FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Application for finding travel companions
Student:	Mykhailo Liutov
Supervisor:	Ing. Ondřej Guth, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Create a software finder of people to organise a trip. Analyse the requirements, design and implement the application as a prototype. Its features should be original. Design the software in client–server architecture, where the client should be a mobile application.
The features of the application should include:
- User profiles and authentication
- Ability to find and create trips with special requirements
- Ability to create and browse posts made by users

Electronically approved by Ing. Michal Valenta, Ph.D. on 28 September 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Application for finding travel companions

Mykhailo Liutov

Department of Software Engineering
Supervisor: Ing. Ondřej Guth, Ph.D.

May 9, 2022

Acknowledgements

I want to thank my supervisor Ing. Ondřej Guth, Ph.D. for the guidance and my family for the support they gave me.

Declaration

In Prague on May 9, 2022 ………………….

Citation of this thesis

Liutov, Mykhailo. *Application for finding travel companions.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Tato práce popisuje proces analýzy, návrhu, implementace a testování systému pro vyhledávání spolucestujících. Tento systém se skládá z aplikace pro Android a serveru. Celý systém je vyvíjen pomocí moderních technologií. Server je nasazen do cloudového prostředí a využívá různé služby nabízené poskytovatelem cloudu.

Výsledkem této práce je kompletní systém, který je připraven k distribuci široké veřejnosti. Na základě provedeného uživatelského testování je zde prostor pro budoucí vylepšení a rozšíření systému o nové funkce.

Klíčová slova Cestování, klient-server, Android, Kotlin, Amazon Web Services

Abstract

This thesis describes the process of analysis, design, implementation, and testing of a system for finding travel companions. This system consists of an Android application and a server. The whole system is developed using modern technologies. The server is deployed into a cloud environment and uses various services offered by a cloud provider.

As the result of this thesis, a complete system that is ready to be distributed to a wide public was created. Based on the conducted user testing, there is space for future improvements and expansion of the system with new features.

Keywords Traveling, client-server, Android, Kotlin, Amazon Web Services

Contents

List of Figures

List of Tables

Introduction

A lot of people today enjoy traveling, however, they can often face the problem of finding someone to join them on their trip. In addition to this, they may not have all the equipment they need for their journey. This is the problem that the system created in this thesis is going to solve.

Currently, there exist some solutions for looking for travel companions, however, they do not cover the mentioned equipment requirements. By offering support for them, the developed system not only will allow users to go on trips where they could not have gone before but also to connect with others and possibly share their hobbies.

The goal is to develop a complete solution to the problem of finding co-travelers in an original way. It should support user accounts and trips with equipment requirements. Additionally, it should allow users to create and see posts, to give people inspiration for future journeys. The resulting system should consist of a server and an Android client and should be tested and ready to be used by real users.

The thesis is split into 4 chapters, each one corresponding to a stage in a software development process. The analysis chapter describes the process of formulating detailed requirements and use cases. The design chapter defines the architecture of the system, while the implementation chapter tells about all the details and important decisions that were made during the development. Finally, the testing chapter describes how the system was tested.

Chapter 1

Analysis

This chapter formulates functional, non-functional requirements and use cases and describes the research of the existing solutions to the problem of finding co-travelers and the analysis of their strengths and weaknesses. In addition to this, it highlights the ways in which TravelMates, which is the name of the developed application, has innovative and useful functionality.

1.1 Requirements

This section describes the functional and non-functional requirements. They were formulated based on the goals of this thesis and modern standards of software development.

1.1.1 Functional requirements

F1: User profile The system should allow any user to register with email and password and store their profile.

F2: Trips The system should allow the creation of trips for any registered user. Also, it should be possible to browse, search and join trips. When creating or joining a trip, the user has to provide their contact. The purpose of this is to allow communication between users since the application does not provide it[1]. It should be possible to provide a different one for each trip, to allow users to select the most suitable contact for every situation, for example, it could be an invitation link to a dedicated group chat for a specific trip.

F3: Equipment requirements It should be possible to add equipment requirements to a trip.

[1]The developed application is a prototype, and such feature would take a lot of time to implement.

F4: Posts The system should allow any registered user to create a post. Users should be able to browse a list of all users' posts.

1.1.2 Non-functional requirements

N1: Client Android application.

N2: Server Server should be running in a cloud environment with available scalability.

N3: Version Android application should support the majority of current Android devices.

N4: Localization Application should be in English, although it should be developed in a way to make it easy to add more languages in the future.

N5: Data security User passwords should be stored in a secure way.

1.2 Existing solutions

There already exist some products that offer a solution to the problem of finding travel companions. By using Google search and informational websites, I have selected products relevant to the stated problem, which means they offer some way to find other people in the context of traveling and fulfill some of the stated requirements.

1.2.1 JoinMyTrip

JoinMyTrip is a website, which offers the possibility to create and search for trips all over the world. Each trip includes a detailed plan for each day and has a price, which has to be paid to the service. This typically includes accommodation, transport, and other services. Compared to the goals of the thesis, this service lacks the ability to add trip requirements and posts and also enforces its payment system.

Advantages:

- Big platform with a lot of users and possible trips
- Built-in payment system, which allows people to know how much precisely the trip will cost

Disadvantages / Missing features:

- The payment system makes it more business-oriented for those who offer trips, instead of people just wanting to share their hobbies or to meet new travelers
- It is not possible to add equipment requirements for the trips

1.2.2 BeATravelBuddy

This service is simpler than the previous one mentioned. What is similar to the goals of this thesis is a list of user-created posts and user profiles. However, this service lacks the ability to create trips.

