
From the web: Oldest square in the historic center with 10th-century origins, site of

(a) Home screen. **(b)** Itinerary screen.

Figure 1.1 Wanderlog screenshots.

[1] User Interface
[2] Artificial Intelligence

1.2.2 TripIt

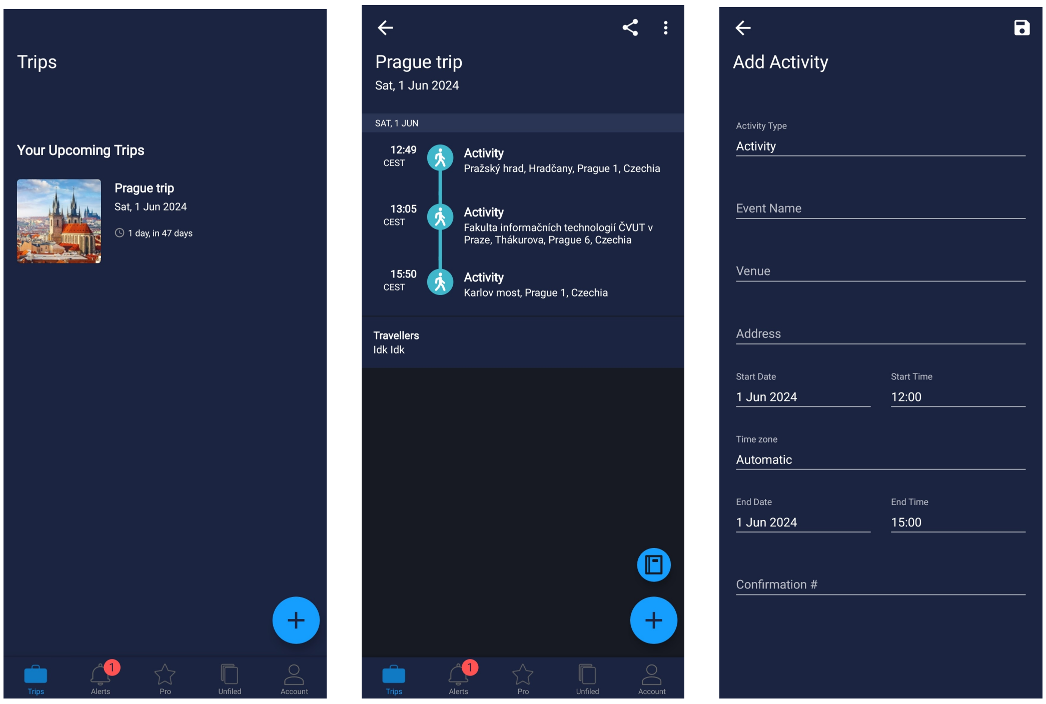
TripIt is another example of an app from the hands of American developers. Their slogan is that they organize all your travel plans in one place. On Google Play, it has a rating of 4.6 stars and over 5 million downloads. It provides a free limited and a paid version, which costs approximately 50 euros a year.[4]

When opening TripIt, the user has to create an account. Otherwise, the app cannot be accessed. The primary brand color is blue, which is prominent throughout the application and implements both light and dark mode 1.2. It is easy to navigate thanks to a bottom bar with five buttons and a floating button to create a new trip.

This app specializes more in the formal side of traveling, offering features like accessing risks, getting help abroad, having travel documents, or insurance. Therefore, the process of creating an itinerary is quite tricky. When adding a new place to the itinerary, the user must always choose from one of the categories. This process is made even more difficult with a cluttered adding screen. The places cannot be easily moved or removed after being added, and the application offers no assistance with ordering.

On the day of the travel, users can get directions to the place via Google Maps, but the link is difficult to find. Moreover, documents can be added to each place, and a safety score is shown. There is no indication of where the user is currently on the itinerary.

After completing the trip, it is saved to the past trip category. From there, it can be edited, shared, or deleted. Nevertheless, there is no way of using this information as a memory of this travel.


Trips

Your Upcoming Trips

Prague trip
Sat, 1 Jun 2024
1 day, in 47 days

Prague trip
Sat, 1 Jun 2024

SAT, 1 JUN

12:49 CEST — Activity — Pražský hrad, Hradčany, Prague 1, Czechia

13:05 CEST — Activity — Fakulta informačních technologií ČVUT v Praze, Thákurova, Prague 6, Czechia

15:50 CEST — Activity — Karlov most, Prague 1, Czechia

Travellers
Idk Idk

Add Activity

Activity Type: Activity
Event Name
Venue
Address
Start Date: 1 Jun 2024
Start Time: 12:00
Time zone: Automatic
End Date: 1 Jun 2024
End Time: 15:00
Confirmation #

(a) Home screen. **(b)** Itinerary screen. **(c)** Add activity screen.

Figure 1.2 TripIt screenshots.

1.2.3 Tripadvisor

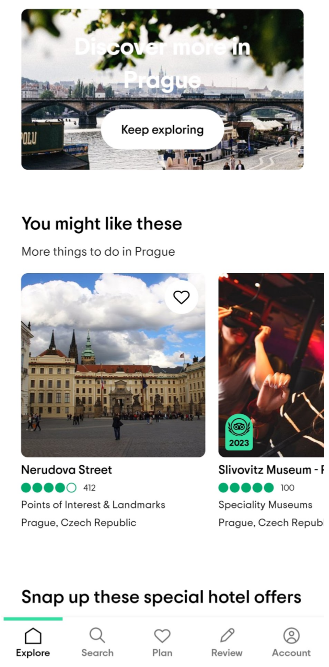
The next application from the list is Tripadvisor, an American company. They aim to empower users to plan their next trip, read reviews, get travel advice, and all of this is free. Released in 2010, it is by far the most popular app in this category, with over a hundred million downloads and a rating of 4.4 out of 5 stars. However, their services mainly concentrate on reviewing popular travel destinations and connecting a community of travelers rather than creating an itinerary. [5]

After opening the application, users are welcomed with a login screen. However, in this case, it is skippable. Behind the login screen was a well-made UI 1.3 with green as the primary color. Offering both light and dark modes, the app is pleasant to look at during the day and at night. From first sight, users can get much inspiration with a minimum of ads that fit well with the context. Navigation is easy to use, with a bottom bar comprising five destinations and clearly labeled buttons and tabs.

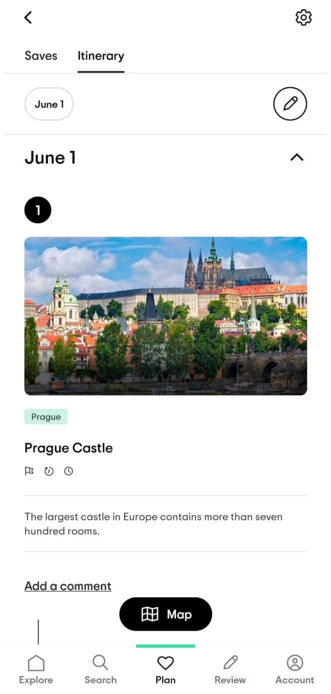
To create an itinerary, users are required to create a free account. On the store pages, an AI-powered itinerary creation is promoted. However, this feature was still in closed beta at the time of testing. Hence, it could not be tested. When adding a location, users can only choose from tourist attractions or restaurants, which excludes places like universities or libraries. After a location is added, it is relatively easy to change the order or remove a location, but adding a new one requires multiple steps and could have been made more accessible. There is no option to optimize the itinerary order. Nevertheless, a welcomed feature is the ability to invite other people to the trip.

This application does not offer any itinerary tracking or navigation on the day of the travel.

No distinction is made between upcoming, current, or past trips.



(a) Home screen.



(b) Itinerary screen.

Figure 1.3 Tripadvisor screenshots.

1.2.4 iplan.ai

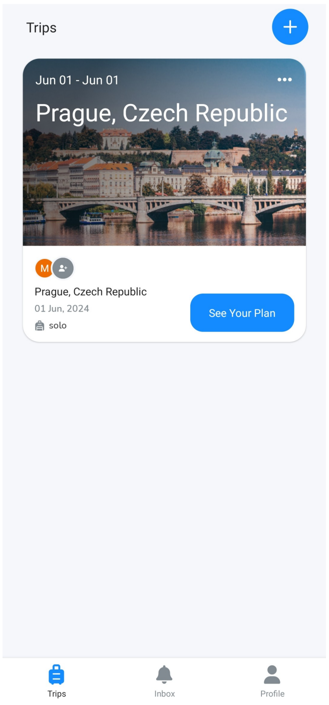
Continuing with iplan.ai, this Canadian application focuses primarily on AI-powered itinerary creation. On Google Play, it has 4 out of 5 stars with more than a hundred thousand downloads. Most of the app is free, with a few features hidden at an 11 euros a year price. These features include offline mode, dark mode, and trip sharing. [6]

When opened for the first time, the app requires users to create a free account. Otherwise, its content is inaccessible. The app has a simple but good-looking UI 1.4. The primary color is blue and is found everywhere. However, it is a shame that dark mode is behind a paywall. Navigation is straightforward, with the bottom bar having three icons and all the buttons adequately labeled.

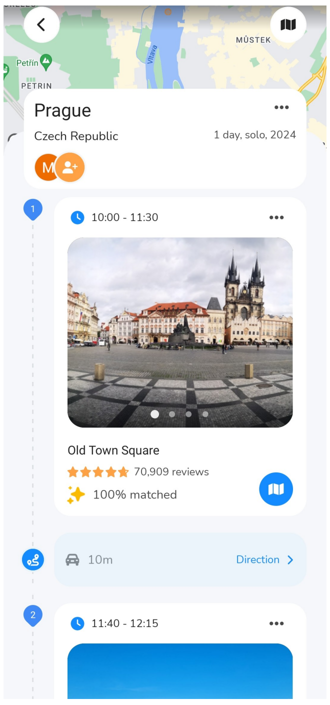
Itineraries can be made in two ways: by the AI or manually by the user. When the trip is made using AI, it is still editable later. Adding activities is simple, but there is no way of adding a place that is not a tourist attraction or a restaurant. Existing places in the itinerary can be easily removed, but changing the order is problematic. Moreover, no optimization is available. The only automatized part is the initial creation, after which everything is up to the user.

When following the itinerary, the app offers a map visualization for an effortless overview. Getting directions is also easy thanks to direction buttons between all the places, except for the first one. However, there is no indication of where the user is currently on the itinerary.

After completing an itinerary, it stays unchanged in the trips section of the application. No further actions are available.



(a) Home screen.



(b) Itinerary screen.

Figure 1.4 Iplan.ai screenshots.

1.2.5 Polarsteps

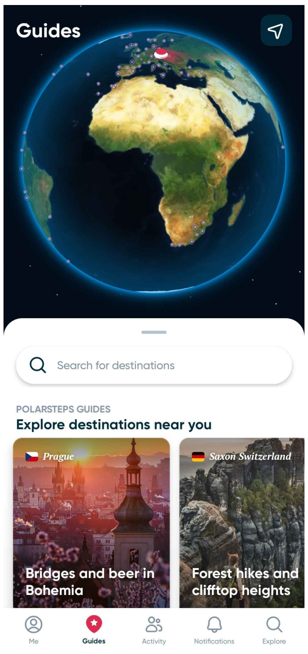
The last application in this analysis is Polarsteps. It is a Netherlands-based company with the aim of being a platform for trip planning, tracking, and later viewing. Since its release in 2016, it has gained over a million downloads with an average rating of 4.8 stars out of 5. The entire application is available for free with no paid tiers. [7]

When launched, the app requires users to create an account. The design of this app is styled in red colors but is made interesting by having a globe always present at the top of the screen. The globe mainly serves to visualize trips and show interesting locations to visit. The bottom bar is simple, offering five different destinations. However, the rest of the application is difficult to read or make sense of. Information is ordered so that it is challenging to understand, and context menus come unpredictably from the right, bottom, or as a dialog.

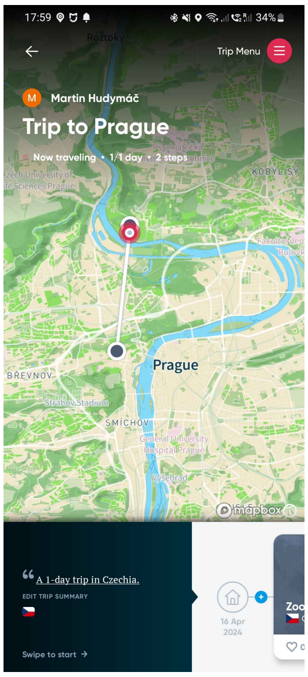
While creating a trip, users have three options: past, present, or future. However, it is hard to understand where and how to add the desired location when creating a trip. The search does not work reliably or show results for popular destinations. Moving and removing locations is also not straightforward. There is no AI-powered helper or way to optimize the route. The only simple way of creating an itinerary is to record a trip, and the app gives a relatively good approximation.

Trying to follow an itinerary also requires some effort. There is no way of getting directions straight from the app. The only found way was to copy the address from the location details. However, the tracking works well, and users can easily see their progress within an itinerary.

Although the negative description so far, the app shines when viewing past trips as a memory. Users can view their journeys on a three-dimensional map and, what is more, can add photos to the individual locations of the trip. This makes reliving past trips engaging and much more enjoyable.



(a) Guides screen.



(b) Itinerary screen.

Figure 1.5 Polarsteps screenshots.

1.2.6 Summary

Five of the most popular applications available on the Android platform were analyzed. Features and design varied vastly between the apps. However, none of them provides a perfect solution for all itinerary-related needs. These are the key findings for each application:

- **Wanderlog**: A clean UI, but the free version suffers from many ads. Itinerary creation is simple and user-friendly. Users do not get any indication during active tracking, although getting directions is simple. Lastly, there is no option to keep trips as a memory.
- **TripIt**: Difficult to navigate. Creation and modification of itineraries are challenging. While traveling, there is no progress tracking, and directions are hidden in the menus. Past trips are saved and can be viewed, but do not offer anything additional.
- **Tripadvisor**: The most significant advantage is free, complete access. UI is simple yet practical. On the other hand, itinerary creation is complicated and limited. User tracking or past trip viewing is missing entirely.
- **iplan.ai**: This application highly leverages AI-powered[3] itinerary creation. The UI is user-friendly and easy to navigate. Trip modification is also low-effort. On the day of the trip, directions are accessible, although there is no user tracking. After completing trips, they stay in the application but are treated as never completed.
- **Polarstep**: This app has exciting visuals but lacks ease of use. Everything is difficult to do. One saving feature is past trip viewing. Users can view their progress with the option of adding pictures, which serves as a way to keep the trip in memory.

1.3 Application Requirements Analysis

In the following section, concrete requirements will be defined. These requirements will be created based on the strengths and weaknesses of the previously tested applications; what is more, the user reviews from these applications will also be considered as they mirror what the general public demands. These will aid in realizing the objective of this thesis.

1.3.1 Functional Requirements

Functional requirements are product features or functions that developers must implement to enable users to accomplish their tasks.

F1: Create a trip

Users can create new trips by specifying their name, date, and itinerary.

F2: Search for places

Users can search online for places of interest relevant to their trip. Any type of location is available to add to the itinerary.

[3] AI is not the focus of this thesis; hence, it will not be implemented.

F3: Change a trip

Users can edit existing trips by modifying the itinerary and changing the date or the name. This includes changing the itinerary order, adding new locations, or removing the existing ones.

F4: View itinerary

Users can view an itinerary sorted in their desired order with the names and addresses clearly visible.

F5: View distances between places

The itinerary shows an estimated distance in minutes between all the locations.

F6: Trip order optimization

Users can optimize their itinerary based on the total distance walked.

F7: Plan a trip for a specific day

Users can schedule a trip for a future date. The itinerary will be displayed on the home screen on that particular day.

F8: Multiple trips

The application allows users to create and store multiple trips.

F9: View upcoming and completed trips

Users can view a list of all future and past trips.

F10: Complete a trip

Users can mark a trip as completed after finishing it. This will move the trip to the completed section.

F11: Duplicate a trip

Users can create a copy of an existing trip for a new trip creation.

F12: Get navigation to a place in the itinerary

Users can initiate navigation to any place in their trip by clicking it. They are redirected to the application of their choosing.

F13: Get details for places

Users can get a detailed view of a place in a separate map application.

F14: Use the camera to photograph a place

Users can capture photos of the place they are currently in, and the photo will be saved with the trip.

F15: Add or remove an image from the gallery

Users can add existing photos from their device's gallery to a specific location within a completed trip or remove them.

F16: Start from current location

Users can add their location as the first place in the itinerary.

1.3.2 Non-functional Requirements

These are the quality constraints that the system must satisfy according to the project specification.

N1: Usability

The application should be easy to learn, navigate, and use with an intuitive user interface.

N2: Reliability

The application should display accurate information and be stable without crashes.

N3: Maintainability

The code should be documented and modular for easy maintenance and future updates.

N4: Localization

The application should be adaptable for international use, supporting multiple languages and regional formats.

N5: Offline functionality

The application allows users to view trips, itineraries, and user-saved photos, change the order, or remove places from the itinerary while not connected to the internet.

1.4 Use Cases

In the following part, use cases will be defined. These outline the interactions between actors and the system to achieve a specific outcome. In this case, every use case has a single actor, the application user, so only flow will be described for each. All the functional requirements are mapped to use cases in the table 1.1.

UC1: Viewing upcoming trips

The user can access upcoming trips on the list screen by selecting the upcoming trips tab.

UC2: Viewing completed trips

The user can access completed trips on the list screen by selecting the completed trips tab.

UC3: Creating a trip

The user will be transferred to the create screen by clicking the create button on the home or upcoming trip screens. He will select the trip's name, date, and itinerary there. Then, the user will save the trip by clicking the tick icon at the top of the screen.

UC4: Searching a place

On the create or edit screen (UC3, UC11), the search screen will be shown after clicking the search button. By pressing the search bar, the user can write the location under which the results will be displayed.

UC5: Using current location

When adding a first place to a trip on the create or edit screen (UC5, UC11), the user has the option to add the current location as the first place by clicking the use current location button.

UC6: Adding a place

After clicking a location on the search screen (UC4) or the current location (U5), the location will be added to the itinerary.

UC7: Removing a place

If the x icon next to a place is clicked on the create or edit screen (UC3, UC11), it will be removed from the trip.

UC8: Changing the place order

On the create or edit screen (UC3, UC11), when the switch at the top of the screen is toggled, the user can change the order of location by dragging and dropping each item.

UC9: Getting trip details

The user can access trip details on the upcoming trips screen (UC1) by clicking the trip.

UC10: Optimizing a trip

The user can optimize the order on the details screen (UC9) by clicking the optimize button.

UC11: Editing a trip

The edit screen will be shown after clicking the edit button on the detail screen (UC9).

UC12: Viewing place details

The user can get place details on the edit screen by clicking the place (UC9).

UC13: Starting a trip

The user can start a trip on the upcoming trips screen or home screen (UC1) by tapping the start trip button.

UC14: Getting directions

When viewing a trip on the home screen, the user gets directions by clicking the place.

UC15: Taking a picture

When a location is highlighted, when the user's location matches the place location, the user can take a picture by clicking the camera button on the highlighted place.

UC16: Finishing a trip

On the home screen, when a trip is started (UC14), the user can finish the journey by clicking the finish button and selecting yes in the dialog.

UC17: Repeating a trip

On the completed trips screen (UC2), the user can repeat a trip by clicking the repeat trip button, after which he will be taken to the create screen (UC3).