Advantages:

- Provides a way to discover other travelers nearby
- Allows to post user stories and browse them

Disadvantages / Missing features:

- No way to create trips and look for them

1.2.3 Meetup

Another service that is worth mentioning is called Meetup. It allows people to create local events, which other people can join. Theoretically, it can be used to organize trips as well, but it is aimed at events nearby. As a result, it differs from the goals of the thesis in the ability to add trip requirements and posts.

Advantages:

- Big community
- Simplicity of usage

Disadvantages / Missing features:

- Not oriented towards traveling
- No built-in support for trips requirements
- No user posts

1.3 Use cases

This section describes the use cases of the application. The use cases are split into 2 categories, *general* and *trip-related*.

1.3.1 General use cases

General use cases cover everything that is related to any registered or anonymous user. They are described in Figure 1.1

UC1: Log in Anonymous user should be able to log into the application using their email and password. Only those users which have verified their ownership of the email should be able to log in. This is done to ensure that the emails are authentic and allows to reach out to the registered users via email in the future.

UC2: Register Anonymous user should be able to register in the application.

UC3: Recover password Anonymous user should be able to recover their password. To do that, they will verify themselves as an owner of the email and set a new password.

UC4: Create post Registered user should be able to create a post with a picture.[2] It should be possible to pick a location, which is shown in the post, on a map.

UC5: Browse posts Registered user should be able to see posts created by other users.

UC6: Edit profile Registered user should be able to edit their profile. It should be possible to edit the picture and the name.

UC7: Send join request Registered user should be able to send a join request to any trip which is gathering people.

UC8: Create trip Registered user should be able to create a trip. It should be possible to pick the location of the trip on a map.

1.3.2 Trip use cases

Trip use cases describe everything related to trip owners and members, they are depicted in Figure 1.2

[2] Ability to edit or delete a post is intentionally not required, as the created application is a prototype, and those features are not essential for a functioning product.

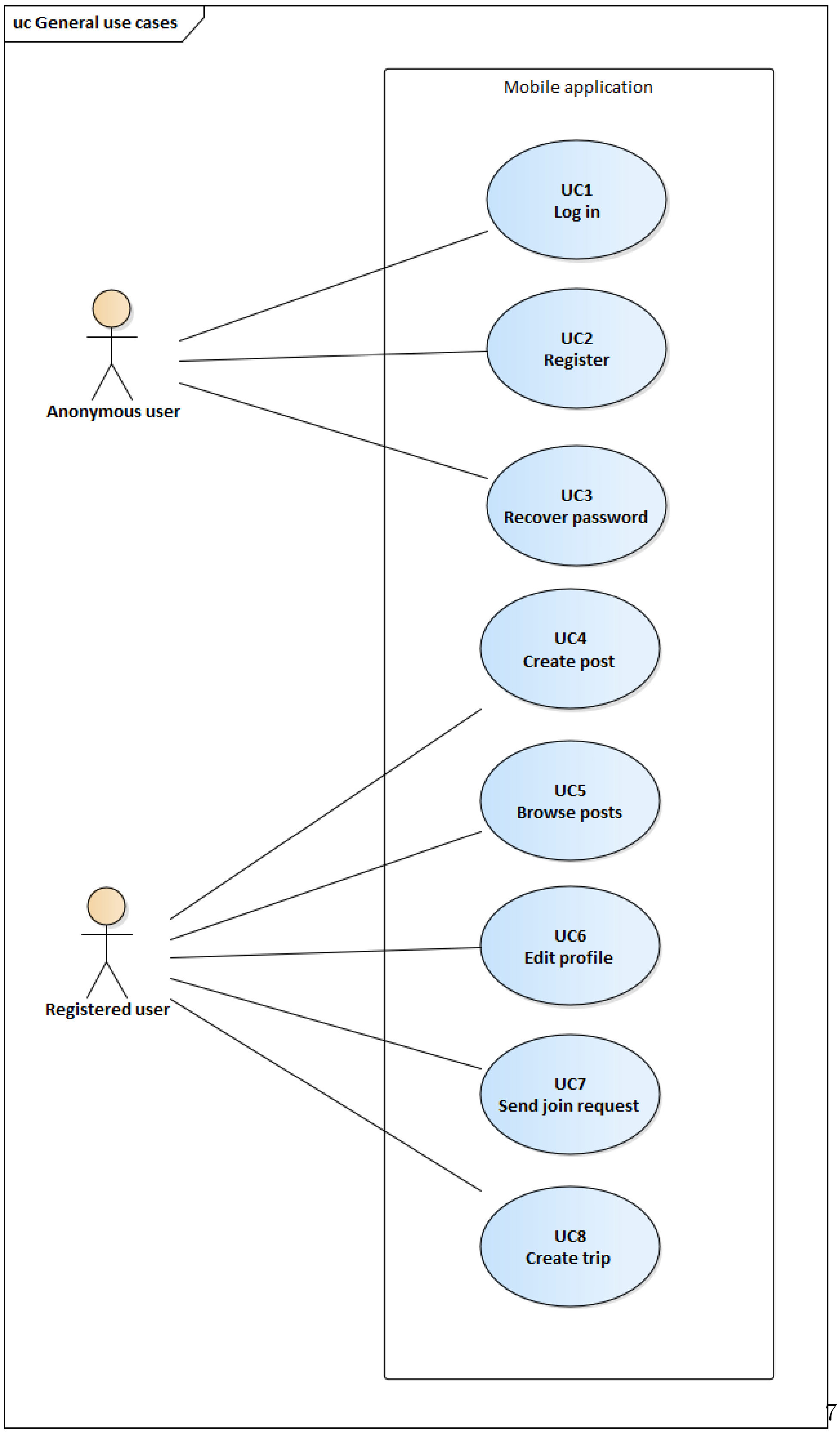


Figure 1.1: General use cases

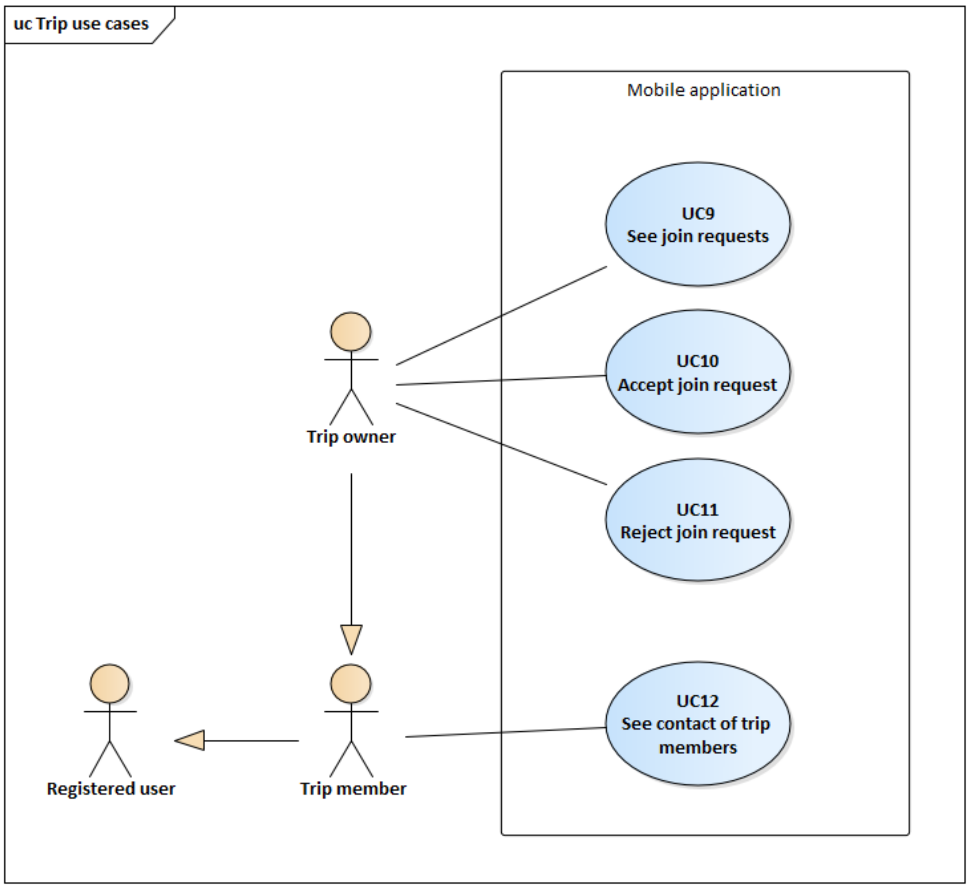


Figure 1.2: Trip use cases

UC9: See join requests Trip owner should be able to see join requests, which users sent to their trip.

UC10: Accept join request Trip owner should be able to accept any join request that a user sent to their trip. Doing so will make the sender a trip member.

UC11: Reject join request Trip owner should be able to reject a join request. When doing so, they should provide a reason for rejecting in a form of a text message. They can also choose whether to allow the sender to send a new join request for this trip.

UC12: See contact of trip members Trip member should be able to see the contact of other trip members.

Requirement	JoinMyTrip	BeATravelBuddy	Meetup
F1 User profile	✓	✓	✓
F2 Trips	✓	X	✓
F3 Equipment requirements	X	X	X
F4 Posts	X	✓	X

Table 1.1: Competition and requirements

1.3.3 Summary

Looking at the competition research, requirements, and use case analysis above, we can see similar services that help people find travel companions already exist. However, none of them fulfill all the requirements, which were specified based on the goals of this thesis. Each competitor and the requirements they fulfill are shown in Table 1.1. As demonstrated, none of the existing services have implemented the feature to set equipment requirements and use them as search filters. To conclude, there is still space in this market for creating a new solution, which will combine the best of the existing features, and add new ones, with built-in support for equipment requirements, a community-oriented approach, and user posts.

1.4 Domain model

Analysis of the requirements and use cases, which were described before, has led to the creation of the domain model, shown in the Figure 1.3.

There are 2 additional entities in the model which were not described before:

Trip state The system should keep track of each trip's state. A trip that is in the "Gathering" state can be seen by all users, and those users can send their join requests to such a trip. A "Gathered" state represents a trip that is not looking for more members but has not happened yet. Finally, the "Finished" state describes a trip that has already happened.

Request state When a join request is sent, it gets the "Pending" state. When the owner wants to reject a join request, they may decide to allow or block the user from sending more requests for the trip. This is represented using the "Rejected without resend" and "Rejected with resend" states.