Table 1.1 Use case requirements coverage table.

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
UC1									X							
UC2									X							
UC3	X	X	X				X									
UC4		X														
UC5																X
UC6	X	X	X													
UC7	X		X													
UC8			X	X												
UC9				X	X				X							
UC10						X										
UC11		X	X	X												
UC12													X			
UC13							X									
UC14												X				
UC15														X		
UC16										X						
UC17		X	X	X			X	X			X					
UC18				X					X							
UC19			X												X	
UC20			X												X	
UC21			X												X	
UC22			X						X							

UC18: Viewing completed trip gallery

The user can view the trip's gallery on the completed trips screen (UC2) by pressing the trip.

UC19: Editing trip gallery

On the gallery screen (UC19), users can toggle editing mode by clicking the edit button at the top of the screen.

UC20: Adding a picture

The user can add a picture by clicking the add button under a place on the edit gallery screen (UC20).

UC21: Removing a picture

The user can remove a picture by clicking the x button on a photo on the edit gallery screen (UC20).

UC22: Removing a trip

The user can remove a trip by clicking the trash icon at the top of the gallery or detail screen (UC9, UC19).

Chapter 2

Design

The design phase in software engineering is a crucial step that bridges the gap between understanding what a software application needs to do and how it will be built. During this phase, the analysis outputs will be put into a blueprint for the application. This includes choosing the right platform and architecture, choosing the correct technologies, and designing the application's user interface.

2.1 Application Platform

When deciding the right mobile platform for the application, it is essential to consider the market share of mobile operating systems. According to statistics, in the year 2023, Android is in the lead with a 70.11% market share. The second is iOS, with 29.19%. The remaining part, at 0.7%, comprises multiple operating systems. Due to their unpopularity, they will not be a focus. [8]

When considering the previous statistics, Android should be a clear choice. However, an alternative conclusion may be drawn considering the prerequisite that individuals must possess financial resources for travel. When changing the focus from the world data to the United States, iOS suddenly makes up more than half, with 57.5% [9]. For this reason, it would be best if the app were available for both platforms. There are multiple ways of achieving this. Firstly, a native application for each system, and secondly, a cross-platform solution.

2.1.1 Native

A native application is one that was designed with the purpose of operating on a single platform. Programming languages and tools specific to a given platform are used to create these apps. For instance, one can build native Android apps with Kotlin and iOS apps with Swift.

Native applications offer advantages such as higher app store ranking, access to all platform tools, higher user satisfaction, enhanced compatibility with code and device resources, and easier publishing. However, they have disadvantages, such as limited use on single platforms due to increasing platform diversity. They are costlier and more time-consuming than cross-platform applications due to different code sets for distinct platforms. [10]

2.1.2 Cross-platform

A cross-platform application is one that functions across multiple platforms as opposed to just one. There are two extents to which this can be realized.

2.1.2.1 Full Cross-platform

Flutter is a graphical engine that simplifies application development by bypassing the native software development kit. It is ideal for simpler applications with identical UIs across platforms, allowing for rapid development with fewer developers. Additional Flutter packages are required to achieve native functionality. [11]

React Native is similar to the Flutter framework in many ways. Once more, the unified UI implementation makes developing applications in smaller teams feasible. However, there is still the problem of dependency on external libraries to provide more extensive functionality. Both frameworks separate the user from native development, which makes it simpler but also limits the developer's options. React Native is unique because it uses native graphical elements to enable the development of applications that maintain some degree of the native user interface's appearance and feel. Nevertheless, coupling the application and native API, the use of JavaScript, and the runtime architecture results in lower performance than Flutter offers. [11]

Compose Multiplatform, similar to React Native and Flutter, aims to streamline mobile application development across various platforms using a single codebase. However, as of this day, it is in its early days, with iOS implementation still being in alpha. The future of this framework has yet to be set. Hence, it cannot be reliably used for any commercial product. [12]

2.1.2.2 Semi Cross-platform

Kotlin Multiplatform Mobile technology is a multiplatform approach that closely resembles native development, allowing for code sharing and reducing issues caused by bypassing the native framework. Application networking, data storage and validation, analytics, computations, and other application logic can be shared across platforms. However, as can be seen in the diagram 2.1, it does not allow for common source code for the presentation layer, increasing development costs but offering greater versatility and adaptability. Kotlin's focus on Android makes it easier for native developers to adapt and allows for a gradual transition to the technology on a smaller scale. [11, 13]

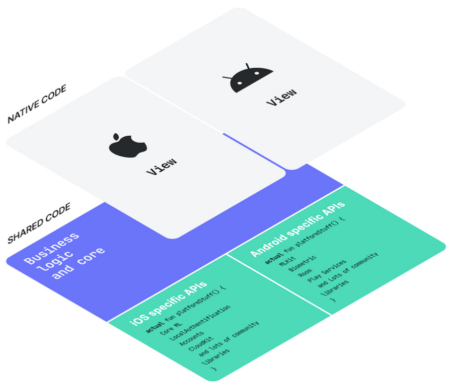


Figure 2.1 Kotlin Multiplatform Mobile [13].

2.1.3 Conclusion

Native development offers performance benefits but requires maintaining separate codebases for different platforms. Cross-platform frameworks like Flutter and React Native offer faster development but compromise on performance and native feel. Compose Multiplatform is promising but lacks maturity for critical projects.

Therefore, considering the native development feel, code sharing, and high performance, Kotlin Multiplatform Mobile emerges as the best option for this project. KMM allows developers to leverage their existing Kotlin expertise while enabling efficient cross-platform development. This approach balances development speed with the ability to create high-quality applications for both Android and iOS.

2.2 Application Architecture

When developing an application, it is essential to choose the correct architecture. Many have emerged over the years, each with its advantages and caveats. The most used in mobile applications are MVC, MVP, MVVM, Clean Architecture, and MVI.

2.2.1 Model-View-Controller (MVC)

MVC is one of the oldest and most well-known architectural patterns. It is composed of three components: The model is the component that holds the logic business logic and is in charge of accessing, manipulating, or storing data in the application. Classes related to data persistence, application communications, and the classes used for parsing outside information are all stored here. The view is the component for the user to see. Its purpose is to show data gained from the model. The controller handles all the communication between the previous two. It is the main component of MVC. It receives and interprets the user's actions from the view and passes them to the model. Vice versa, if the model changes, it updates the view. [14]

It provides clear distinctions between the data, UI, and logic. Therefore, it is a great option for prototyping small projects; however, it can lead to tight coupling between the components, making it less reusable and more difficult to scale, maintain, or test. [15]

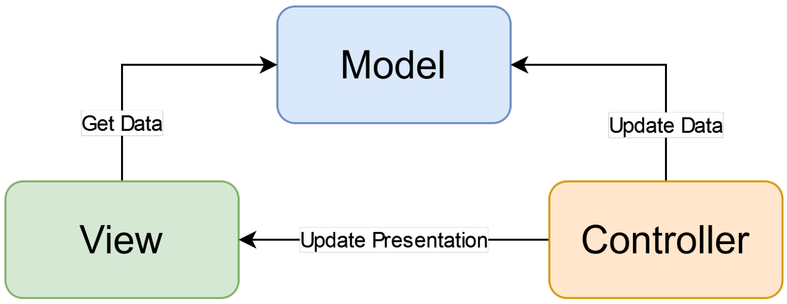


Figure 2.2 Model-View-Controller architecture.

2.2.2 Model-View-Presenter (MVP)

MVP is an evolution of MVC that further divides UI and business logic concerns. The Model is a component in the MVP pattern responsible for business logic, data persistence, communication, and transforming external information into model objects. It contains classes related to data persistence, extensions, and constants. In MVP, the Model layer communicates only with the Presenter layer, unaware of the existence of a View. User interactions with the View are transmitted to the Model through the Presenter, and the Model is updated accordingly. The View layer combines the View and Controller components in one layer. Both components store less logic, making them lighter. The Controller handles coordination and routing functions, navigation, and information passing via Delegation. In MVP, the Controller instantiates the View and passes it to the Presenter. The Presenter receives and passes View events to the Model, updating the View when data changes. [14]

The MVP is used for medium-sized projects with more complex requirements. The clear separation between the components makes it easier to test and understand. It promotes the Single Responsibility Principle[1]. However, as the project grows, it may still have much boilerplate code[2] and become more complicated. [15]

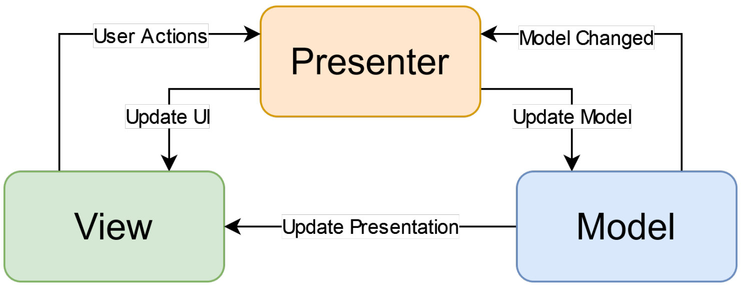


Figure 2.3 Model-View-Presenter architecture.

2.2.3 Model-View-ViewModel (MVVM)

MVVM is a current architectural style that uses data binding and reactive programming techniques. The Model, like in MVP or MVC, is a component responsible for business logic and application data manipulation and access. This layer communicates only with the ViewModel layer, unaware of a View's existence. The View displays user-updated information from the ViewModel without logic and can have multiple references. The ViewModel is the central element of the MVVM architecture, interacting with the View and Model. It represents the View's current state and handles input/output information. The View owns the ViewModel, and the ViewModel owns the Model. ViewModel and the View are connected by what is known as Data Binding[3], ensuring automatic UI updates if the ViewModel's information state changes. [14]

MVVM is ideal for large projects focusing on data-driven UIs and complex data flows. It offers a clear separation between View, ViewModel, and Model, automatic UI updates, and better decoupling, testability, and maintainability. However, it requires an understanding of reactive programming and data binding concepts. [15]

[1] A module should be responsible to one, and only one, actor. [16]
[2] Sections of code repeated in multiple places with little to no variation. [17]
[3] The process that establishes a connection between the app UI and the data it displays [18].

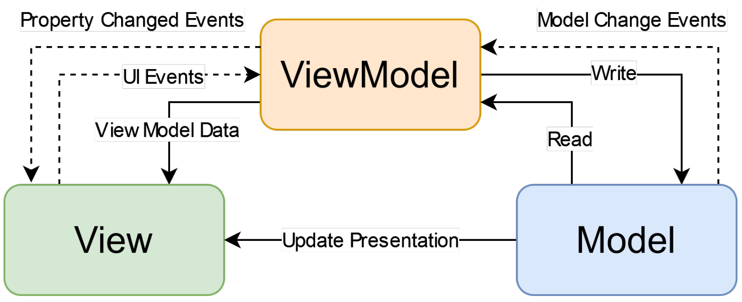


Figure 2.4 Model-View-ViewModel architecture.

2.2.4 Clean Architecture

Clean Architecture is a software design principle introduced by Robert C. Martin, known as "Uncle Bob," that focuses on organizing code and defining the architecture of a software application to achieve better maintainability, flexibility, and testability. The central concept behind Clean Architecture is the separation of concerns, which emphasizes that different parts of the application should have clearly defined responsibilities and be independent of each other. This allows each component to be developed, tested, and modified without affecting others, leading to a more robust and maintainable codebase.

The architecture is usually divided into concentric circles 2.5, each representing a different application layer. The Domain Layer, the heart of the application, contains the core business logic and entities. The Application Layer serves as an intermediary between the Domain Layer and the Presentation Layer, containing application-specific use cases and orchestrating data flow between the Domain Layer and external interfaces. The Presentation Layer handles user interactions and displays information, while the Data Layer handles data access and storage.

The key benefit of Clean Architecture is the development of independent and interchangeable components, allowing for changes in database technology, user interface, and overall test coverage. Emphasizing the separation of concerns and organizing code into distinct layers promotes modularity, maintainability, and testability in software development, helping teams build scalable and maintainable applications over the long term. [19]

Clean Architecture is ideal for large, complex projects focusing on maintainability and testability, but it can introduce complexity and over-engineer for smaller projects. [15]

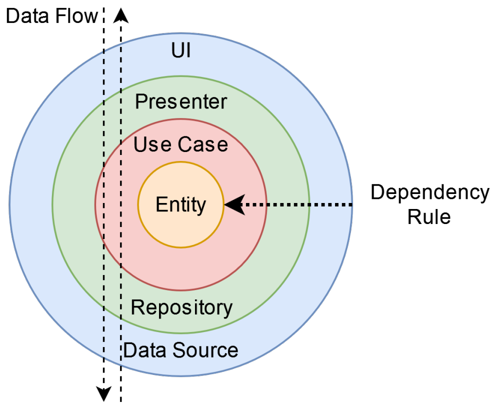


Figure 2.5 Clean architecture.

2.2.5 Model-View-Intent (MVI)

MVI is an emerging architectural pattern that aims to provide a predictable and reactive UI. The architecture features a single direction of data flow, ensuring clarity and predictability. It separates concerns for Model, View, and Intent components, with Model managing state, View handling UI rendering, and Intent capturing user actions. Immutability ensures the Model's state remains unchanged, promoting predictability and reliability in the application.

The user's intent is a state input to a model, which stores and sends the requested state to a view. The View loads the state from the Model and displays it to the user. This flow happens only in one direction. Hence, it is called unidirectional architecture. The user can do another action that follows the same flow. Therefore, it is also cyclic. [20]

MVI is suitable for projects requiring strict unidirectional data flow and interactive UIs but can introduce additional complexity, especially for simple projects. [15]

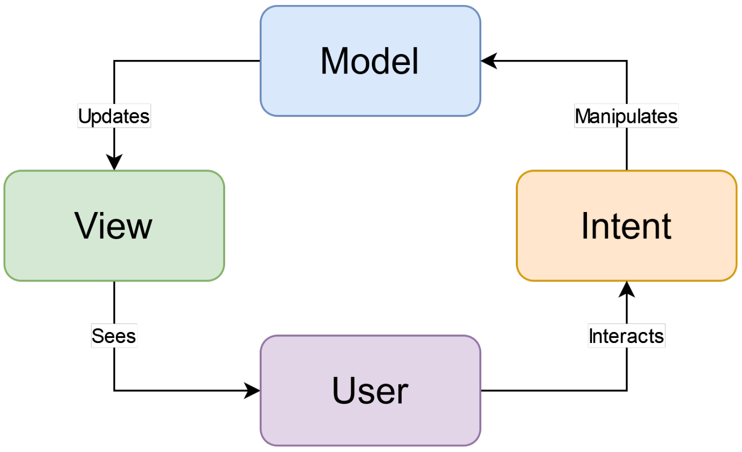


Figure 2.6 Model-View-Intent architecture.

2.2.6 Conclusion

Clean architecture emerges as the best option for this project. KMM projects stride for reusability across platforms. Hence, clean architecture's focus on independent, interchangeable components supports this idea. It is an excellent choice for projects designed to grow and adapt to the users' preferences because it allows for a seamless feature addition or change. Finally, since the concerns are clearly separated, adjusting to using different UI frameworks is simple.

2.3 Chosen Technologies

When creating an application, it is crucial to identify and choose the correct technologies based on the requirements. This section will outline the key technologies that have been chosen and explain why they were selected.

2.3.1 Jetpack Compose

Regarding native Android UI, there are two main options: view-based XML or Jetpack Compose. Jetpack Compose is a toolkit made by Google based on declarative Kotlin API. The second option is a traditional approach that defines UI layout in XML files and handles interactions in code. Jetpack Compose will be chosen for this project as it is Google's preferred option. They list four main reasons why it is the better option to use [21]:

- **Less code**: Compose is a tool that allows developers to write less code, reducing the number of lines needed for various components. This reduces time spent on testing, debugging, or fixing bugs and allows developers to focus more on delivering value to customers. The code is written in Kotlin, making it easier to understand and maintain.
- **Intuitive**: Compose is a declarative API that simplifies UI design by allowing developers to describe the UI. Its intuitive APIs are easy to learn and use, allowing a single Kotlin file to substitute multiple XML files. Compose also allows for creating stateless components, making them easy to reuse and test. The state is explicit and passed to the composable, ensuring a single source of truth for the UI.
- **Accelerated development**: Compose is compatible with existing code, allowing easy integration with common libraries like Navigation, ViewModel, and Kotlin coroutines. Its interoperability makes it seamless for developers. Android Studio support, including live previews, allows faster iteration and code shipping. This feature saves time by allowing developers to check UI components in different states or settings.
- **Powerful**: Compose is a tool that allows developers to create beautiful apps with direct access to Android platform APIs and built-in support for Material Design, Dark themes, animations, and more. It improves the accessibility APIs, layout, and reduces the number of steps to achieve developers' ideas. Compose also makes it easy to add animations, such as color, size, and elevation changes, without requiring anything special. It provides flexibility for implementing own design systems.

2.3.2 Ktor

Ktor is an open-source framework for creating asynchronous servers and clients using the Kotlin programming language developed by JetBrains. It has been chosen as it is the most powerful and used solution that fully supports KMM. Hence, there is no need to implement separate client logic for either platform. Ktor focuses heavily on being a lightweight and modular solution, adding functionality to a project without much boiler code and allowing the developers to choose only the needed components. Furthermore, Ktor leverages Kotlin features, such as having extension functions instead of annotations or being built on Kotlin coroutines. This makes it more intuitive for developers already familiar with Kotlin. [22]

2.3.3 SQLDelight

SQLDelight is a robust database library specifically made for Kotlin-based projects. Its primary goal is simplifying database operations with a type-safe and compile-time approach. Unlike traditional SQL libraries, it allows developers to write queries directly into the code. SQLDelight generates type-safe Kotlin APIs from SQL statements, underlying efficiency, and reliability of the data access code. It streamlines the writing and maintenance of SQL code by offering features like schema verification, compile-time migrations, and IDE support. SQLDelight also allows switching the underlying database. It supports multiple platforms like SQLite, MySQL, Postgres, or HSQL/H2. [23]

It is an ideal database library for this project because of its cross-platform compatibility. Thanks to platform-specific drivers, the same database interactions can be written for both iOS and Android implementations. The type-safe queries catch errors at compile time, reducing runtime exceptions. The automatic Kotlin code generation makes it more straightforward to use without writing many lines of boilerplate code. It also supports coroutines for asynchronous data access. Lastly, thanks to an IDE plugin that adds features like syntax highlighting and auto-completion, developers can focus more on the logic and less on the syntax. [24, 23]

2.3.4 Koin

Dependency Injection is a software design pattern that decouples application components and manages their dependencies flexibly and modularly, transferring responsibility from the dependent component to an external entity.

Koin is a lightweight dependency injection framework for Kotlin that simplifies managing dependencies in KMM projects, allowing code decoupling and making testing and maintenance easier.

It was chosen for this project because it lessens the coupling between all the components in the application's hierarchy. It allows the definition and management of dependencies in a shared module that can be utilized across different platforms, such as iOS and Android. It also seamlessly integrates with the frameworks of the mentioned platforms, keeping it more consistent. Koin furthermore allows the definition of dependencies at runtime, giving flexibility when the dependencies need to be resolved dynamically. It is a lightweight framework with an intuitive API, leveraging Kotlin's features like DSLs and extensions. [25]

2.3.5 Fused Location Provider API

Regarding location on Android, there are two main possibilities: using Android's built-in Location Manager of a Fused Location Provider from Google Play Services.