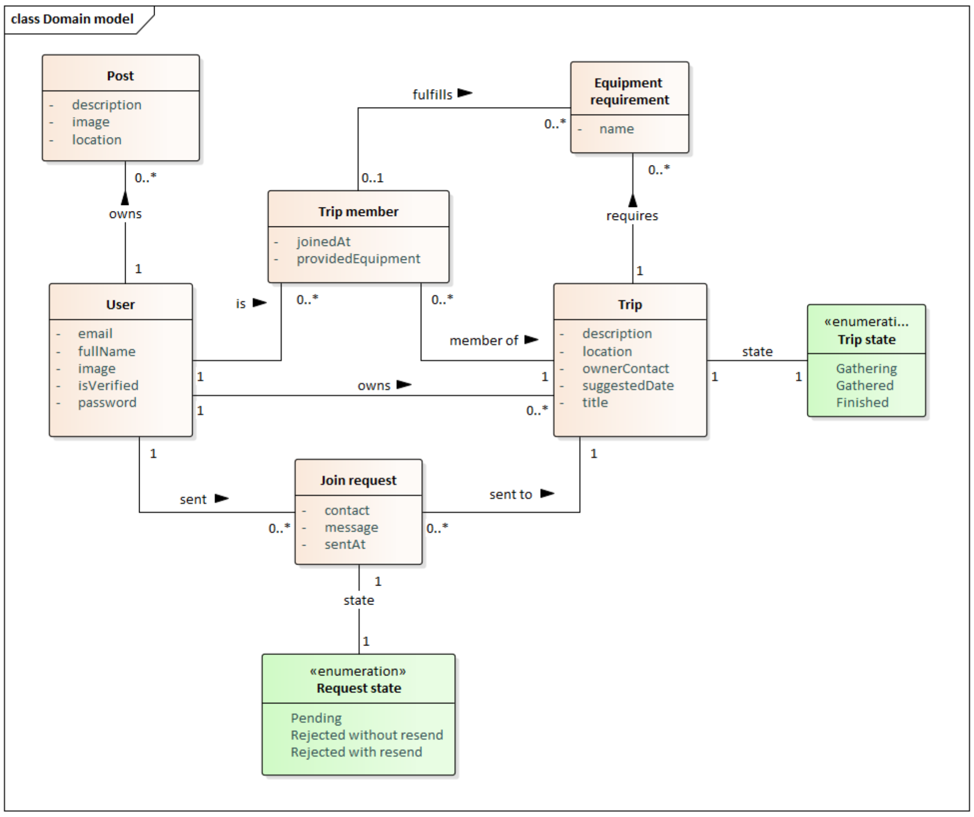


Figure 1.3: Domain model

Chapter 2

Design

2.1 Wireframes

In order to have an understanding of what the screens of the application will look like and help design the requests that the server will have to support, I have prepared wireframes. They are based on the use cases, which were formulated during the analysis.

Figure 2.1 demonstrates the screens related to authentication of a user, and covers use cases UC1, UC2, UC3. This process is described in more detail in subsection 3.3.2.

Figure 2.2 demonstrates the screens which allow users to see and create posts and covers use cases UC4, UC5.

Figure 2.3 shows the user profile screen, which also supports editing, and covers the use case UC6.

In Figure 2.4 screens related to join requests are shown. They cover the use cases UC7, UC10, UC11.

Figure 2.5 depicts screens that show details of a trip and cover the use cases UC9, UC12.

Lastly, Figure 2.6 shows the process of creating a trip, which covers the last remaining UC8.

This way, the implementation of the application based on the created wireframes will cover all needed use cases.

2.2 API design

Before starting the implementation, an application programming interface (API) definition needs to be created. In order to make future development easier, this definition should be hosted on a remote server to allow easy access for any future developers. For this reason, I have decided to use a service called Stoplight. It provides a user-friendly interface for creating the API documentation, and then hosts it on its own servers, making it publicly available.

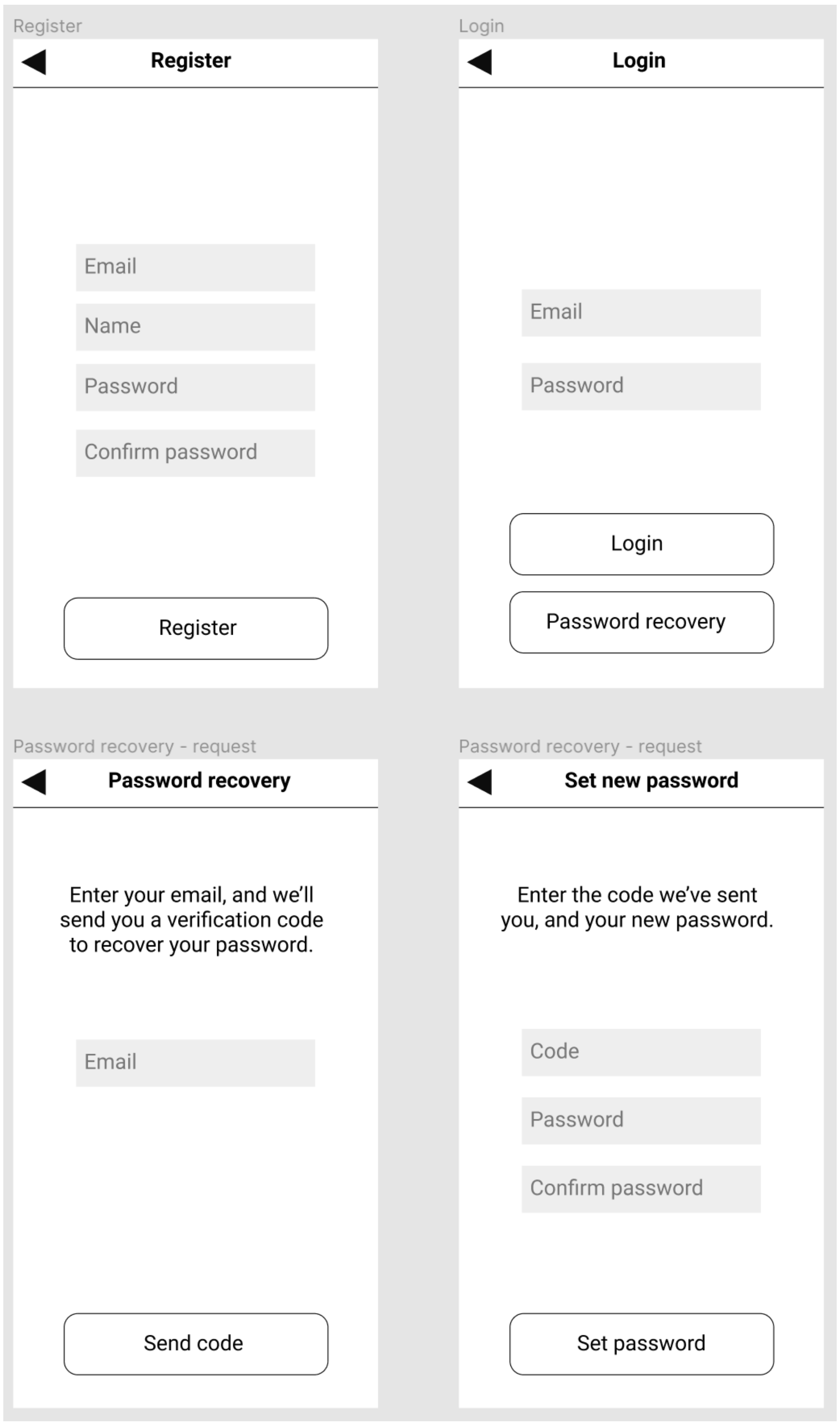

Register
Register
Email
Name
Password
Confirm password
Register
Login
Login
Email
Password
Login
Password recovery
Password recovery - request
Password recovery
Enter your email, and we'll send you a verification code to recover your password.
Email
Send code
Password recovery - request
Set new password
Enter the code we've sent you, and your new password.
Code
Password
Confirm password
Set password

Figure 2.1: Authentication wireframes

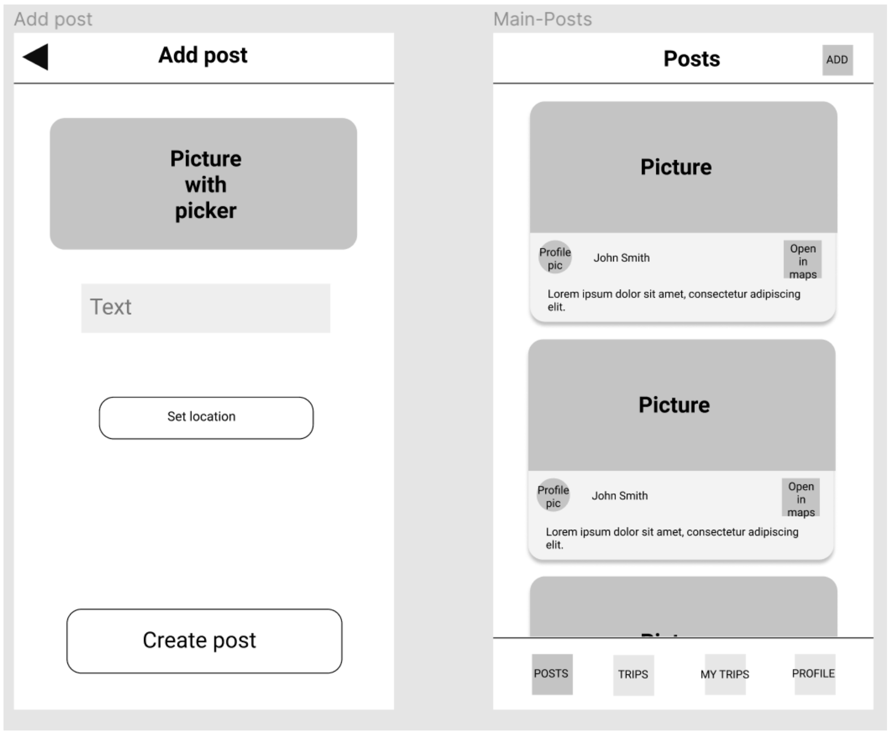


Figure 2.2: Posts wireframes

The design of the endpoints themselves follows the best practices of Representational state transfer (REST) API design from [1]. Based on the Domain model and mentioned practices, I have identified 4 resources and created the endpoints based on them:

Trips Those endpoints start with "/trips" and cover operations that are related to trips and their state.

Posts They start with "/posts" and allow to create and get a list of posts.

Join request Endpoints start with "/requests" and handle rejecting and accepting join requests.

User Endpoints from this group start with "/users" and expose the functionality of getting the user's profile and updating it.

To give an example of a design of a specific endpoint, we can take a look at the endpoint which allows changing the user's name and picture. As stated above, it belongs to the group related to users and operates directly on this resource, therefore it is suitable to use the path "/users" for it. The request itself is meant to update an existing entity, so the most suitable Hypertext