The Location Manager class in the Android framework enables access to a device's location services, allowing users to request updates from various providers like GPS, network, and other applications. It provides a basic set of APIs for determining the device's location. However, managing it is more challenging and requires more code than the Fused Provider. [26]

The Fused Location Provider is a Google Play services library API that simplifies accessing location information from GPS, Wi-Fi, and cellular networks. It combines data from multiple sensors for accurate and efficient updates. Recommended for location-aware applications, it offers a more advanced and flexible API than the Location Manager. [26]

Since the application's prototype must track the user actively to update their itinerary progress, the Fused Location Provider will be used.

2.3.6 Coil

There are plenty of Android libraries for image loading. However, between them shines a newcomer, Coil. It will be used for this prototype because it is lightweight and designed with a focus on Kotlin, utilizing coroutines and extension functions to simplify image loading. Its lightweight nature ensures optimized performance without unnecessary overhead. Coil's automatic memory management prevents memory leaks and out-of-memory errors, ensuring stable apps. Built with modern Android development practices, Coil remains up-to-date with the latest libraries and APIs for compatibility and stability. It prioritizes performance, efficiently minimizing network requests and maximizing image loading speed, even with high-resolution images. [27]

2.3.7 Maps

Because this application is made for itinerary creation, a platform for getting places, searching places, getting distances, and reverse geocoding is necessary. Three options were considered when deciding the best solution: Google Maps Platform, Mapbox, and OpenStreetMap.

GMP[4] is a suite of APIs and SDKs that allow developers access to Google's geographical and location-based data. It offers features like map visualization, search for points of interest, geocoding, or distance matrix. Furthermore, GMP provides specialized functionalities like air quality, traffic conditions, and elevation. While underlying data details are not publicly available, it has been established that Google relies on its extensive collection of user data, satellite imagery, and algorithmic processing. Thanks to this, it reliably delivers actual and accurate information. Combined with the vast user base and Google's stride to improve it continuously, it is one of the best solutions for location-centered applications. [28]

Mapbox is a cloud-based platform of developer tools for integrating map services into their applications. It specializes in map creation and navigation but offers many more functionalities. In contrast to GMP, which relies on proprietary data, Mapbox leans towards an open-source approach. It uses OpenStreetMap data, a community-based model that can sometimes be inaccurate. However, this data is not its only source. Mapbox uses this open-source base and builds on this by adding features like offline maps or having more comprehensive documentation. It also undermines Google Maps with their pricing, which is consistently lower. [29]

OpenStreetMap is a similar solution that uses an entirely different approach. It does not rely on proprietary data but emphasizes community-driven data. OpenStreetMap is completely open-source and encourages everyone to actively participate in mapping their surroundings. This approach might not always be accurate. However, it can leverage the local knowledge of its user base more. This community approach is its biggest strength and weakness. Developers are not locked in one vendor and can edit or add new data. On the other hand, the coverage is imperfect as it does not have a big user base, and the data can be outdated. [30]

GMP will be chosen for this application as it provides more extensive, up-to-date, and accurate data. Furthermore, GMP provides images of places and generally more information about them, making the application prototype more informative and intuitive. All the relevant information for the application is summarized in the table 2.1.

[4] Google Maps Platform

Table 2.1 Comparison of the three maps services [28, 29, 30].

	Google Maps Platform	Mapbox	OpenStreetMap
Ease of use	high	good	difficult
Places (Search)	Yes (extensive)	yes	limited
Places photos	yes	no	no
Distances	yes	yes	yes
Reverse geocoding	yes	yes	yes
Open-source	no	partially	yes
Cost	free tier, paid plans	free tier, paid plans	free, but may require self-hosting
Data Accuracy	high	good	variable

2.4 Domain Diagram

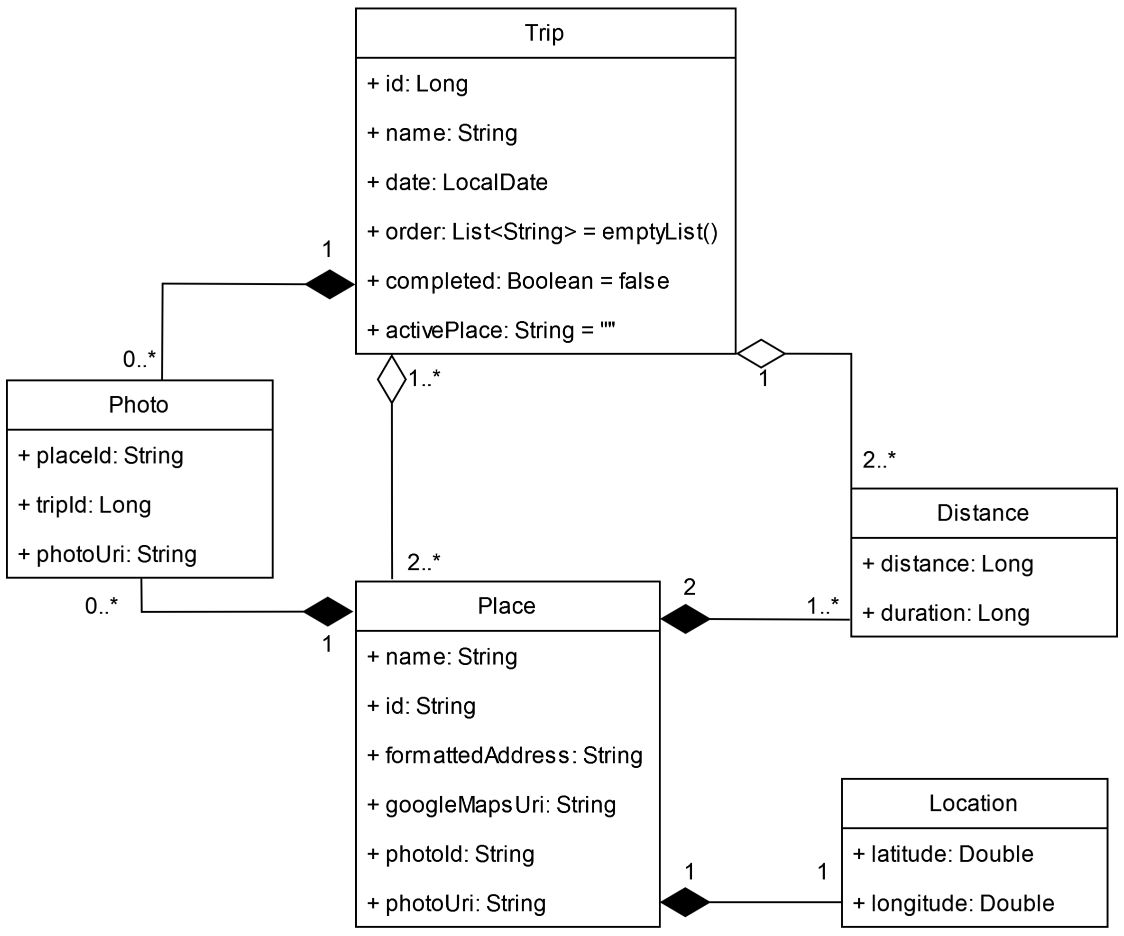
A domain diagram is a foundational tool for capturing the core concepts and their relationships within a specific problem domain. This visual representation, employing UML notation, depicts entities, attributes, and associations relevant to the system's functionality. [31]

The model proposed for this application is depicted in the following diagram 2.7. The Trip entity is the cornerstone of the entire application; it has relations with Places, which create the itinerary for the trip. The Distance entity relates to both the trip and two places as the distances between places are accessible from the trip; however, more importantly, it portrays a distance between two distinct places. Photos are accessible from a trip, like distances, but belong to the place where the picture is taken. Each Place has one associated Location.

2.5 Activity Diagram

Activity diagrams, a part of the UML, provide a detailed visual representation of dynamic system behavior. By illustrating the sequential and concurrent flow of actions, decisions, and iterations, activity diagrams effectively model complex processes, aiding in requirement analysis, design specification, and validation. [31]

The process of creating a trip is shown in the following figure 2.8. To create a trip, three main components are required: the name of the trip, the date, and the itinerary. For the itinerary to make sense, at least two places are required. To add more places, the user needs to use the search feature; based on the user's query, a list of locations is returned from the Places API. After the details are entered, the user can save the trip. When saving, the user is prompted to optimize the trip. The optimization then happens in the domain; in both cases, the trip is saved in the database.

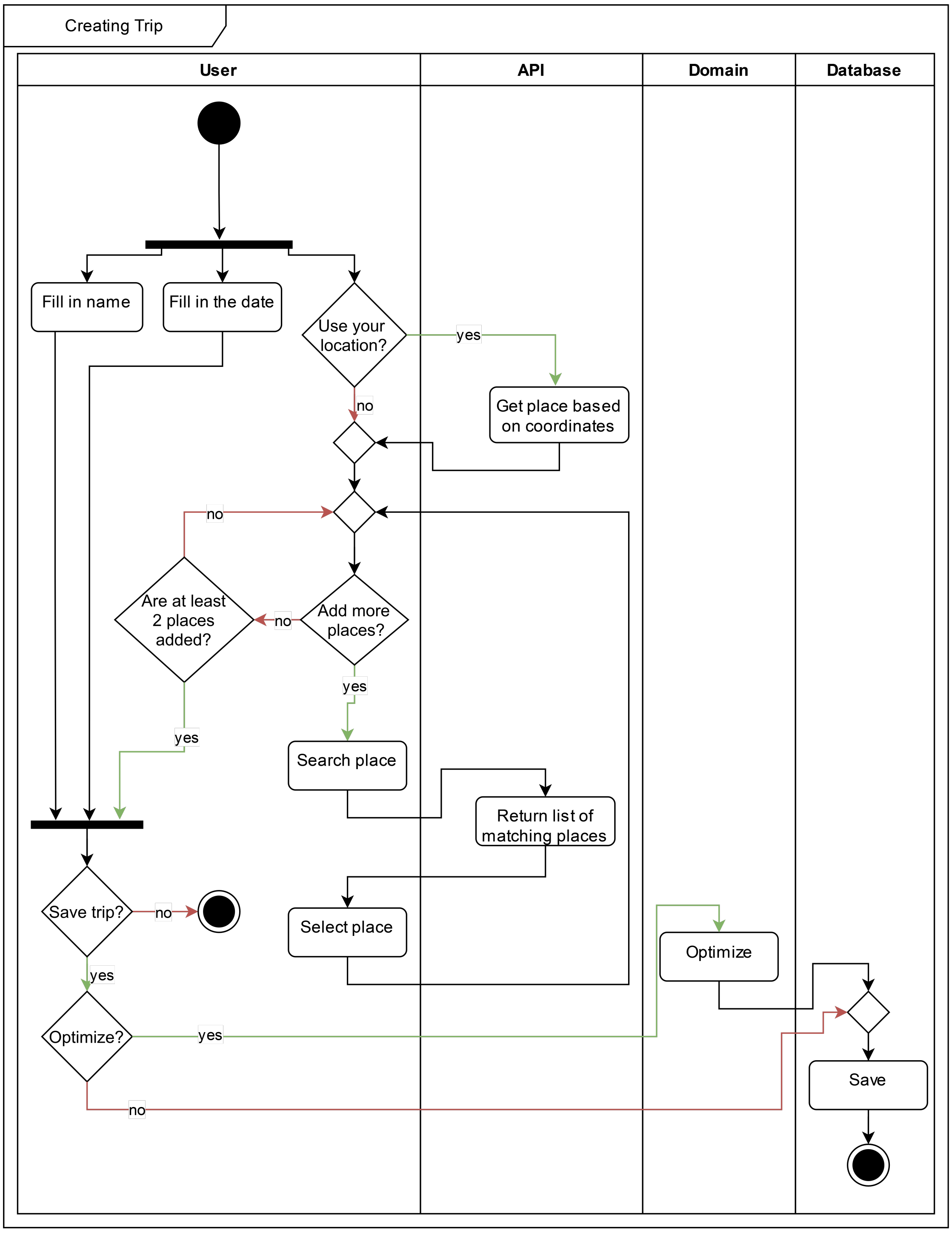


■ **Figure 2.7** UML Domain diagram.

2.6 State Diagram

A state diagram visually represents the states an object or a system can be in and the transitions between them. It illustrates how events trigger changes from one state to another, aiding in understanding complex system behavior and ensuring correct implementation. [31]

A diagram in figure 2.9 shows the different states a trip can have. A trip follows a sequential life cycle within the application. It begins in a Created state, where it is initially added to the system. Optionally, it can transition to an Optimized state depending on the user's choice. Next, the trip moves to the Started state, indicating it is actively underway. Upon completion, it enters the Completed state. At any point before starting or after completing, the trip can be removed from the system and moved to the Deleted state.



■ **Figure 2.8** UML activity diagram of trip creation.

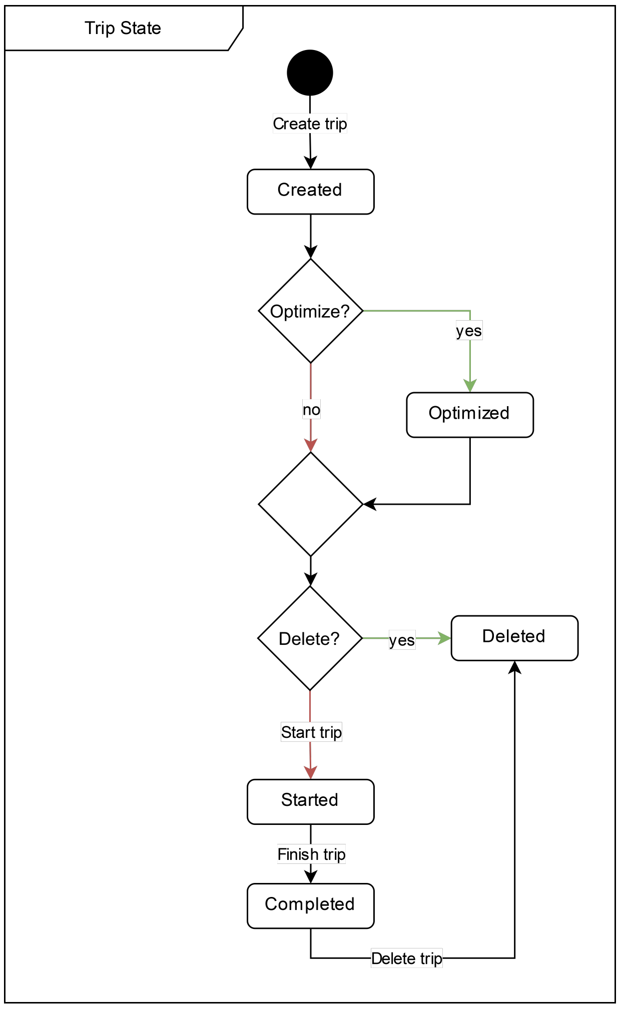


Figure 2.9 UML state diagram of a trip entity.

2.7 User interface

The UI is one of the most crucial aspects of designing a mobile application. It is how users interact with the app, so it must be intuitive to motivate users to keep using it. Designing a wireframe is generally a good idea before implementing the UI in the platform-specific framework.

"*Wireframes are basic blueprints that help teams align on requirements, keeping UX design conversations focused and constructive* [32]." They contain the base of the design formed primarily by abstract shapes, and the text is usually randomly generated. It serves just as a mere idea of a concept. They are helpful as they allow iteration and refining of ideas before investing in detailed visuals or code, thus minimizing rework and ensuring a smoother development process.

To create a wireframe, a designer does not need anything more than a piece of paper. However, for the wireframe to be more valuable, it can be created in one of the many apps designed for this purpose. Figma was chosen for this project because of its intuitive UI. Two of its features will be used: drawing wireframes and prototyping. Prototyping allows simple interaction with the wireframe. Hence, it can be adequately tested and the design iterated upon.

2.7.1 Application's Design

Based on the first chapter's analysis, the application prototype was designed with user-friendliness in mind. The main navigation was designed with a navigation bar at the bottom of the screen so that navigation between the main parts of the application is simple. This bar features a home screen, a list screen, and a create screen. These make up the central part of the application. The following list illustrates the ideas behind the design of these screens:

- **Home Screen:** This screen guides the user during travel. It was made to display the itinerary clearly, with the time remaining until the trip and the trip's name on the top of the screen. The itinerary comprises small cards highlighting the place's name, the approximate time of stay, and the image. Image is an essential part of the card, allowing users to identify a place quickly upon arrival. Every card also has a button to get directions to the place in the user's desired application. Another important feature of the application is that upon arrival at a place, the card is expanded, allowing the user to take pictures and see a small gallery of already-taken pictures.

  Furthermore, between all the place cards is a small indicator of how far to the next place. Lastly, a floating button in the bottom right corner highlights an option related to the trip's current state. This means that when the trip has not started, it starts early; during the trip, it allows the user to finish it at any point. If there is no upcoming trip, a button is shown in the middle of the screen with a prompt to create a new one.

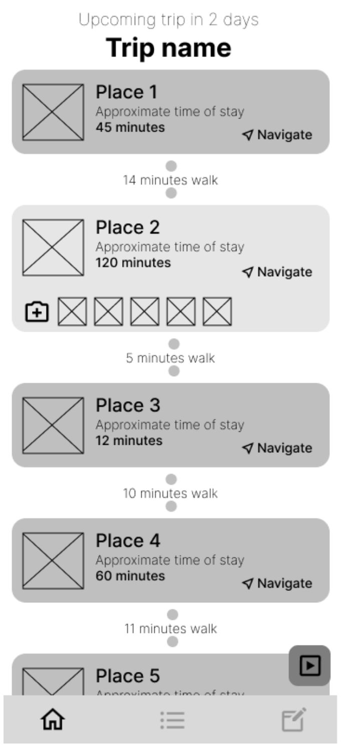
- **List Screen:** It is a screen allowing the user to see all their trips straightforwardly. This screen comprises two main categories of trips: already completed and not completed. Navigating between these two categories is achieved via tabs at the top. A trip card, similar to places, represents every trip. This card shows all the essential details to identify a trip. Firstly, a user-created name is displayed, with the city's name under it. In the upper right corner is the date the trip happened or is about to happen.

  Moreover, there are two buttons depending on the selected tab. In the upcoming screen, one navigates to the trip's details, and the other serves to start the journey immediately. On the completed screen, one repeats the trip, and the other navigates to the trip's gallery. In the lower right corner of the screen lies a floating button that navigates the user to the create screen; this button is only visible on the new trips tab.

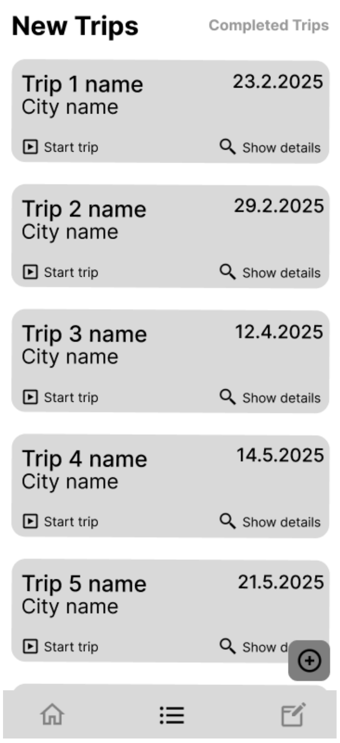
- **Create Screen:** This screen serves to create a new trip. Firstly, there are two text boxes at the top of the screen; the first is the trip's name, and the second is the date of the trip. Then, there is the starting place, with the rest of the places in the itinerary under it. These comprise a modified place card, highlighting the place address with a button that opens the place in the Google Maps application. New places can always be added by clicking the plus button in the list. Lastly, there is a tick button in the top-right corner to save the trip in the application.