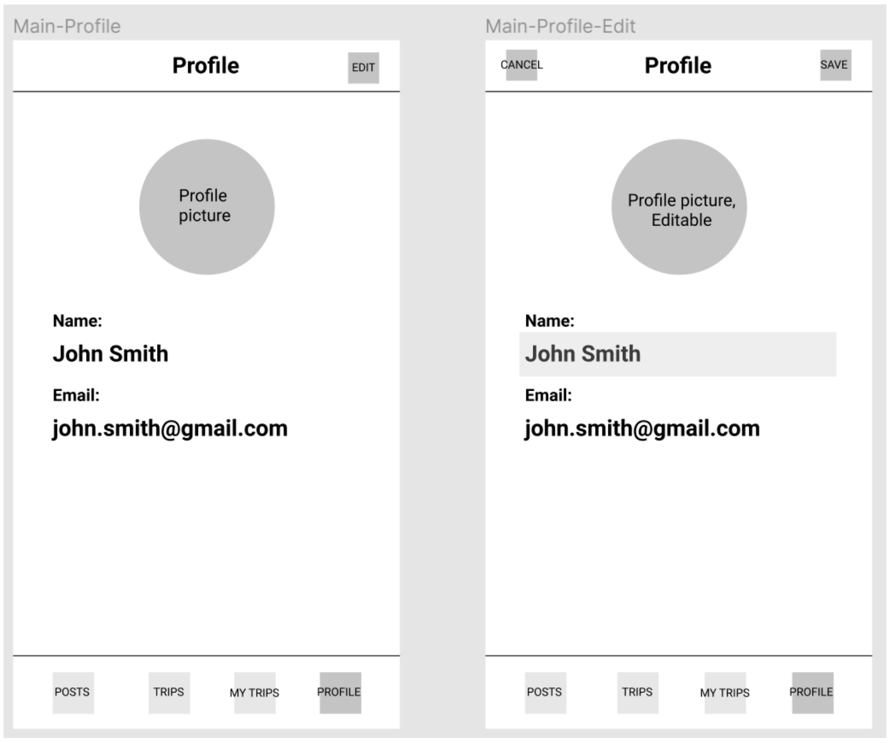


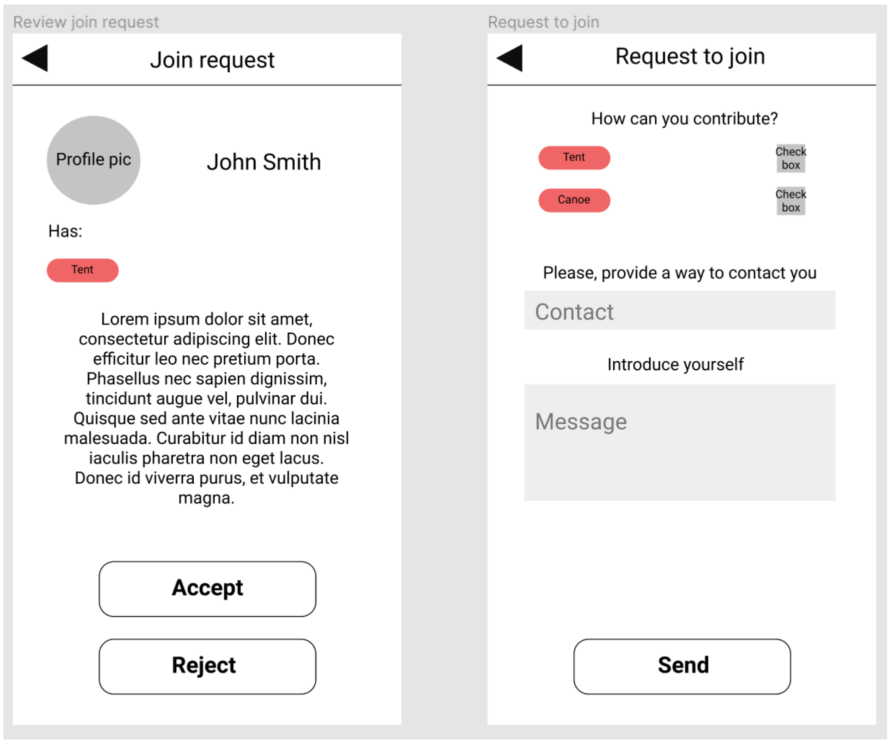
Figure 2.3: Profile wireframes

Transfer Protocol (HTTP) method, in this case, is PUT. The body of the request should include the fields that are being updated, meaning the name and the picture. An identification (ID) of the sender is included in the header of each request. If the operation succeeds, the server should return code 200 and the updated user profile, which the application will be able to present to the user.

2.3 System architecture

2.3.1 Account management

Requirement F1 states that the system should store user profiles and allow them to register. Since users tend to forget their passwords, it should be possible to recover a password. Also, requirement N5 says that the password should be stored in a secure way to avoid data leaks. Of course, this can be implemented using the conventional programming tools and a database, however, such custom implementation is prone to having security vulnerabilities. For this reason, I have decided to research and choose an existing solution for user identity management. Such solutions are closely related to cloud providers and therefore are described in the next subsection.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec efficitur leo nec pretium porta. Phasellus nec sapien dignissim, tincidunt augue vel, pulvinar dui. Quisque sed ante vitae nunc lacinia malesuada. Curabitur id diam non nisl iaculis pharetra non eget lacus. Donec id viverra purus, et vulputate magna.

Figure 2.4: Join requests wireframes

2.3.2 Cloud providers

According to the requirements, which were created during analysis, and based on requirement N2, which states that the server should be running in a cloud, I have formulated the following criteria for choosing a cloud provider:

1. It should support user account management, which includes the ability to register, login, and recover the password in a secure way.

2. It should be possible to host a server, which will be running around-the-clock to allow users to use the system at any time.

3. It should allow to host a database server, which would be accessible from the back end. A need for such a database is clear from functional requirements F2 and F4, which infer that data should be persisted and shared among different users.

Because the implementation of the needed features may differ based on the vendor, it is not possible to create a system architecture before choosing the cloud provider first. Therefore, it is necessary to find a service that will fulfill the stated criteria in the best way.

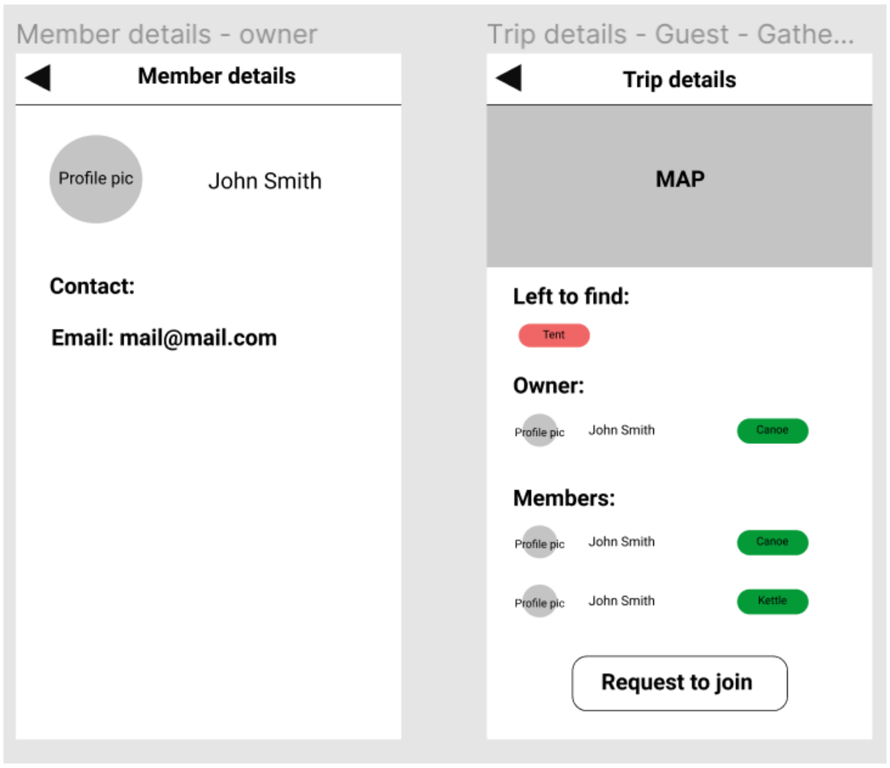


Figure 2.5: Trip details wireframes

Based on [2], there are 3 biggest cloud providers, Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). All of them satisfy the requirements, however, there is a difference in ease of usage and documentation when it comes to user account management.

AWS They provide a software development kit (SDK) called Amplify, which implements all needed logic in a form of Kotlin functions. The documentation is well written and clear.

Azure An SDK called Microsoft Authentication Library is also provided, however, the interfaces they provide are outdated by modern Android development standards, and documentation is often unclear.

GCP Firebase, a platform developed by Google, provides a modern and well-documented SDK to access their identity management system.

As we can see, all providers offer some kind of SDK that simplifies the development. However, AWS and GCP provide better interfaces and documentation, which are also important. To summarize, both options are equally good, so I have made my decision based on a fact that I already had previous experience with AWS and knew how to work with their products.

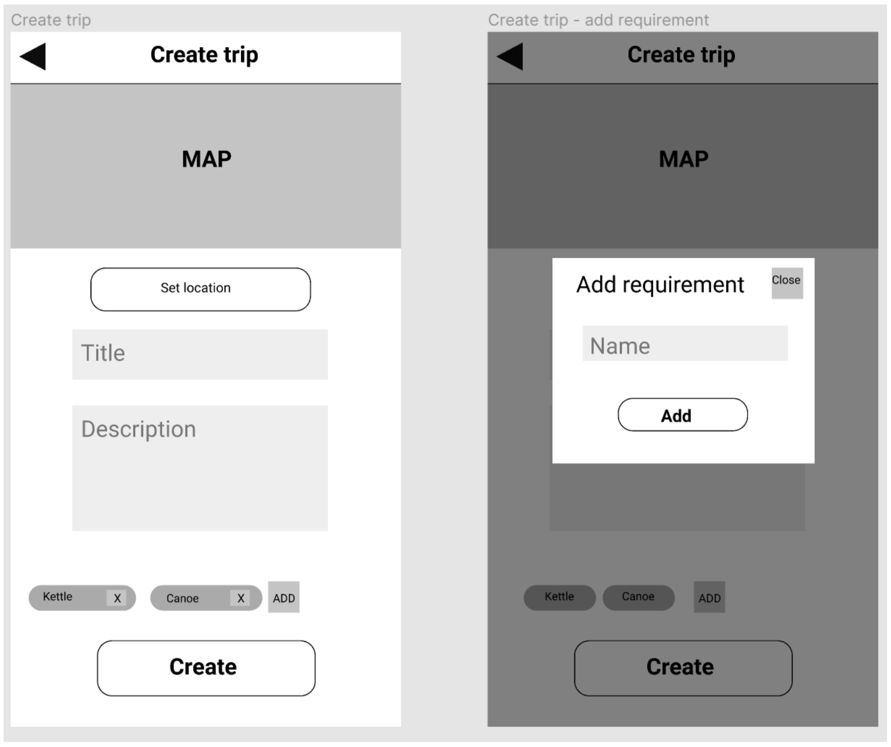


Figure 2.6: Create trip wireframes

2.3.3 Final architecture

Since the decision about how user management is going to work has been made, it is now possible to design the full system architecture. As depicted in Figure 2.7 and in accordance with the system requirements, the architecture consists of an Android application client, a server, and cloud services. AWS Amplify SDK is used in the client to authorize the user with the account management service, Cognito, which is described in more detail in subsection 3.3.2. After this is done, the HTTP client in the application gets an authorization token from the SDK and uses it to authorize requests to the server. The server uses Cognito to validate that token is valid and grants the user access to requested resources. All this communication is handled via HTTP. Communication with the database, which is also running in the cloud, is done using Java Database Connectivity (JDBC).

2.4 Android design

This section describes all important decisions in the design of the Android application.