- **Gallery screen:** The gallery screen is a simple screen showing the user photos taken in a horizontal list organized by the place where they were taken.

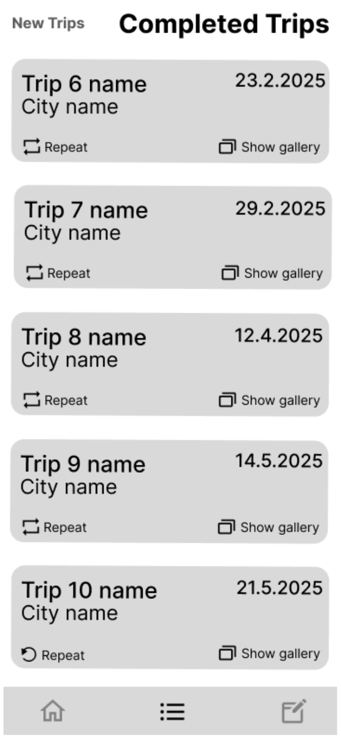
Moreover, there are three more screens: trip details, its corresponding edit screen, and the gallery edit screen. These screens, however, are designed in the same manner as the already-introduced screens and, therefore, do not need separate explanations. The main screens accessible from the bottom bar are shown in figure 2.10 with the rest of the screens in the appendix A.



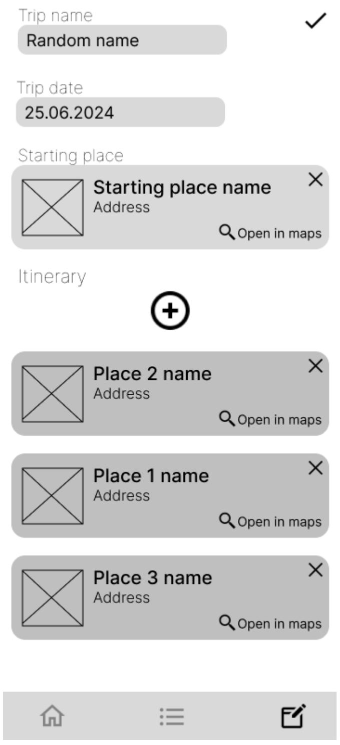
(a) Home screen.



(b) New trips screen.



(c) Completed trips screen.



(d) Create screen.

Figure 2.10 Wireframes created in Figma.

2.7.2 Design Testing

After creating the wireframe, it was transformed into a low-fidelity prototype that allowed first UI testing. For this prototype, the following test scenario was created:

1. Navigate to the trip creation screen.
2. Enter the following trip details:
   - name
   - date
   - starting location
3. Add two places to the trip itinerary:
   - “Place 1” as the first location.
   - “Place 2” as the second location.
4. Create the trip.
5. On the “Smart Order” screen, opt out of automatic ordering.
6. Navigate to the trip details screen.
7. Manually select smart order for this trip.
8. Start the trip early.
9. Upon arrival at “Place 1”, add a picture.
10. Skip to “Place 2” and complete the trip.
11. Navigate to the details of the finished trip screen.
12. Add a picture to the “Place 1”.

Six different people completed this scenario. Three of these people were actively engaged in mobile development; the other three were from different development-unrelated backgrounds.

2.7.3 Results of Testing

Based on the testing results, the following changes were implemented:

- The Create screen was removed from the bottom bar. It was distracting as there was already a button to create a trip on the home and the list screen. This way, the information will not be duplicated in the app.
- The starting place card was moved to the rest of the itinerary. There is no need to differentiate between the start and the rest of the itinerary other than being first on the list.
- Primary button on a card: navigate, show details, show gallery, and open in maps was removed. Its action will happen after clicking the card.
- New trips were renamed to Upcoming trips for more clarity.
- All of the floating buttons will be expandable, describing the action that happens after pressing.

- The Optimize trip button was moved from the top of the edit screen to an extended floating button on the trip detail screen. Since it is a crucial app feature, there it is more visible.
- The reorder itinerary button was changed to a switch so it is clearly visible when ordering is toggled.
- On the create screen, the itinerary order has been reversed to correspond to the naturally perceived order from top to bottom.

Chapter 3

Implementation

In this chapter, the design is translated into an actual product, a prototype of a mobile application. The tools used to implement the application will be stated. Then, important parts of the code will be highlighted based on the chosen architecture. Lastly, screenshots of the implemented application will be shown.

This application was designed to be multiplatform. However, during the implementation, there was no access to a device that would compile iOS code. For this reason, the UI was only implemented for the Android platform using KMM architecture. Business logic is stored in a shared platform, and UI-specific code is stored in a separate Android module.

The project was started from a devstack [33] to guide the application to be written according to the norms and structurally sound.

3.1 Kotlin

Kotlin was used as the primary language for this project because, on May 7, 2019, Google announced that after two years of official support for Kotlin, Android development will become Kotlin-first. All new Jetpack APIs and features would come to Kotlin before Java. [34]

Kotlin is a statically typed, object-oriented programming language interoperable with the JVM[1], Java Class Libraries, and Android. Originally designed to improve Java, Kotlin is used for various application types, including Android mobile app development, server-side development, full-stack web development, multiplatform mobile development, data science, and cloud-based resource management. [35]

Kotlin's advantages include interoperability, safety, clarity, tooling support, and community support. It can be compiled into JavaScript or an LLVM encoder, enabling just-in-time compiling and easy migration of Java applications to Kotlin. It also features null safety, eliminating null pointer exception errors and eliminating redundancy in basic syntax. [35]

Kotlin also provides more concise code, reducing redundancy in the syntax of popular languages like Java. It has tooling support from Android, including Android Studio, Android KTX, and Android SDK, and a community of developers who work to improve the language and provide

[1] Java Virtual Machine

documentation. Despite being the preferred development language of Android, Kotlin's interoperability with Java has led to its widespread use across various application types. [35]

3.2 Development Tools

This section will list the tools used to implement the prototype.

3.2.1 Android Studio

The entirety of the prototype was programmed in Android Studio. It is Google's official IDE[2] for the Android operating system, based on JetBrains' IntelliJ IDEA software. It is available for download on Windows, macOS, and Linux operating systems and is licensed under the Apache license. Originally announced in 2013, the first stable version was released in December 2014. It is now the sole officially supported IDE for Android development. Android Studio supports all IntelliJ and CLion programming languages with extensions such as Go. Android Studio 3.0 or later supports Kotlin. [36]

3.2.2 Git

To keep the code safe, always up-to-date, and synced across devices used for development, the FIT CTU Gitlab was used. However, for the supervisor to be able to see changes made to the application prototype, it was also versed on Github under the author's private account.

3.3 Kotlin Multiplatform Mobile Structure

Since KMM was chosen for this application, a specific project structure was set for this application to adhere to its conventions. This structure typically consists of three main parts of the project: shared, android, and iOS. However, since the iOS UI was not yet implemented, only two remain.

3.3.1 Shared module

This module illustrates an advantage of the KMM development approach: the code written here does not need to be duplicated across platforms. It contains the domain and data layer of the application, housing the core business logic, models, services, and everything else related to data handling. This is all housed in the directory named *commonMain*. However, as mentioned previously, some features need a platform-specific implementation. This involves database drivers, client engines, or location controllers. These are located in their respective *iosMain* or *androidMain* directories.

[2]Integrated Development Environment

3.3.2 Android Module

Because a semi-cross-platform approach was chosen for this application prototype, the presentation layer needs a separate implementation for each platform, in this case, Android. This trade-off allows for a native-like UI experience but requires more code. This module is called *android*. Inside, it's divided into directories based on features. In this application, *gallery*, *home*, *search*, and *trip* were the different parts of the UI. These are all connected through the *app* directory, which contains the root of the application with the *MainActivity* class. The last directory contained in this module is *shared*. The *shared* directory contains everything shared across the different parts, like reusable Jetpack Compose UI components or abstractions like the *PermissionRequest* class.

3.4 Domain layer

In clean architecture, the domain layer represents the center of the application. It encapsulates the application's business logic and should be independent of all frameworks, so it is not specific to any platform. Furthermore, while it deals with data, it does not see the implementation of the data. It uses abstraction over the data; therefore, the data sources can be changed without affecting the application's core.

3.4.1 Model

The application's model represents the core concepts of the domain. It is made of entities, which are simple data objects.

Trip 3.1 is the entity that represents a trip that is the base of the entire application. The data class contains the trip's name, date, and itinerary. The order value represents the order of the itinerary. Completed represents the trip's state, whether it is completed or not. ActivePlace represents the place that matches the device's current location. Photos contain all users taken or added from the library pictures. Distances are the last value in this entity, which serves to keep distances between all places in the trip's itinerary.

```kotlin
data class Trip(
    val id: Long,
    val name: String,
    val date: LocalDate,
    val itinerary: List<Place>,
    val order: List<String>,
    val completed: Boolean = false,
    val activePlace: String = "",
    val photos: List<Photo> = emptyList(),
    val distances: Map<Pair<String, String>, Distance> = emptyMap(),
)
```

■ **Code listing 3.1** Trip entity.

Trips' itinerary comprises **Place** entities 3.2. Every place has a name, a unique ID, an address formatted to a human-readable format, a location, and a Google Maps address. Furthermore, it contains an ID of a photograph from a Google database and a link to that photo.

```
data class Place (
    val name: String,
    val id: String,
    val formattedAddress: String,
    val location: Location,
    val googleMapsUri: String,
    val photoId: String?,
    val photoUri: String? = null,
)
```

■ **Code listing 3.2** Place entity.

Location 3.3 is a simple data class for storing geographical coordinates.

```
data class Location(
    val latitude: Double,
    val longitude: Double
)
```

■ **Code listing 3.3** Location entity.

Distance 3.4 is yet another simple data class. It serves to store distance in both time and the actual geographical distance.

```
data class Distance(
    val distance: Long,
    val duration: Long
)
```

■ **Code listing 3.4** Distance entity.

The last entity is a **Photo** entity 3.5. It stores users' photos containing the exact place and trip to which it is assigned.

```
data class Photo(
    val placeId: String,
    val tripId: Long,
    val photoUri: String
)
```

■ **Code listing 3.5** Photo entity.

3.4.2 Repositories

Repositories in the domain layer define a contract for data access operations. They are defined here, so they are close to the use cases. Their specific implementation lies in the data layer. The example 3.6 shows a part of the trip repository.

```
interface TripRepository {
    suspend fun getUncompletedTrips(): Flow<List<Trip>>
    suspend fun getCompletedTrips(): Flow<List<Trip>>
    suspend fun getTripById(id: Long): Flow<Trip?>
    ...
}
```

■ **Code listing 3.6** Trip repository in the domain layer implementation.

3.4.3 Use cases

Use cases are a part of the clean architecture that defines all the actions possible in the application. They are called from ViewModels and interact with the repositories injected into the constructor. This provides the significant advantage of clean architecture as the application concerns are clearly separated.

Use case interfaces used in this application, shown in the following listing 3.7, are differentiated by their output and input. For example, a simple use case for getting all completed trips does not need a parameter and returns a flow 3.8. However, a use case for removing a trip needs its id as a parameter and returns a Result 3.9.

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

■ **Code listing 3.7** Use case interfaces implementation.

```
interface GetCompletedTripsWithoutPlacesUseCase :
↪   UseCaseFlowNoParams<List<Trip>>
internal class GetCompletedTripsWithoutPlacesUseCaseImpl(
    private val tripRepository: TripRepository
): GetCompletedTripsWithoutPlacesUseCase {
    override suspend fun invoke(): Flow<List<Trip>> =
    ↪   tripRepository.getCompletedTrips() }
```

■ **Code listing 3.8** Use case for getting Completed trips without the optional arguments implementation.

```
interface RemoveTripUseCase: UseCaseResult<Long, Unit>
internal class RemoveTripUseCaseImpl internal constructor(
    private val tripRepository: TripRepository
) : RemoveTripUseCase {
    override suspend fun invoke(params: Long): Result<Unit> =
    ↪   tripRepository.deleteTripById(params) }
```

■ **Code listing 3.9** Use case for removing a trip by its id implementation.

The Result class is an abstraction above any data that can be a success or an error based on the result of the operation. The result class is defined as follows 3.10.

```
sealed class Result<out T : Any> {
    data class Success<out T : Any>(val data: T) : Result<T>()
    data class Error<out T : Any>(val error: ErrorResult, val data: T? = null):
    ↪   Result<T>()
}
```

Code listing 3.10 Result class implementation.

All of the application features can be found in this layer. One of the most critical features of this application, trip optimization, is also executed at this level. This listing 3.11 illustrates a pragmatic approach to trip optimization on mobile devices, emphasizing speed and efficiency. The nearest neighbor algorithm rapidly generates a reasonable itinerary, while the two opt algorithm iteratively improves upon it by swapping pairs of locations to reduce overall distance. This combined approach ensures a responsive user experience, avoiding computationally intensive algorithms, like genetic algorithms or simulated annealing, that could hinder performance on mobile devices.

```
override suspend fun invoke(params: Trip): Result<Unit> {
    return when (
        val distances = distancesRepository.getDistancesByTripId(params.id)
    ) {
        is Result.Success -> {
            try {
                val initialOrder = nearestNeighbor(distances.data,
                ↪   params.order)
                val optimizedOrder =  twoOpt(distances.data, initialOrder)
                updateTripUseCase(params.copy(order = optimizedOrder))
            } catch (e: Exception) {
                Result.Error(TripError.OptimisingTripError)
            }
        }
        is Result.Error -> Result.Error(distances.error)
    }
}
```

Code listing 3.11 Use case for optimizing trips implementation.

3.4.4 Location Controller

The last part of the domain layer is blueprints for features that must be implemented separately for each platform. This is because there can be platform-specific APIs that need to be accessed from multiplatform code. In Kotlin, this is achieved via expected and actual declarations. This works as follows: "*During compilation for a specific target, the compiler tries to match each actual declaration it finds with the corresponding expected declaration in the common code* [37].". One of these platform-specific APIs is the location API. Hence, there is a need for an expected location controller 3.12 and its corresponding Android-specific implementation 3.13 using the Fused Location Provider API 2.3.5.

```
internal expect class LocationController {
    val locationFlow: Flow<Location>
    suspend fun getCurrentLocation(): Location?
}
```

■ **Code listing 3.12** Expected LocationController implementation.

```
internal actual class LocationController(
    private val context: Context,
    private val locationProvider: FusedLocationProviderClient,
) {
    ...
    actual suspend fun getCurrentLocation(): Location? {
        if (permissionGranted) {
            return suspendCoroutine { continuation ->
                val cancellationSource = CancellationTokenSource()
                val token = cancellationSource.token
                val request = CurrentLocationRequest.Builder()
                    .setPriority(Priority.PRIORITY_HIGH_ACCURACY)
                    .build()

                locationProvider.getCurrentLocation(request, token)
                    .addOnSuccessListener { location ->
                        cancellationSource.cancel()
                        location?.let {
                            continuation.resume(Location(it.latitude,
                            ↪   it.longitude))
                        }
                    }.addOnCanceledListener {
                        continuation.resume(null)
                    }
            }
        } else return null
    }
    ...
}
```

■ **Code listing 3.13** Actual Android LocationController implementation.

3.5 Data Layer

The data layer in clean architecture represents the abstraction between the domain layer and data. It handles the specific implementation of data persistence and retrieval, like handling APIs and databases. It contains data sources and the implementation of repositories.

3.5.1 Coroutines

Like many others, this application handles both saving data into a database and getting data from a remote API. These operations are typically longer-running, meaning they cannot be run on the main thread. If this were the case, the application's UI would freeze, and the user would be unable to interact with it. When getting data from a database, the UI must continuously listen to data changes. In this case, the operation must run asynchronously with the UI.

Kotlin handles this through the **Coroutine API**. A coroutine is a suspendable computation that runs a block of code concurrently with the rest of the code, similar to a thread. It is not bound to any particular thread and can suspend execution in one thread and resume in another. They can be considered lightweight threads. [38]

Suspending functions are at the center of using coroutines in Kotlin. A suspending function is a function that can be paused and resumed at a later time. They can execute a long-running operation without blocking and wait for it to complete. [39]

Another essential aspect of asynchronous data access is the **Flow API**. It is a type of data stream that can emit multiple values sequentially, unlike suspend functions that return only one value. It is built on top of coroutines and can have different uses, such as receiving live updates from a database. Flows use suspend functions to produce and consume values asynchronously. They can safely make network requests to produce the next value without blocking the main thread. Three entities are involved in data streams: a producer, which can produce data asynchronously; intermediaries, which can modify each value emitted into the stream or the stream itself; and a consumer, which consumes the values from the stream. [40]

3.5.2 Database

Local data persistence in this application was implemented with SQLDelight 2.3.3. Because this is a KMM library, a platform-specific driver is necessary to create a database. This driver is created via a DriverFactory 3.14, another expected function that needs implementation in iOS and Android modules.

```kotlin
internal expect class DriverFactory {
    fun createDriver(dbName: String): SqlDriver
}
internal fun createDatabase(driverFactory: DriverFactory): Database =
↪   Database(driverFactory.createDriver("trip.db"))

internal actual class DriverFactory(private val context: Context) {
    actual fun createDriver(dbName: String): SqlDriver =
    ↪   AndroidSqliteDriver(Database.Schema, context, dbName)
}
```

Code listing 3.14 Multiplatform database creation.

This database implementation uses .sq files. These files define the tables and queries in an SQL-like language that is not dependent on any specific platform. Based on them, Kotlin classes are automatically generated during the compilation. This automation saves development time and helps maintain consistency between the data definition and the code that interacts with it. Local data sources can then use those classes to implement methods for handling data persistence. The listing, 3.15 shows part of the *Trip.sq* file, which is used to define a table and methods for the Trip entity. The generated *TripQueries* class is then used in the *TripLocalSourceImpl* class, which is shown in the example 3.16.

```sql
CREATE TABLE TripEntity (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    date TEXT NOT NULL,
    place_order TEXT NOT NULL,
    completed INTEGER NOT NULL
);

tripWithPlaces:
SELECT TripEntity.*, PlaceEntity.*
FROM TripEntity LEFT JOIN PlaceEntity ON TripEntity.id = PlaceEntity.trip_id
WHERE TripEntity.id = ?;

getNearestTrip:
SELECT * FROM TripEntity
WHERE date = (SELECT MIN(date) FROM TripEntity WHERE date >= ? AND completed =
↪  0) AND completed = 0;
...
```

Code listing 3.15 Trip SQLDelight file.

```kotlin
class TripLocalSourceImpl(
    private val tripQueries: TripQueries
): TripLocalSource {

    override fun getTripById(id: Long): Flow<List<TripWithPlaces>> {
        return
        ↪  tripQueries.tripWithPlaces(id).asFlow().mapToList(Dispatchers.IO)
    }
    override fun getNearestTrip(): Flow<List<TripEntity>> {
        val date =
        ↪  Clock.System.todayIn(TimeZone.currentSystemDefault()).toString()
        return
        ↪  tripQueries.getNearestTrip(date).asFlow().mapToList(Dispatchers.IO)
    }

    ...
}
```

Code listing 3.16 Trip local data source implementation.

3.5.3 Networking

For searching locations, getting information about them, getting distances, and reverse geocoding, GMP was used. Google offers separate SDKs for iOS and Android. This could be used to make expected and actual functions. However, API was chosen instead of an SDK so as not to duplicate platform code. This way, there needs to be only one implementation using Ktor 2.3.2 for both platforms.

Two endpoints from Place API (New) were used for search, details, and photos implementations. Distance Matrix API was used to get the distance between places. Lastly, Geocoding API was used for coordinates-based address lookup. All the details for all the APIs can be found in the documentation at `https://developers.google.com/maps/documentation`.

Ktor library allows for a definition and creation of an asynchronous client via the *HttpClient* function that takes a platform-specific engine and configuration. This configuration enables tailoring the client to the application's exact needs. In the following snippet 3.17, the *PlacesClient* is implemented.

```
internal object PlacesClient {
    fun init( config: Config, engine: HttpClientEngine, apiKey: String ) =
    ↪   HttpClient(engine){
        expectSuccess = true
        ...
        defaultRequest {
            url {
                protocol = URLProtocol.HTTPS
                host = "places.googleapis.com"
                headers {
                    append("X-Goog-Api-Key", apiKey)
                    append("Content-Type", "application/json")
                }
            }
            contentType(ContentType.Application.Json)
        }
    }
}
```

Code listing 3.17 Ktor client configuration for Places API.

This client is then used by a service to define methods for specific endpoints from the API. The listing 3.18 shows the implementation for a POST[3] request on the text search endpoint. This implementation allows for a biased search based on the location provided, which is then used in the *PlaceRemoteSource* 3.19.

[3]POST method sends data to the server.

```
internal object PlacePaths {
    private const val root = "/v1"
    const val textSearch = "$root/places:searchText"
    ...
}
internal class PlaceService(private val client: HttpClient) {
    suspend fun searchPlaces(
        query: String,
        maxResultCount: Int = MAX_RESULT_COUNT,
        location: Location? = null,
        radius: Int = RADIUS
    ): Result<TextSearchResponse> {
        return runCatchingCommonNetworkExceptions {
            client.post(PlacePaths.textSearch) {
                headers.append("X-Goog-FieldMask", searchFieldMask())
                if(location != null)
                    setBody(
                        TextSearchRequestBody( query, maxResultCount,
                        ↪  location.latitude, location.longitude, radius)
                    )
                else setBody(TextSearchRequestBody(query, maxResultCount))
            }.body()
        }
    }
    ...
}
```

■ **Code listing 3.18** Places API service.

```
internal class PlaceRemoteSourceImpl( private val service: PlaceService,
↪  private val mapsService: MapsService ): PlaceRemoteSource {
    override suspend fun searchPlaces(query: String):
    ↪  Result<TextSearchResponse> {
        return try{ service.searchPlaces(query) } catch (e: Exception) {
            return Result.Error(TripError.SearchError)
        }
    }
    override suspend fun searchPlacesWithBias(query: String, location:
    ↪  Location): Result<TextSearchResponse> {
        return try{
            service.searchPlaces(query = query, location = location)
        }catch (e: Exception) {
            return Result.Error(TripError.SearchError)
        }
    }
    ...
}
```

■ **Code listing 3.19** Place remote data source implementation.

3.5.4 Repositories

In the data layer lies the specific implementation of the repository defined in the domain layer. Here, the source data is translated into the domain data, which is later used by use cases. The implementation of the trip repository shown before 3.6 is in the following listing 3.20.

```
internal class TripRepositoryImpl(
    private val source: TripLocalSource
): TripRepository {
    override suspend fun getUncompletedTrips(): Flow<List<Trip>> =
        source.getUncompletedTrips().map { it.map(TripEntity::asDomain) }
    override suspend fun getCompletedTrips(): Flow<List<Trip>> =
        source.getCompletedTrips().map { it.map(TripEntity::asDomain) }
    override suspend fun deleteTripById(id: Long): Result<Unit> =
        source.deleteTripById(id)
    ...
}
```

Code listing 3.20 Implemented trip repository.

3.6 Presentation Layer

In clean architecture, the presentation layer is the outermost layer interacting with the users. It should not contain business logic as it is only responsible for presenting information from the data layer to the user. It comprises of two main parts:

- **View** implemented using aforementioned Jetpack Compose 2.3.1. The UI components and layout are defined via declarative functions that are annotated with *@Composable*. This represents a single unit, a building block, that can be reused and nested within other composables. Composable functions are lightweight, meaning they can be efficiently recomposed upon a state update.
- **ViewModel** is a class that holds and manages the UI state in a lifecycle-aware way, meaning it survives configuration changes like screen rotation. It acts like a single source of truth for a composable function, holding the data composables interact with and handling UI-related logic.

3.6.1 Navigation

In Jetpack Compose, navigation is managed through a navigation graph. This graph, defined using *NavHost* and *NavGraphBuilder*, specifies the available destinations, composable functions, and the connections between them. A central navigation controller, *NavController*, manages the back stack and facilitates transitions between destinations based on user interactions or internal logic.

This prototype handles the navigation through a bottom navigation bar with two destinations, the home screen and the list screen. The *Root* composable 3.21 is the root of the entire prototype; it holds the bottom bar and the *NavHost*. It has all the navigation graphs in the application. The rest of the screens implemented are accessible through Place or Trip items on the list screen. As an example, the trip graph is shown in the following listing 3.22.

```
@Composable
fun Root( modifier: Modifier = Modifier ) {
    val navController = rememberNavController()
    Scaffold(
        modifier = modifier,
        bottomBar = { BottomBar(navController) },
    ) { padding ->
        Box( modifier = Modifier.padding(padding).fillMaxSize() ) {
            NavHost( navController, startDestination = HomeDestination.route ){
                tripNavGraph(
                    navHostController = navController,
                    navigateToGallery = { id ->
                     ↪  navController.navigateToGalleryScreen(id) },
                    navigateToHomeScreen = {
                     ↪  navController.navigateToHomeScreen() }
                )
                galleryNavGraph( navHostController = navController )
                homeNavGraph { navController.navigateToCreateScreen() }
            }
        }
    }
}
```

■ **Code listing 3.21** Navigation root implementation.

```
fun NavGraphBuilder.tripNavGraph(
    navHostController: NavHostController,
    navigateToGallery: (Long) -> Unit,
    navigateToHomeScreen: () -> Unit
) {
    navigation(startDestination = TripGraph.List.route, route =
     ↪  TripGraph.rootPath) {
        tripListRoute(
            navigateToCreateScreen = {
             ↪  navHostController.navigateToCreateScreen() },
            navigateToDetailScreen = { tripId ->
             ↪  navHostController.navigateToDetailScreen(tripId) },
            navigateToGalleryScreen = { tripId -> navigateToGallery(tripId) },
            ...
        )
        detailScreenRoute(
            navigateUp = { navHostController.navigateUp() },
            navigateToEdit = { tripId ->
             ↪  navHostController.navigateToEditScreen(tripId = tripId) }
        )
        editScreenRoute( navigateUp = { navHostController.navigateUp() })
    }
}
```

■ **Code listing 3.22** Trip graph implementation.

3.6.2 Camera Manager

One of the requirements was that users could take a picture when at the place in their itinerary. This functionality was implemented via a textItrememberCameraManager function. This function creates and returns a textItCameraManager 3.23, a simple abstraction for interacting with the camera, which provides one function, launch, that initiates the process.

Internally, it uses the Jetpack Compose's textItrememberLauncherForActivityResult function to handle the camera intent lifecycle and outcome. A temporary file is created in the application's cache directory upon launch. This file's URI is then passed to the camera intent, specifying the location where the captured image should be saved. If this process succeeds, the photo is copied from the cache to the application's internal memory.

```kotlin
class CameraManager( private val onLaunch: () -> Unit ) {
    fun launch() { onLaunch() }
}
@Composable
fun rememberCameraManager(onResult: (Uri?) -> Unit): CameraManager {
    val context = LocalContext.current
    var tempPhotoUri by remember { mutableStateOf(value = Uri.EMPTY) }
    val cameraLauncher = rememberLauncherForActivityResult(
        contract = ActivityResultContracts.TakePicture(),
        onResult = { success ->
            if (success) {
                val savedUri = saveImageToInternalStorage( context,
                ↪  tempPhotoUri, "picture_${System.currentTimeMillis()}.png" )
                onResult.invoke(savedUri)
            }
        }
    )
    return remember { CameraManager(
        onLaunch = {
            tempPhotoUri = ComposeFileProvider.getImageUri(context)
            cameraLauncher.launch(tempPhotoUri)
        }
    )}
}
```

■ **Code listing 3.23** Camera manager implementation.

3.6.3 Gallery Manager

Another requirement is that users can add a picture from their own gallery. This is done similarly to the aforementioned camera functionality. A *GalleryManager* is a simple abstraction with only one method, same as the *CameraManager*. However, This time, no temporary file is created in the *rememberGalleryManager* function 3.24. An intent for picking visual media is used, and upon success, the image is copied from the URI given by the intent to the application's internal memory.

```
@Composable
fun rememberGalleryManager(onResult: (Uri) -> Unit): GalleryManager {
    val context = LocalContext.current
    val galleryLauncher = rememberLauncherForActivityResult(
            contract = ActivityResultContracts.PickVisualMedia(),
            onResult = { uri ->
                uri?.let {
                    val savedUri = saveImageToInternalStorage( context, URI,
                    ↪  "picture_${System.currentTimeMillis()}.png" )
                    onResult.invoke(savedUri)
                }
            }
        )

    return remember { GalleryManager(
        onLaunch = {
            galleryLauncher.launch(
                PickVisualMediaRequest( mediaType =
                ↪  ActivityResultContracts.PickVisualMedia.ImageOnly )
             )
        }
    )}
}
```

Code listing 3.24 Gallery manager implementation.

3.6.4 Permissions

Android employs a permission system to access certain device functionalities like a camera, a gallery, or a location used in this prototype. Permissions are declared in the application's manifest file. `<uses-permission android:name="android.permission.CAMERA"/>`. Since Android 6.0, however, there have been two types of permissions. Normal permissions do not require user confirmation and are automatically granted during installation. On the other hand, dangerous permissions[4] require an explicit permission request during runtime. Users can either grant or deny the request. After permission is granted, the functionality can be accessed until the user revokes it in the settings.

This prototype abstracted the runtime permission requests into a *PermissionRequest* 3.25. This has two properties: the granted property, which defines if the permission was granted or not, and the launcher, which launches the request.

```
abstract class PermissionRequest(
    protected open val launcher: ActivityResultLauncher<String>,
    open val granted: State<Boolean>,
) {
    abstract fun requestPermission()
}
```

Code listing 3.25 Permission request implementation.

[4]"*Dangerous permissions are higher-risk permissions that grant requesting applications access to private user data, or control over a device, which can negatively impact the user* [41]."

Specific permissions are then inherited from this class, overriding the abstract method *request-Permission*. For example, this class 3.26 for requesting the camera functionality.

```
class CameraPermissionRequest(
    launcher: ActivityResultLauncher<String>,
    granted: State<Boolean>,
) : PermissionRequest(launcher, granted) {
    override fun requestPermission(): Unit =
        launcher.launch(Manifest.permission.CAMERA)
}
```

■ **Code listing 3.26** Permission request implementation.

3.6.5 Composable Functions

As an example of a composable function, the *EmptyPlaceCard* 3.27 was chosen as it shows how the composables can be nested within each other. The function takes three parameters: a lambda expression invoked when the card is clicked, a Modifier[5] for customizing the card's appearance, and composable content displayed inside the card. It then uses these parameters to create an ElevatedCard, a material design card with a shadow. Furthermore, it shows how the components can be aligned using columns, rows, and grids.

```
@Composable
internal fun EmptyPlaceCard(
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
    content: @Composable () -> Unit,
) {
    ElevatedCard(
        modifier = modifier.height(120.dp).padding(vertical = 8.dp),
        onClick = onClick,
    ) {
        Column(
            modifier = Modifier.fillMaxSize(),
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.Center
        ) {
            content()
        }
    }
}
```

■ **Code listing 3.27** Empty card implementation.

[5] Almost every composable has a modifier argument, allowing for further decorating or augmenting of the said composable.

3.6.6 ViewModel

As mentioned, the ViewModel is an intermediary between the View and the domain layer. Every ViewModel in this prototype holds its state data class, which contains all the data the View needs to display information to users correctly. The ViewModel interacts with use cases in the domain layer to access and manipulate data. User interactions are handled through ViewModel's functions.

As an example, the following listing is a part of the detail screen ViewModel 3.28. The constructor has three use cases for getting, removing, and optimizing a trip. This is the only way the ViewModel communicates with the domain. The functions implemented in the ViewModel are usually run asynchronously using the *launch* function so they do not block the main thread on which the UI runs. These functions then update the state, which triggers a recomposition in the View when the function finishes. The state is held in the *ViewState* data class; in this case, it holds the information if the trip is loading or being optimized, the error message, and the trip itself. Lastly, it contains the *errorFlow*, to which all the errors from use cases are emitted.

```
class DetailViewModel(
    private val getTripById: GetTripUseCase,
    private val deleteTripUseCase: DeleteTripUseCase,
    private val optimiseTripUseCase: OptimiseTripUseCase
): BaseStateViewModel<DetailViewModel.ViewState>(ViewState()) {
    private val _errorFlow = MutableSharedFlow<ErrorResult>(replay = 1)
    val errorFlow: Flow<ErrorResult> get() = _errorFlow

    fun getTrip(tripId: Long) {
        launch {
            update { copy(loading = true) }
            getTripById(tripId).map {
                when (it) {
                    is Result.Success -> { update {
                        copy(trip = it.data, loading = false, optimisingLoading
                        ↪  = false)
                    }}
                    is Result.Error -> { update { copy(
                        error = it.error.message?: "",
                        loading = false,
                        optimisingLoading = false
                    )}}
                }
            }.collect()
        }
    }
    ...
    data class ViewState (
        val trip: Trip? = null,
        val loading: Boolean = false,
        val optimisingLoading: Boolean = false,
    ): State
}
```

Code listing 3.28 Detail screen ViewModel implementation.

3.6.7 Final Design

All screens were implemented based on wireframes from the design chapter 2.7.1 and subsequent testing results. Due to the limitations of the technologies, some features, like showing the approximate time of stay, had to be removed. Additionally, several minor challenges arose during the implementation process. One of the issues was that multiple trips could occur on the same date. This fact was overlooked during the design process. The home screen was designed only to display the closest trip. The problem was solved using a side sheet showing the multiple trips scheduled for this date. The sheet can be expanded by sliding to the right or using the hamburger-style button. A badge displays the number of available trips if there is more than one. The essential screens can be seen in figure 3.1 with the rest of the screens in the appendix B.

3.7 Dependency Injection

The previously mentioned Koin 2.3.4 was used to manage dependencies in this prototype. Koin offers different ways to create dependencies:

- **single** Creates a single instance of the dependency throughout the application's lifetime[6].
- **factory** Provides a new instance of the dependency every time it's requested.
- **viewModel** Specifically designed for Koin to manage ViewModels within the Android lifecycle

The prototype has multiple modules for managing dependencies across the application. Firstly, there is the *commonModule*. This module contains all the shared logic across the platforms: networking, database, sources, repositories, and use cases are all created here.

```
private val commonModule = module {
    ...
    single(named("PlacesClient")) { PlacesClient.init(get(), get(), get()) }
    // Use cases
    factory<SaveTripUseCase> { SaveTripUseCaseImpl(get(), get(), get()) }
    ...
    // Repositories
    single<PlaceRepository> { PlaceRepositoryImpl(get(), get()) }
    ...
    // Sources
    single<PlaceRemoteSource> { PlaceRemoteSourceImpl(get(), get()) }
    ...
    // Http Services
    single { PlaceService(get(named("PlacesClient"))) }
    ...
    // Database
    single { createDatabase(get()) }
    ...
}
```

Code listing 3.29 Shared part of the application dependency injection implementation.

[6] Also known as the singleton pattern.

Then there is the *platfromModule*, which contains the platform-specific drivers, implementation, and the actual classes.

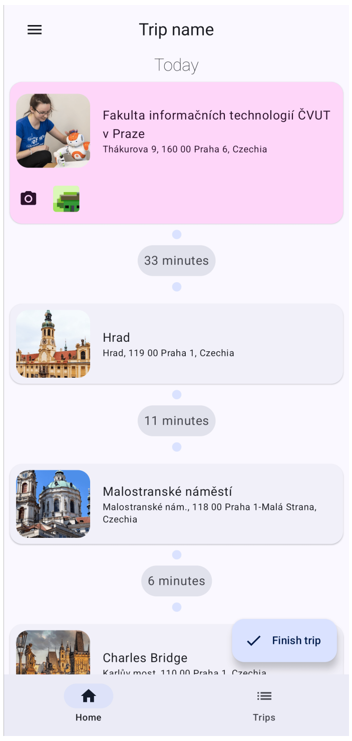
```
actual val platformModule = module {
    ...
    single { DriverFactory(get()) }
    single { Android.create() }
    single<LocationController> {
        LocationController(
            context = get(),
            locationProvider = LocationServices
             ↪  .getFusedLocationProviderClient(get<Context>()),
        )
    }
}
```

■ **Code listing 3.30** Android-specific dependency injection implementation.

Lastly, there is a separate dependency management in the presentation layer on Android. Here, the ViewModels are created and injected into the screen composables. An example of this is the *tripModule*.

```
val tripModule = module {
    viewModel { SearchViewModel(get(), get(), get()) }
    viewModel { EditViewModel(get(), get(), get(), get(), get()) }
    ...
}
```

■ **Code listing 3.31** Android presentation layer dependency injection implementation.


Trip name

Today

Fakulta informačních technologií ČVUT v Praze
Thákurova 9, 160 00 Praha 6, Czechia

33 minutes

Hrad
Hrad, 119 00 Praha 1, Czechia

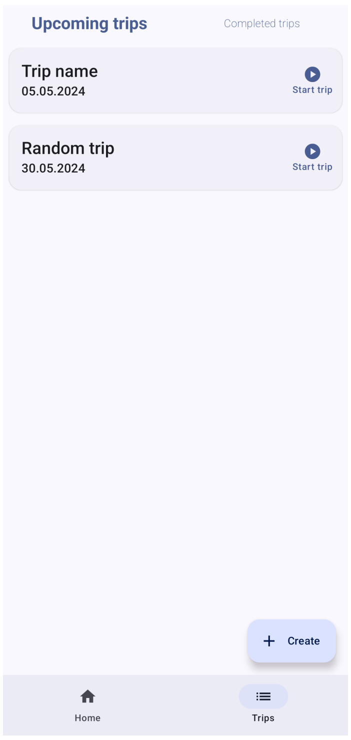
11 minutes

Malostranské náměstí
Malostranské nám., 118 00 Praha 1-Malá Strana, Czechia

6 minutes

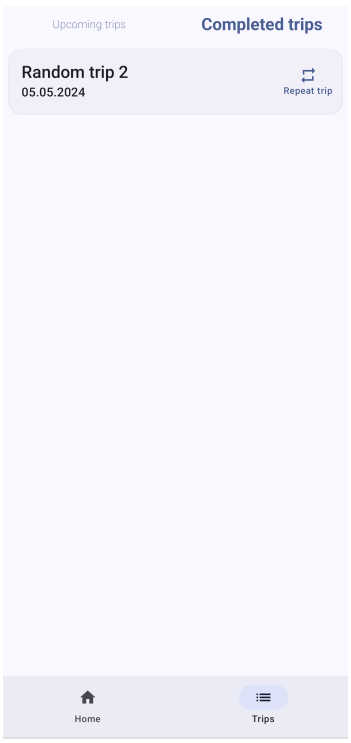
Charles Bridge
Karlův most, 110 00 Praha 1, Czechia

(a) Home screen

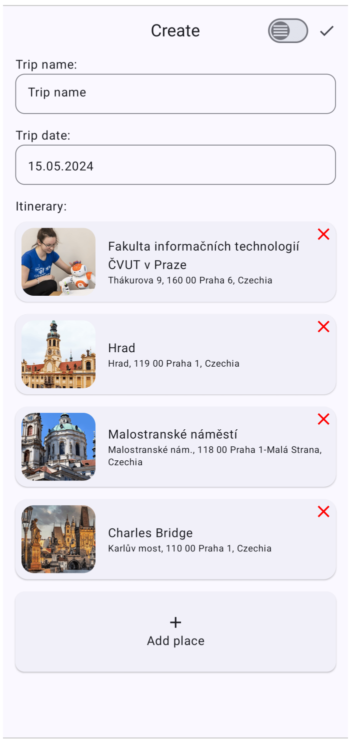

Trip name
05.05.2024

Random trip
30.05.2024

(b) New trips screen


Random trip 2
05.05.2024

(c) Completed trips screen


Create

Trip name: Trip name

Trip date: 15.05.2024

Itinerary:

Fakulta informačních technologií ČVUT v Praze
Thákurova 9, 160 00 Praha 6, Czechia

Hrad
Hrad, 119 00 Praha 1, Czechia

Malostranské náměstí
Malostranské nám., 118 00 Praha 1-Malá Strana, Czechia

Charles Bridge
Karlův most, 110 00 Praha 1, Czechia

(d) Create screen

Figure 3.1 Screens implemented in Jetpack Compose..

Chapter 4

Testing

The testing phase in software engineering is an important step that ensures the quality and functionality of a software program before it's released to users. It involves the evaluation of the software to identify and fix any defects, bugs, or errors that might have been introduced during the development phase. In this chapter, some of the testing methods will be introduced. Actual examples of this application's testing will be shown. Lastly, Google Firebase will be mentioned as a tool aiding this process.

4.1 Testing Methods

There are different ways that an application can be tested. Firstly, the tests could be divided into two categories based on their automatization: manual and automated.

- Manual testing is the more expensive and time-consuming method, requiring an actual person to perform these tests individually. It is also prone to human error because a person can make a typo or omit a step in a script.
- On the other hand, automated tests are usually used in a real-life scenario. They are a key component of continuous integration and continuous delivery. Unlike manual tests, automated tests are performed by a machine based on a test script written beforehand and can be set up to run on each push to a repository. This omits human error; however, finding edge cases can be more difficult this way.

Another way to differentiate tests is by their type. First, there are **unit tests**, which are low-level tests that solely concentrate on the application code. They test individual functions, classes, and small units of the application code and are often used as they are easy to perform and automate.

Integration tests ensure the various modules or services the application uses work properly together. This might involve testing how the user interface interacts with the domain layer or how different functionalities like search communicate with backend servers. However, these tests are more expensive to conduct and more complex to set up because they require numerous application components to be operational.

Functional tests focus on the application's business requirements. They only verify the output without controlling the intermediate system states. As opposed to the integration tests, these not only control whether the user can search for something and the backend responds but also test whether the response is correct.

End-to-end tests replicates user-like behavior. It tests the common expected user flows in the complete application environment. These are very useful but difficult and costly to implement and automate. [42]

Acceptance testing is a method of formal testing to evaluate the system's compliance with the business requirements, whether it is acceptable for release or not. [42]

Performance testing assesses a system's performance under different workloads. It tests the reliability, speed, scalability, and responsiveness; from this, it can define the possible bottlenecks[1] of the application.

The last mentioned here are the **user tests**. These are performed by actual people testing the usability and intuitiveness of the application, providing valuable insights into the UX that other testing methods might miss. While user testing requires more resources compared to automated testing, it is crucial for ensuring a user-friendly application.

Other forms of testing exist, but these are the most used according to [42]. Henceforth, they are mentioned and explained in this chapter. Continuing concrete examples used for testing this application prototype will be shown.

4.2 Unit Tests

For Unit testing of this application, multiple approaches had to be taken. Because it runs on different platforms, distinct testing libraries were used.

4.2.1 Android Unit Tests

In the Android implementation, the *GalleryViewModel* was tested. A well-known framework, Mockito, was used to create isolated tests. Not to test use case implementations, the use cases were mocked in order to create the ViewModel.

The *GalleryViewModel* instance was created within a *@Before* annotated function to streamline test setup. This ensured a new instance for each test, preventing any potential influence from previous tests. This annotation is from the JUnit framework.

Because the methods in the *GalleryViewModel* run their calls to the use cases in a coroutine scope, another library was needed to test the application. This library is *kotlinx-coroutines-test*. This library facilitates testing asynchronous functions in Kotlin. The *runTest* function allowed tests to be launched and managed in a synchronous test environment. In order to wait for the use case calls to complete, the *advanceUntilIdle* was used; it advances the virtual clock of the test dispatcher until all the coroutines launched before being completed.

A unit test usually comprises three parts:

1. **Setup** Here, everything needed for the test to be executed is created, and behavior is mocked using the *whenever* function.
2. **Execution** It is the actual execution where the functionality tested is run.
3. **Verification** The assertions and verifications are made, and function calls are verified to confirm that the function or class behaved correctly under the test.

A part of the test mentioned above can be seen in the listing 4.1.

[1] A point in a system that restricts its overall performance.

```kotlin
class GalleryViewModelTest {
    @OptIn(ExperimentalCoroutinesApi::class)
    private val testDispatcher = UnconfinedTestDispatcher()

    private lateinit var getTripUseCase: GetTripUseCase
    private lateinit var deleteTripUseCase: DeleteTripUseCase
    private lateinit var getPhotosByTripUseCase: GetPhotosByTripUseCase
    private lateinit var savePhotoUseCase: SavePhotoUseCase
    private lateinit var galleryViewModel: GalleryViewModel

    @Before
    fun setup() {
        getTripUseCase = mock()
        deleteTripUseCase = mock()
        getPhotosByTripUseCase = mock()
        savePhotoUseCase = mock()
        removePhotoByUriUseCase = mock()

        galleryViewModel = GalleryViewModel( getTripUseCase, deleteTripUseCase,
        ↪   getPhotosByTripUseCase, savePhotoUseCase, removePhotoByUriUseCase,
        ↪   testDispatcher)
    }

    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun `get all correctly sets the trip and photos`() = runTest {
        //Setup
        val trip = Trip(id = 1L, name = "Test Trip", itinerary = listOf(), ...)
        val photo = Photo(placeId = "1", photoUri = "uri", tripId = trip.id)

        whenever(getTripUseCase.invoke(GetTripUseCase.Params(trip.id)))
        ↪   .thenReturn(flowOf(Result.Success(trip)))
        whenever(getPhotosByTripUseCase.invoke(
        ↪   GetPhotosByTripUseCase.Params(trip.id)) )
        ↪   .thenReturn(flowOf(listOf(photo)))
        // Execute
        galleryViewModel.getAll(trip.id)
        advanceUntilIdle()
        // Verify
        verify(getTripUseCase).invoke(GetTripUseCase.Params(trip.id))
        verify(getPhotosByTripUseCase)
        ↪   .invoke(GetPhotosByTripUseCase.Params(trip.id))
        assertEquals(trip, galleryViewModel.lastState().trip)
        assertEquals(listOf(photo), galleryViewModel.lastState().photos)
    }
    ...
}
```

■ **Code listing 4.1** Gallery ViewModel implementation.

4.2.2 Common Code Unit Tests

In the common implementation, however, the code runs on different platforms. Therefore, with the libraries Mockito and JUnit designed to run on JVM and Android, a different solution was needed to test the multiplatform solution's business logic.

The best candidates for unit tests in the shared code are the use cases, as their implementation rarely changes, even when the rest of the system does. As an example, the *SaveDistancesUseCase* was chosen 4.2.

In this case, to separate the tests from the rest of the implementation, the MocKMP library came up as the best choice. This library uses annotations to choose which classes will be mocked in the test class. Next, a mocker must be used in the constructor of each mocked class so its behavior is later modifiable.

To test suspending functions utilized by the *SaveDistancesUseCase*, *runBlocking* was used as it was designed to run asynchronous context synchronously. Moreover, the MocKMP library also offers functions to mock suspending functions and verify calls to these functions. This functionality is used by adding the prefix -*Suspending* after calls on the mocker.

The actual tests were then realized using a Kotlin multiplatform testing library, *kotlinx-test*, whose structure is very similar to the JUnit for a familiar testing experience and is developed directly by JetBrains.

The three testing steps were performed:

1. **Setup** The behavior of mocked dependencies is defined using the *mocker.everySuspending* function.
2. **Execution** The use case is invoked.
3. **Verification** Assertions are made; however, they are insufficient to determine if the use case works as intended. For this reason, all the calls that are supposed to happen from the tested class are verified using the *mocker.verifySuspending*.

4.3 User Tests

The user test was already executed in the design phase of the application 2.7.2. Nevertheless, to test if the design changes positively impacted the application, the same scenario was intended to be used. However, in this phase, it had to be slightly modified to accommodate the changes made to the UI. The changed scenario reads as follows:

1. Navigate to the "Create Trip" screen.
2. Enter the following trip details:
   - name
   - date
3. Add three places to the trip itinerary:
   - "Prague Castle" as the first location.
   - "Prague Main Station" as the second location.
   - "Charles Bridge" as the third location.
4. Create the trip.

5. In the dialog, opt out of the automatic ordering.
6. Navigate to the "Trip Details" screen.
7. Manually select smart order for this trip.
8. Start the trip early.
9. Upon arrival to the first place, add a picture.
10. Finish the trip.
11. Navigate to the details of the now-finished trip.
12. Add a picture to "Prague Main Station".

This time, six new people were chosen to test the application prototype so the results are not biased. Again, three of the chosen people were from a computer science background, with two familiar with mobile development. The other three are the author's friends from different backgrounds unrelated to computer science. The tests were performed on the author's phone, the model being *Samsung S21 Ultra*, and people's reactions were recorded as feedback.

4.3.1 Results

Firstly, the reactions to the changes made after the first testing are listed:

- Unlike the first testing, all the test subjects navigated to different screens without a problem
- Thanks to the primary action being moved to the item card, no one had a problem figuring out how to get into the details.
- With the optimize button moved to the detail screen as a floating button with text, it took less time for everyone to have the trip automatically sorted.
- Reversed order during the creation process made it more intuitive for everyone to add new places.
- Thanks to the modified order feature being changed from a button to a switch, people identified the feature more quickly and did not mistake it for the optimize feature.

Furthermore, new observations that previously were not mentioned arose from the testing:

- The search feature on the bottom sheet collapses after a place is searched, making adding new places more difficult.
- When entering a name into the text box, the focus is not lost after clicking elsewhere. The user has to directly click on the done button on the keyboard to stop it from being focused.
- Some places from the search are in different languages, the locally used language and English. This creates an inconsistency, which worsens the UX.

These results show that the improvements made to the UI were successful in making the UX better. After more rigorous testing with even more people, new changes could be found to make it even better. For this project, these results are satisfactory, as they point out only bugs in the application, which are not flaws in the UI but rather mistakes made in the code to be addressed. For now, however, there is a lack of time to address these problems, and as they are not obfuscating any features, their solution was postponed.

```
@UsesMocks(PlaceRepository::class, DistanceRepository::class)
class SaveDistancesUseCaseTest {

    private val mocker = Mocker()
    private val mockPlaceRepository: PlaceRepository =
        MockPlaceRepository(mocker)
    private val mockDistanceRepository: DistanceRepository =
        MockDistanceRepository(mocker)
    private val saveDistancesUseCase: SaveDistancesUseCase =
        SaveDistancesUseCaseImpl(mockPlaceRepository, mockDistanceRepository)

    private val trip = Trip(id = 1L, name = "Test Trip", itinerary = listOf(),
    ↪   ...)
    private val distance = Distance(10, 10)
    private val results = listOf(Triple("origin", "destination", distance),
    ↪   Triple("origin2", "destination2", distance))

    @Test
    fun `invoke returns Success when distances are successfully retrieved and
    ↪   saved`() = runBlocking {
        // Setup
        mocker.everySuspending {
        ↪   mockPlaceRepository.getDistanceMatrix(trip.order) } returns
        ↪   Result.Success(results)
        mocker.everySuspending { mockDistanceRepository.saveDistance("origin",
        ↪   "destination", distance, trip.id) } returns Result.Success(Unit)
        mocker.everySuspending { mockDistanceRepository.saveDistance("origin2",
        ↪   "destination2", distance, trip.id) } returns Result.Success(Unit)
        // Execute
        val result = saveDistancesUseCase.invoke(trip)
        // Verify
        mocker.verifyWithSuspend {
            mockPlaceRepository.getDistanceMatrix(trip.order)
            results.forEach { (origin, destination, distance) ->
                mockDistanceRepository.saveDistance(origin, destination,
                ↪   distance, trip.id)
            }
        }
        assertEquals(Result.Success(Unit), result)
    }
    ...
}
```

Code listing 4.2 Distance saving use case test implementation.

4.4 Google Firebase

Besides the testing methods used so far, developers often use what could be called "real-world" testing because it can give the most significant insight into the application. This concentrates on having real people use the application and share their thoughts. In big companies, people are usually hired to perform the testing. However, it can also rely on volunteers. There are different ways this testing can be facilitated. The Firebase platform was used for this application.

Firebase is a toolbox for building, improving, and growing Android, iOS, or web applications. The tools it provides cover a significant portion of the services that developers typically have to build themselves but do not want to since they would instead focus on the app experience. This covers analytics, authentication, databases, file storage, push messaging, and many more features. The services are hosted in the cloud and may be scaled with little to no development work. [43]

To integrate Google Firebase into a project, there are a few crucial steps necessary:

1. A Firebase account has to be created by signing in with a Google account.
2. A new Firebase project needs to be created through a setup offered on the website.
3. Services wanted in the application have to be enabled in the console or can be added later.
4. Depending on the platform, the Firebase SDK libraries need to be installed using platform-specific instructions Firebase provides.
5. The Firebase console then provides a unique configuration file to be put into the application directory.

4.4.1 App Distribution

The first feature used to facilitate the other functions of the Firebase platform is the App Distribution. Here, the application's APK, AAB for Android or IPA for iOS files, used for installing the application, can be put and distributed to users [44]. The generated file can be dragged and dropped onto the page. Once it is uploaded, the testers can be assigned to receive access. Different groups of different people can be created for various purposes, allowing for the separation of demographics or internal testers.

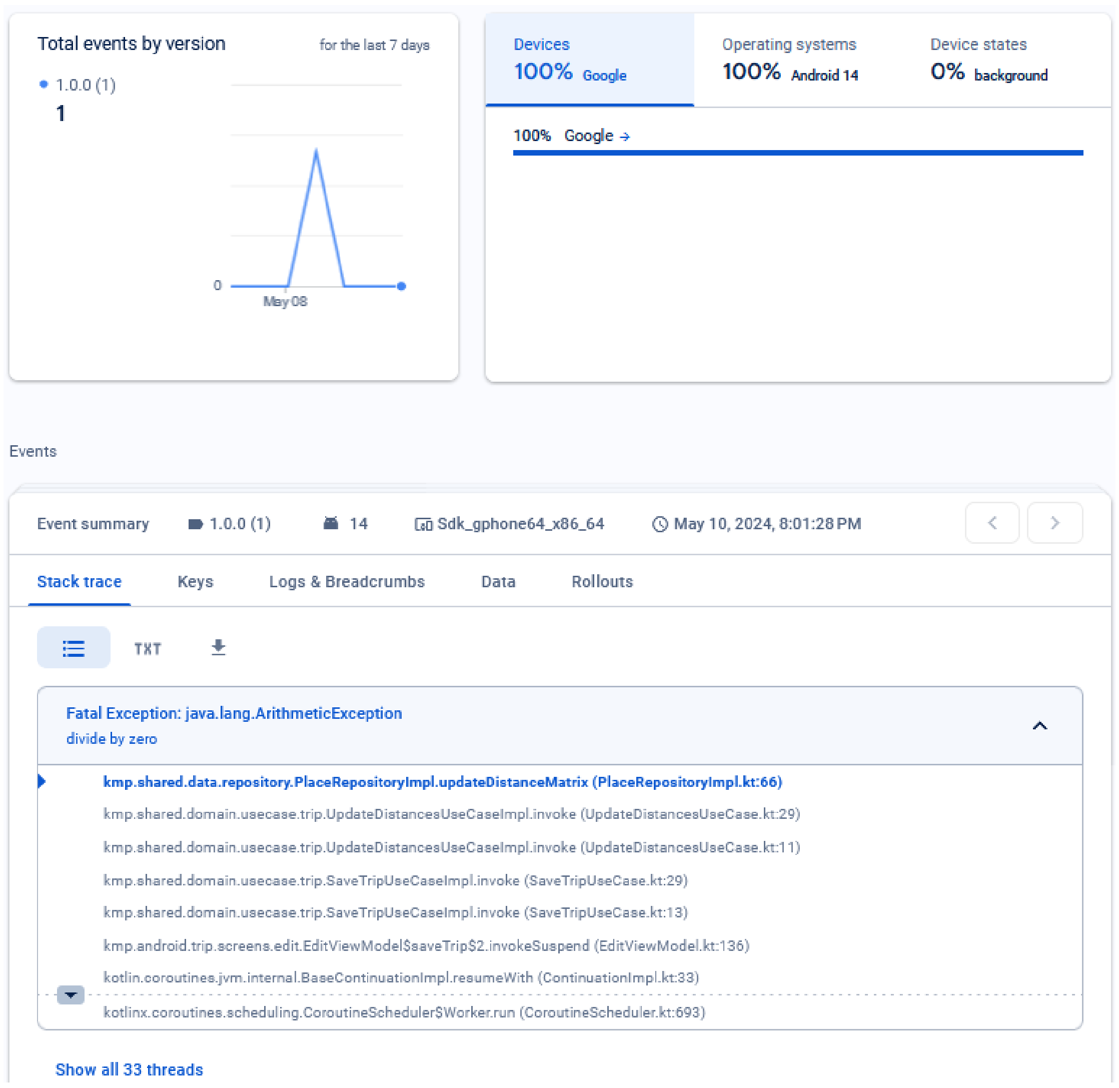
Another benefit of the app distribution is that it maintains a history of uploaded builds, allowing change tracking or reverting to previous versions if necessary. Moreover, after finding a critical issue in a build, it is easy to do a rollback and restrict access to that version.

Moreover, a new feature enables automated testing upon uploading a new version. The developer can select which and how many devices the application will be installed on. Then, Firebase uses bots that click on interactive components of the application's UI. This does not serve as a test and cannot replace human testing. However, it serves as a great tool for finding obvious crashes before the application is made accessible to the testing group.

4.4.2 Crashlytics

In the realm of mobile app development, stability is one of the most critical aspects of ensuring user satisfaction. The Crashlytics, another part of the Firebase platform, is an invaluable tool in this regard. As a comprehensive crash report system, it automatically captures and reports if the application ever crashes. These reports provide the error messages, specific device information, and the app's state before the crash occurred. They are sent to the console from a few seconds to thirty minutes after the crash so the development team can start working on the fix immediately, knowing the exact circumstances in which the error happened.

The image 4.1 shows an example of such a report. Thanks to the stack trace from the report, the problem was hastily found. Here, an exception occurred while trying to calculate distances between places. This happened because places were both removed and added to a trip and when calculating the number of new places, which is later used to calculate API calls, zero was the result.



```
Fatal Exception: java.lang.ArithmeticException
divide by zero
kmp.shared.data.repository.PlaceRepositoryImpl.updateDistanceMatrix (PlaceRepositoryImpl.kt:66)
kmp.shared.domain.usecase.trip.UpdateDistancesUseCaseImpl.invoke (UpdateDistancesUseCase.kt:29)
kmp.shared.domain.usecase.trip.UpdateDistancesUseCaseImpl.invoke (UpdateDistancesUseCase.kt:11)
kmp.shared.domain.usecase.trip.SaveTripUseCaseImpl.invoke (SaveTripUseCase.kt:29)
kmp.shared.domain.usecase.trip.SaveTripUseCaseImpl.invoke (SaveTripUseCase.kt:13)
kmp.android.trip.screens.edit.EditViewModel$saveTrip$2.invokeSuspend (EditViewModel.kt:136)
kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith (ContinuationImpl.kt:33)
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run (CoroutineScheduler.kt:693)
```

Figure 4.1 Screenshot showing one of the reports from Crashlytics.

4.4.3 Analytics

Within the Firebase suite of services, another used in this project was Analytics. This is the cornerstone of understanding the users' behavior within the mobile application, as it was designed for developers to help better understand and optimize the application. It integrates with the application seamlessly without needing any additional code. Even without extra code or settings, the analytics captures a lot of data.

Moreover, besides tracking behavior, Analytics offers in-depth data on user demographics, such as device model, operating system version, language preference, and geographical location. The results can be filtered or compared based on these criteria. This allows for an even better understanding of the user base.

Unfortunately, due to its late implementation in this project, the collected user behavior data only spans one week. This limited timeframe somewhat restricts the in-depth understanding of the application trends, as that kind of data only shows after a broader time horizon. However, it does provide a glimpse into real-world-like usage patterns. For example, the following graph shows how many times the app was opened throughout the week 4.2.

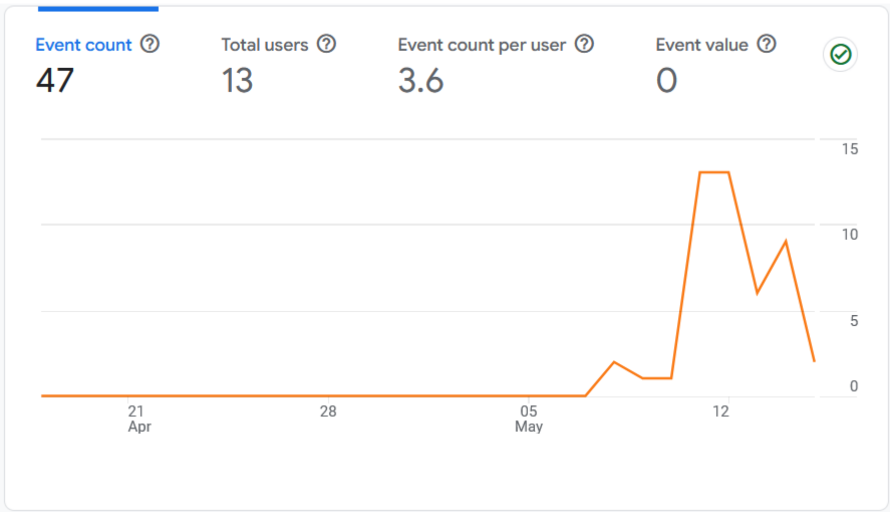
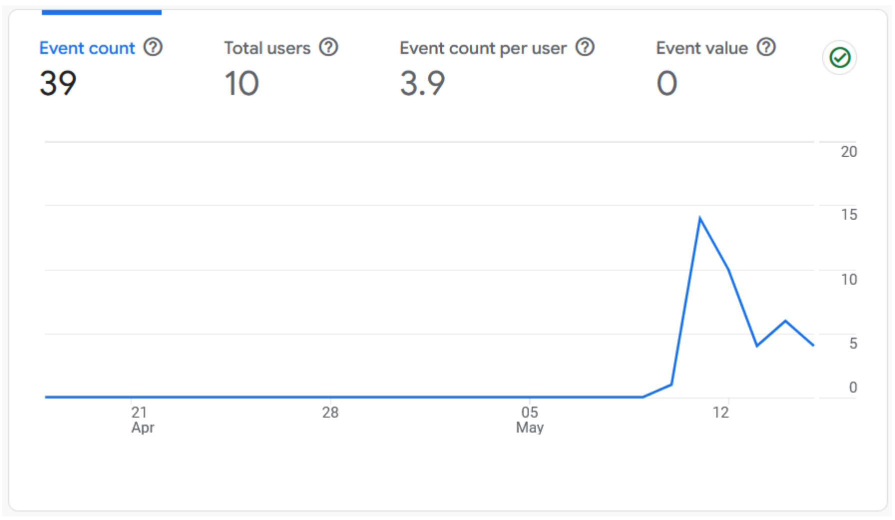


Figure 4.2 Number of times the app was opened by day.

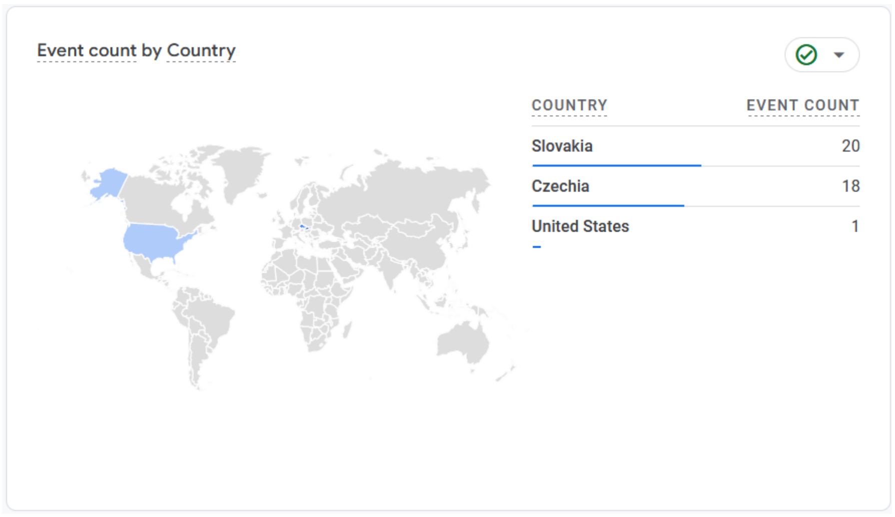
Furthermore, what is even more helpful is that the developers can implement their custom events. It is done simply by injecting *FirebaseAnalytics* class into anything from where the data will be collected. After injection, it provides a simple function for logging data that contains the event's name and optional parameters or bundles for additional data gathering. An example of such a use can be seen in the listing 4.3. This data is automatically added to the Analytics console, and an intuitive visualization is offered. In the following figures, the number of created events can be divided by the time of when they happened4.3, or by the different countries where they happened 4.4.

```
analytics.logEvent("trip_created"){
    param("trip_name", trip.name)
    param("trip_itinerary", trip.itinerary.joinToString { it.name })
}
```

Code listing 4.3 Logging a custom event to analytics.



■ **Figure 4.3** Number of times trip was created by day.



COUNTRY	EVENT COUNT
Slovakia	20
Czechia	18
United States	1

■ **Figure 4.4** Number of times trip was created by country.

Chapter 5

Discussion

This chapter stems from the current state and focuses on the application's future. Can the application be released, and what problems lie ahead in future development? What features would aid in making the user experience better?

To answer the first question: No, the application is currently in the prototype stage of its life cycle. Several features are critical for a public release.

First and foremost, the Google Maps Platform the application is built on is a paid platform. This means that, as it is right now, the developer would have to pay for the expenses generated during the use. Multiple solutions are available to deal with such a problem: The application could have a subscription model, where the user pays a monthly fee to use the application. Moreover, a free tier could utilize Google Ads to pay for Google's APIs. A thorough analysis is required to determine the best way to monetize and make the application profitable.

Another essential feature for release is user authentication and data backup capabilities. This is required to allow a subscription model. Almost every significant application nowadays has some form of managing its users. Integration with multiple platforms would be ideal, supporting logging in through platforms like Google or Apple ID. This would make the application more usable as the data could be stored in a cloud solution like the one Firebase offers; this way, users would not lose their data when changing devices or reinstalling their system.

Furthermore, more rigorous testing is needed, as it currently contains multiple bugs and problems worsening the UX. More testing, however, would require more time, which is presently not feasible. The app should have a higher unit test coverage in the future, more tests from the categories mentioned in the previous chapter 4.1 should be implemented, and more testers should be included in the private release to guarantee a stable application experience.

Early during the implementation phase, it was realized that one person developing a cross-platform solution for this problem is insufficient. The application has a potential far beyond one developer's abilities. This application merely serves as a proof of concept. The lack of development power results in a lack of features, as the application is currently only bare-bones[1].

[1]Only what is most basic or necessary.

5.1 Future of the application

The answer to the second question is somewhat subjective as it is mainly based on the author's own opinions and results of testing done, mostly the testing done through Firebase. However, these suggestions are still viable and worth considering when continuing with the development. Here, the features deemed to improve the UX most are listed:

- **iOS Application:** Thanks to the sound foundations of the application, with the business logic being coded multiplatform-ready, creating a presentation layer for an iOS application is a simple task. The only obstacle to achieving this is a lack of the correct tools. If implemented in the future, it would bring even more users to forming an active community.
- **Modes of Transport:** The application currently only supports walking everywhere on foot. It may be feasible in small cities with limited-sized city centers; however, when the user wants to visit a bigger city, it is necessary to include services like public transit or taxi, supported by the Google Routes API used in this implementation. Moreover, support for cars with an automatic addition of parking locations to the itinerary would make it easy for the people who travel to the city by car.
- **Implementing a Back-end Server:** Introducing a server as a mediator between the mobile client and the Google Maps Platform offers exciting possibilities. This server could handle the complex logic behind trip creation and optimization. Currently, the focus is on efficient mobile device performance. A server, however, could house a more powerful optimization algorithm. Moreover, if there was a back-end server, social features could be implemented. This would increase people's engagement with the application, allowing trip sharing, photo sharing, travel tips, collaborative itinerary building, and much more.
- **Itinerary Customization:** Users would also benefit if there were an option to split an itinerary into multiple days. For now, the user needs to create multiple itineraries for one trip. It would also be helpful if the user could add notes to certain places, like check-in or reservation times. Additionally, if the user decides to optimize a trip, it switches all the places to create a more efficient route. However, it would be helpful if the user could lock places at their point in the trip and have the rest optimized.
- **Displaying More Information:** Currently, the application only shows the place's name, picture, and address. In the future, this could be expanded to show information like opening hours user ratings or an icon according to what type of a place it is. All this information is accessible from the Google APIs; however, it would require changing the domain model to store more information in the *Place* entity.
- **Itinerary Map:** Adding a map to an itinerary would help make more sense of the order given by the application. On the map, users could see an overview of their trip and would be able to make adjustments according to their preferences.
- **Notifications:** When traveling through a city, users can quickly lose track of their progress. If the user would get a notification after arriving at a place of interest, it would help to clarify their progress. Moreover, this could be an invitation to take a picture of the location; this way, more users would take pictures, improving the process of recollecting a past trip.
- **More Exciting UI:** At the current stage of the application, the design is very analytical, focusing on the efficiency of the travel. Changes could be made to help motivate the user more. For example, praise or encouragement after optimizing or finishing a trip would inspire the user to repeat this process in the future. Furthermore, adding more playful animations would make the UX friendlier.

Conclusion

This thesis aimed to analyze, design, and implement an Android application to help travelers get the most out of their travels.

In the beginning, the potential of the application was found by identifying the possible user base. It was found to be a significant amount of people, with the number still rising. After, the existing solutions from Google Play were analyzed. Their strengths and weaknesses were identified and translated into a set of requirements and use cases for the future application.

Subsequently, the thesis focused on what the application should be and on what technologies it should be built. It was determined that KMM is the appropriate strategy for such an application since it balances effort and outcome with multiple platforms in mind. Clean Architecture was chosen to ensure the best future-proofing and scalability. After establishing the application's base, the technologies required to implement the solution were evaluated and selected. The final stage of this phase involved designing and testing the UI. Six people tested the UI, and based on their feedback, final revisions were made before implementation.

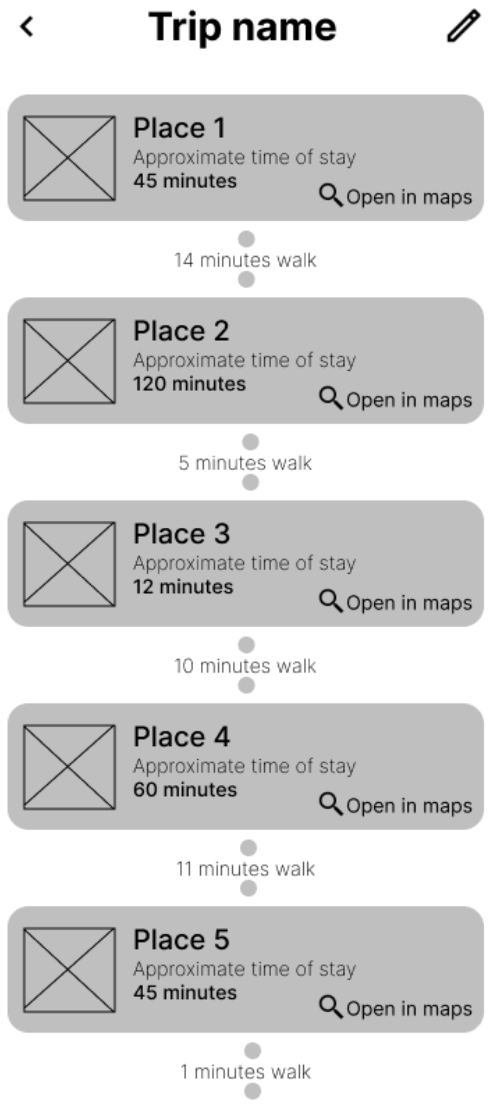
The application's development began with a dev-stack, which served as a model for a properly structured project. The application was only created for Android, but with KMM, adding an iOS presentation layer in the future will be simple. The use of modern technologies came to fruition as they enabled smooth integration and fast-paced development. All functional requirements were successfully implemented.

Following implementation, the application was tested. The testing phase would require more time than was available; the unit test coverage is small, and more testing scenarios would be necessary to evaluate the application properly. However, because it is only a prototype that is unsuitable for public release, further testing has not yet been considered required. Google Firebase was found to be an invaluable tool for distributing, analyzing, and guarding the application in the early phases of its development.

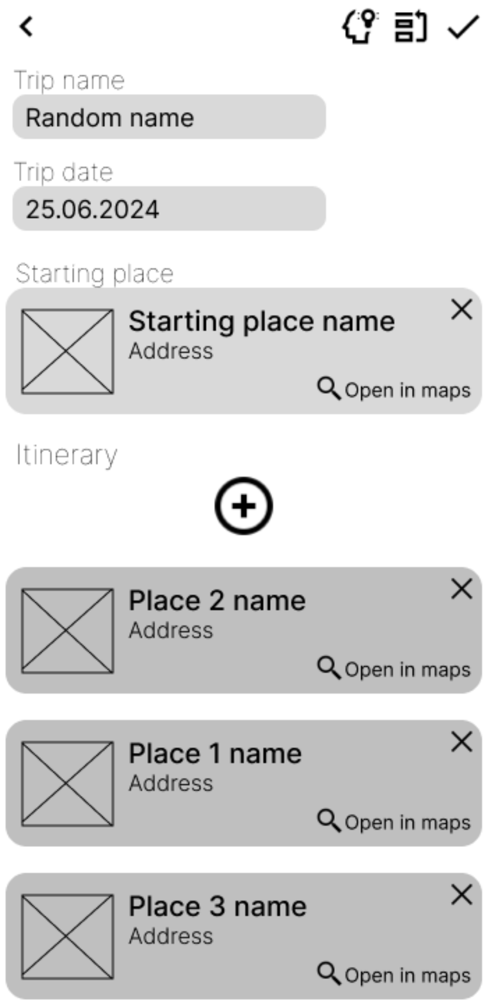
In the final chapter, the outcome of this thesis was discussed. This highlighted issues previously overlooked or out of this project's scope. Nonetheless, thanks to the feedback from Firebase-enabled testing, concrete examples of possible improvements were found to aid this application's future development.

All in all, this thesis accomplished the set goals. The outcome is a functioning application implemented on the Android operating system. The application was met with a positive attitude from the testers, and with their help, it can one day become a successfully deployed application with a thriving community.

Appendix A

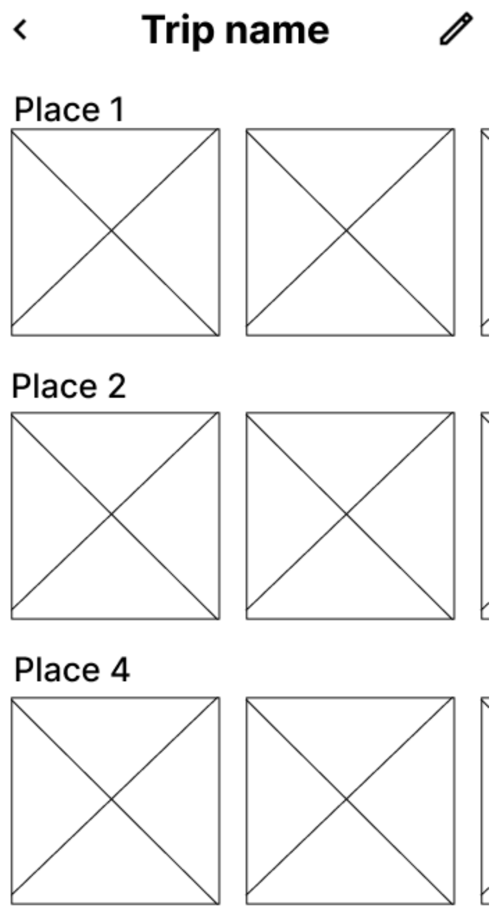


(a) Trip detail screen

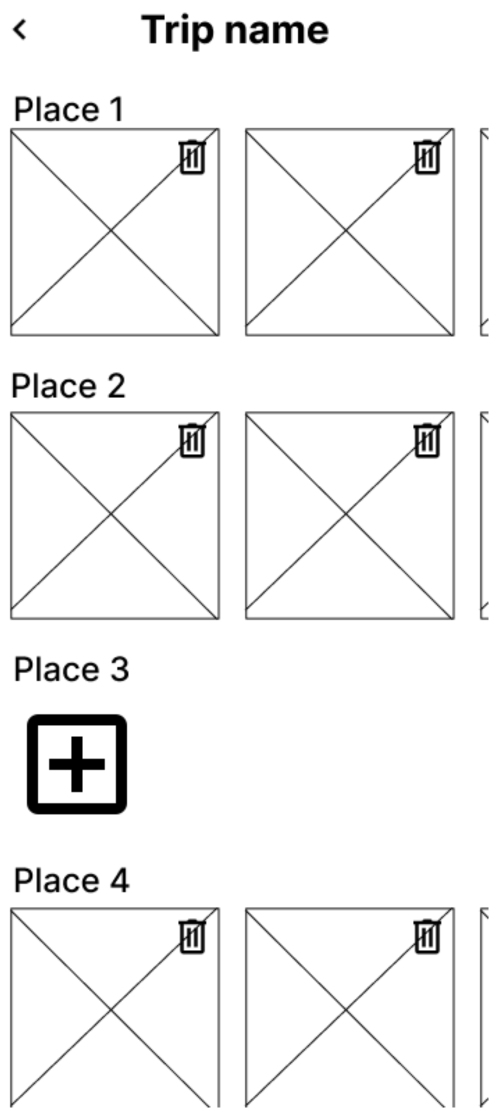


(b) Trip edit screen

Figure A.1 Wireframes designed in Figma.



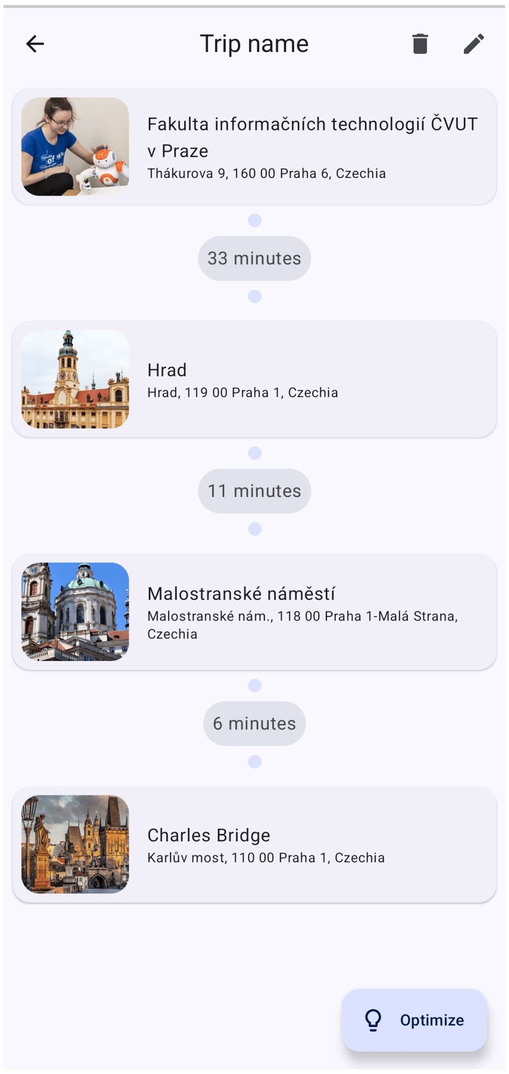
(a) Completed trip detail screen



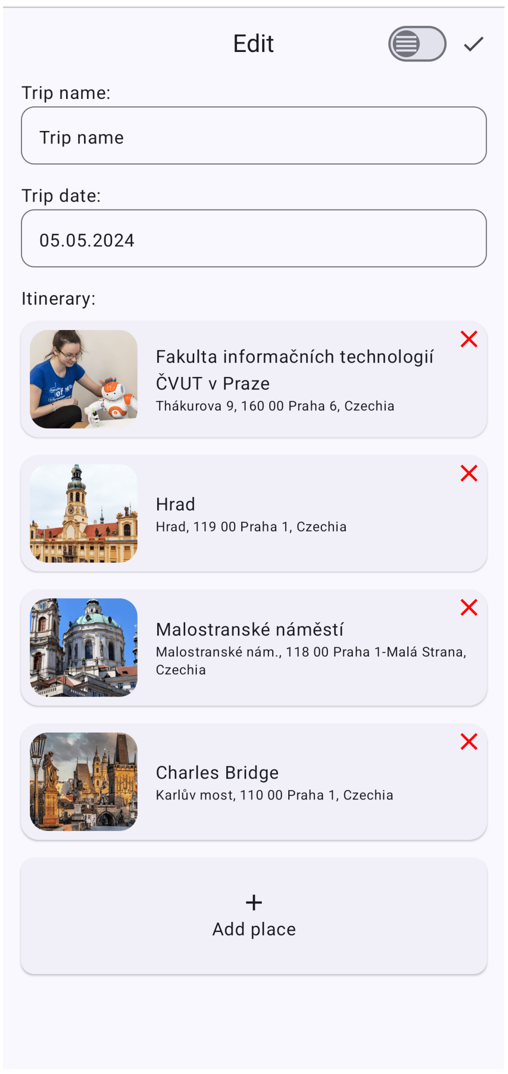
(b) Completed trip edit screen

Figure A.2 Wireframes designed in Figma.

Appendix B

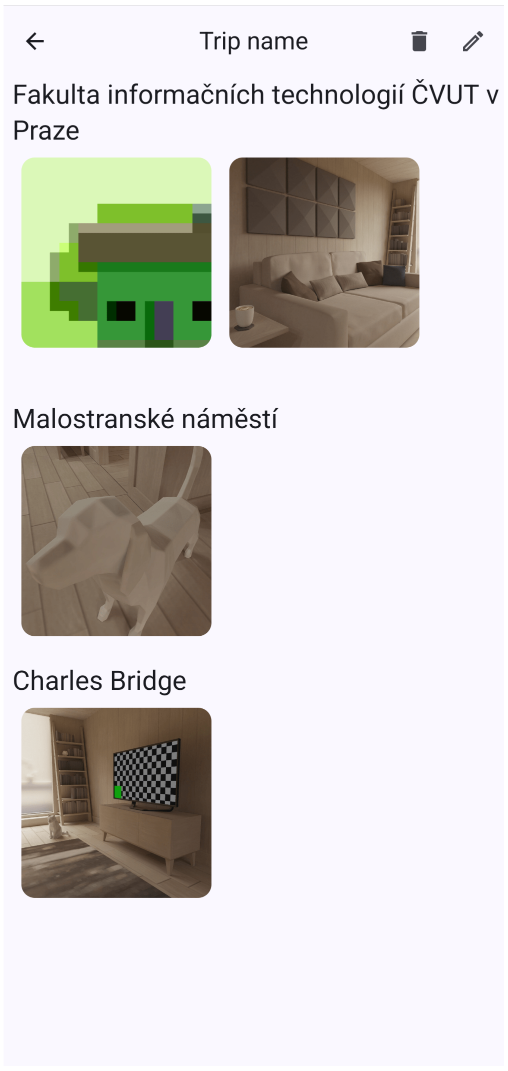

Trip name
Fakulta informačních technologií ČVUT v Praze
Thákurova 9, 160 00 Praha 6, Czechia
33 minutes
Hrad
Hrad, 119 00 Praha 1, Czechia
11 minutes
Malostranské náměstí
Malostranské nám., 118 00 Praha 1-Malá Strana, Czechia
6 minutes
Charles Bridge
Karlův most, 110 00 Praha 1, Czechia
Optimize

(a) Trip detail screen

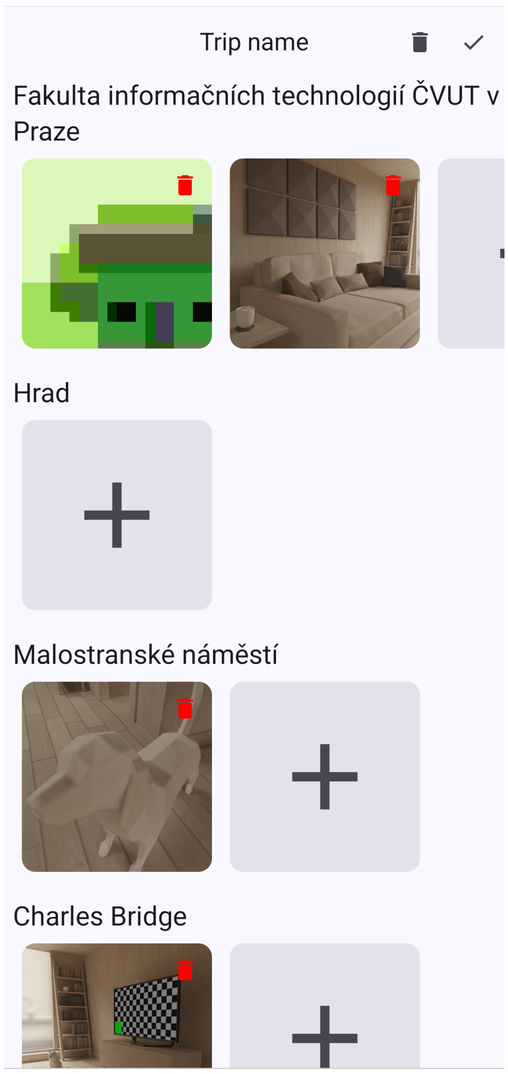

Edit
Trip name:
Trip name
Trip date:
05.05.2024
Itinerary:
Fakulta informačních technologií ČVUT v Praze
Thákurova 9, 160 00 Praha 6, Czechia
Hrad
Hrad, 119 00 Praha 1, Czechia
Malostranské náměstí
Malostranské nám., 118 00 Praha 1-Malá Strana, Czechia
Charles Bridge
Karlův most, 110 00 Praha 1, Czechia
+
Add place

(b) Trip edit screen

Figure B.1 Screens implemented in Jetpack Compose.



(a) Completed trip detail screen



(b) Completed trip edit screen

Figure B.2 Screens implemented in Jetpack Compose.

Bibliography

1. HERRE, Bastian; SAMBORSKA, Veronika; ROSER, Max. Tourism. *Our World in Data* [online]. 2023. Available also from: `https://ourworldindata.org/tourism`. [Accessed 1-04-2024].
2. STATISTA. *Global: number of smartphone users 2014-2029 | Statista — statista.com* [online]. 2024. Available also from: `https://www.statista.com/forecasts/1143723/smartphone-users-in-the-world`. [Accessed 1-04-2024].
3. WANDERLOG. *Wanderlog - Trip Planner App - Apps on Google Play — play.google.com* [online]. 2024. Available also from: `https://play.google.com/store/apps/details?id=com.wanderlog.android`. [Accessed 1-04-2024].
4. TRIPIT, INC. *TripIt: Travel Planner - Apps on Google Play — play.google.com* [online]. 2024. Available also from: `https://play.google.com/store/apps/details?id=com.tripit`. [Accessed 1-04-2024].
5. TRIPADVISOR. *Tripadvisor: Plan & Book Trips - Apps on Google Play — play.google.com* [online]. 2024. Available also from: `https://play.google.com/store/apps/details?id=com.tripadvisor.tripadvisor`. [Accessed 1-04-2024].
6. VOYAGE AI INC. *iplan.ai - Travel Planner - Apps on Google Play — play.google.com* [online]. 2024. Available also from: `https://play.google.com/store/apps/details?id=ai.iplan.app`. [Accessed 1-04-2024].
7. POLARSTEPS. *Polarsteps - Travel Tracker - Apps on Google Play — play.google.com* [online]. 2024. Available also from: `https://play.google.com/store/apps/details?id=com.polarsteps`. [Accessed 1-04-2024].
8. STATCOUNTER. *Mobile OS market share worldwide 2009-2023 | Statista — statista.com* [online]. 2024. Available also from: `https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/`. [Accessed 17-04-2024].
9. STATCOUNTER. *Mobile OS share North America 2018-2024 | Statista — statista.com* [online]. 2024. Available also from: `https://www.statista.com/statistics/1045192/share-of-mobile-operating-systems-in-north-america-by-month/`. [Accessed 17-04-2024].

10. SURI, Bhawna; TANEJA, Shweta; BHANOT, Isha; SHARMA, Himanshi; RAJ, Aanchal. Cross-Platform Empirical Analysis of Mobile Application Development frameworks: Kotlin, React Native and Flutter. In: *Proceedings of the 4th International Conference on Information Management & Machine Intelligence*. Jaipur, India: Association for Computing Machinery, 2023. ICIMMI '22. ISBN 9781450399937. Available from DOI: `10.1145/3590837.3590897`.
11. DLABAL, Matouš. *Vývoj mobilních aplikací pomocí technologie Kotlin Multiplatform*. 2023. PhD thesis.
12. JETBRAINS. *Compose Multiplatform UI Framework | JetBrains — jetbrains.com* [online]. 2024. Available also from: `https://www.jetbrains.com/lp/compose-multiplatform/`. [Accessed 19-04-2024].
13. JETBRAINS. *Kotlin Multiplatform Mobile Goes Alpha | The Kotlin Blog* [online]. 2024. Available also from: `https://blog.jetbrains.com/kotlin/2020/08/kotlin-multiplatform-mobile-goes-alpha`. [Accessed 19-04-2024].
14. GARCÍA, Raúl Ferrer. *IOS architecture patterns: MVC, MVP, MVVM, Viper, and VIP in Swift*. Apress L. P., 2023.
15. SHAJI, Pooja. Choosing Android Architectures: MVC, MVP, MVVM, Clean Architecture, and MVI. *Medium* [online]. 2023. Available also from: `https://medium.com/@KodeFlap/choosing-android-architectures-mvc-mvp-mvvm-clean-architecture-and-mvi-8ad2a43f7f9b`.
16. MARTIN, Robert C. *Clean architecture: A craftsman's guide to software structure and Design*. Prentice Hall, 2018.
17. LÄMMEL, Ralf; PEYTON JONES, Simon. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In: 2003, vol. 38, pp. 26–37. Available from DOI: `10.1145/604174.604179`.
18. *Data Binding Library* [online]. 2024. Available also from: `https://developer.android.com/topic/libraries/data-binding`. [Accessed 19-04-2024].
19. TEKIN, Semih. What is the Clean Architecture? - Semih Tekin - Medium. *Medium* [online]. 2023. Available also from: `https://semihtekin.medium.com/what-is-the-clean-architecture-c80c2a2ff69a`.
20. KHUDAIR, Mohammed. MVI Architecture Pattern in Android | Medium. *Medium* [online]. 2024. Available also from: `https://medium.com/@mohammedkhudair57/mvi-architecture-pattern-in-android-0046bf9b8a2e`.
21. GOOGLE. *Why Compose | Jetpack Compose | Android Developers — developer.android.com* [online]. 2024. Available also from: `https://developer.android.com/develop/ui/compose/why-adopt`. [Accessed 22-04-2024].
22. JETBRAINS. *Ktor: Build Asynchronous Servers and Clients in Kotlin — ktor.io* [online]. 2024. Available also from: `https://ktor.io/`. [Accessed 23-04-2024].
23. SQUARE, Inc. *Overview - SQLDelight — cashapp.github.io* [online]. 2024. Available also from: `https://cashapp.github.io/sqldelight/2.1.0-SNAPSHOT/`. [Accessed 23-04-2024].
24. KANAKE, Ezra. *A Guide to SQLDelight | Baeldung on Kotlin* [online]. 2024. Available also from: `https://www.baeldung.com/kotlin/sqldelight`. [Accessed 23-04-2024].
25. REDA, Karim. Dependency Injection using Koin in Kotlin Multiplatform Mobile (KMM). *Medium* [online]. 2023. Available also from: `https://medium.com/arconsis/dependency-injection-using-koin-in-kotlin-multiplatform-mobile-kmm-eb4cfe82ed6`.

26. MUNUSAMY, Boobalan. Accessing User's Location Guide Android 2023 - Boobalan Munusamy - Medium. *Medium* [online]. 2023. Available also from: `https://medium.com/@boobalaninfo/accessing-users-location-guide-android-2023-60a6f018a718`.

27. KAUSHIK, Vijay Kant. An In-depth Comparison of Glide and Coil for Efficient Image Loading in Android. *Medium* [online]. 2023. Available also from: `https://medium.com/@vijaykantkaushik/an-in-depth-comparison-of-glide-and-coil-for-efficient-image-loading-in-android-c9298016c4b0`.

28. GOOGLE. *Google Maps Platform: Location and Mapping Solutions* [online]. 2024. Available also from: `https://mapsplatform.google.com/`. [Accessed 24-04-2024].

29. MAPBOX. *Mapbox Docs — docs.mapbox.com* [online]. 2024. Available also from: `https://docs.mapbox.com/`. [Accessed 24-04-2024].

30. NOMINATIM. *Overview - Nominatim 4.4.0 Manual — nominatim.org* [online]. 2024. Available also from: `https://nominatim.org/release-docs/latest/api/Overview/`. [Accessed 24-04-2024].

31. FOWLER, Martin. *Patterns of enterprise application architecture*. Addison-Wesley, 2015.

32. FIGMA. *What is Wireframing? The Complete Guide [Free Checklist]| Figma — figma.com* [online]. 2024. Available also from: `https://www.figma.com/resource-library/what-is-wireframing/`. [Accessed 24-04-2024].

33. MATEE DEVS. *GitHub - MateeDevs/devstack-native-app: Matee KMP DevStack — github.com* [online]. 2024. Available also from: `https://github.com/MateeDevs/devstack-native-app`. [Accessed 27-04-2024].

34. GOOGLE. *Google I/O 2019: Empowering developers to build the best experiences on Android + Play — android-developers.googleblog.com* [online]. 2019. Available also from: `https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html`. [Accessed 28-04-2024].

35. LUTKEVICH, Ben. Kotlin. *WhatIs* [online]. 2022. Available also from: `https://www.techtarget.com/whatis/definition/Kotlin`.

36. GOOGLE. *Download Android Studio & App Tools - Android Developers — developer.android.com* [online]. 2024. Available also from: `https://developer.android.com/studio`. [Accessed 28-04-2024].

37. JETBRAINS. *Expected and actual declarations | Kotlin — kotlinlang.org* [online]. 2024. Available also from: `https://kotlinlang.org/docs/multiplatform-expect-actual.html#rules-for-expected-and-actual-declarations`. [Accessed 30-04-2024].

38. JETBRAINS. *Coroutines basics | Kotlin — kotlinlang.org* [online]. 2024. Available also from: `https://kotlinlang.org/docs/coroutines-basics.html`. [Accessed 01-05-2024].

39. JETBRAINS. *Composing suspending functions | Kotlin — kotlinlang.org* [online]. 2024. Available also from: `https://kotlinlang.org/docs/composing-suspending-functions.html`. [Accessed 03-05-2024].

40. GOOGLE. *Kotlin flows on Android | Android Developers — developer.android.com* [online]. 2023. Available also from: `https://developer.android.com/kotlin/flow`. [Accessed 01-05-2024].

41. GOOGLE. *Runtime Permissions | Android Open Source Project — source.android.com* [online]. 2024. Available also from: `https://source.android.com/docs/core/permissions/runtime_perms`. [Accessed 04-05-2024].

42. GEEKSFORGEEKS. Types of Software Testing. *GeeksforGeeks*. 2024. Available also from: `https://www.geeksforgeeks.org/types-software-testing`.

43. GOOGLE. *Firebase | Google's Mobile and Web App Development Platform — firebase.google.com* [online]. 2024. Available also from: `https://firebase.google.com/`. [Accessed 13-05-2024].
44. GOOGLE. *Firebase App Distribution — firebase.google.com* [online]. 2024. Available also from: `https://firebase.google.com/docs/app-distribution`. [Accessed 13-05-2024].

Attachments

`readme.txt`	concise description of medium content
`apk`	directory with the installation apk
`  tripplanner.apk`	installation file
`src`	directory with source codes
`  impl`	source codes of the implementation
`  thesis`	source codes of the thesis in LaTeX format
`text`	directory with the text of the thesis
`  thesis.pdf`	text of the thesis in PDF format