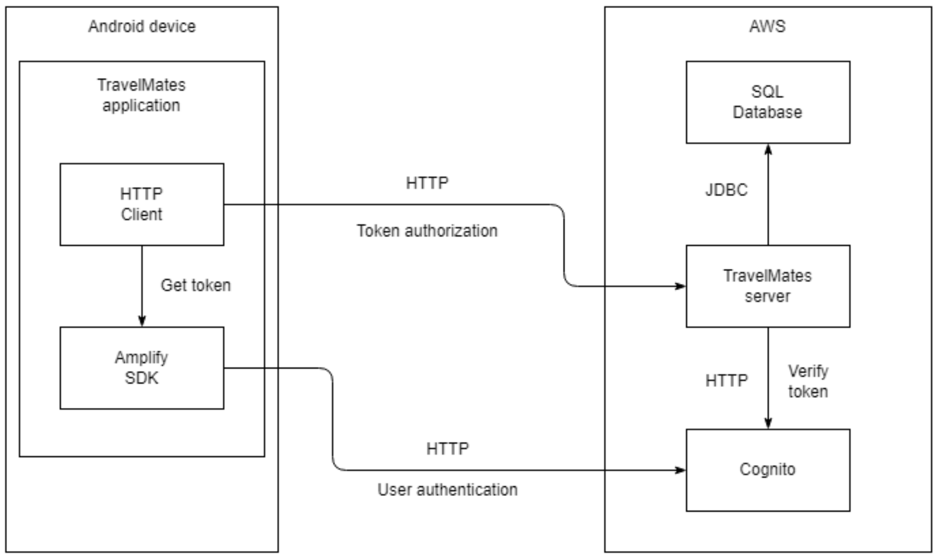


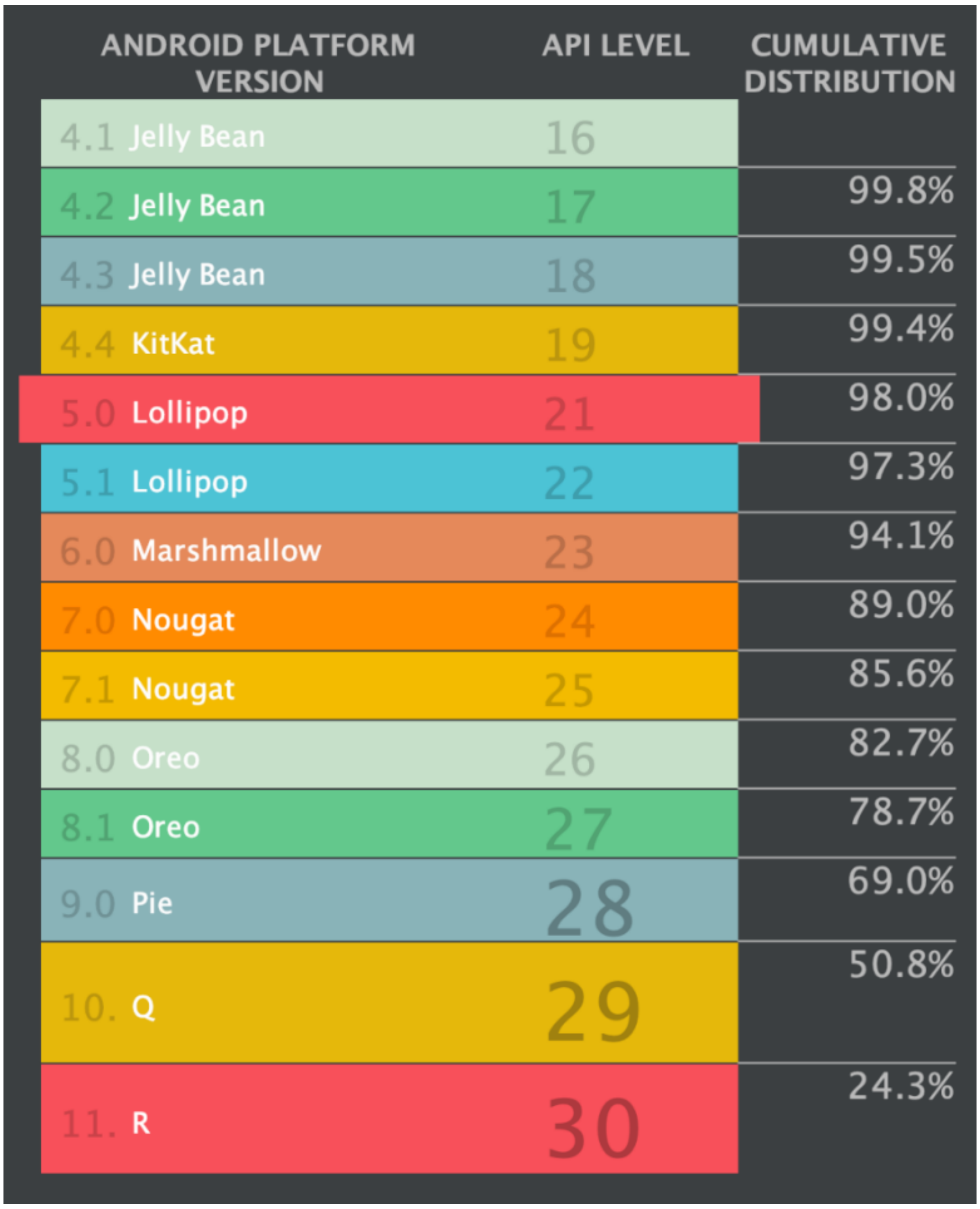
Figure 2.7: System architecture

2.4.1 Android minimum version

As stated in requirement N3, a decision that has to be made is to select a minimum version of the Android operating system (OS) that the application will support. The benefits of setting a lower requirement for the version are clear – it will allow more users to download and use the product. However, this may create a lot of problems during the development due to the need to use only backward-compatible components and the lack of modern features on older devices. As a result, a compromise between the covered percentage of the market and available components and features has to be made. According to the latest information provided by Google, which is depicted in Figure 2.8 [3], 98% of devices are running at least Android 5.0 Lollipop, which fulfills the given requirement of supporting the majority of devices. Also, it needs to be mentioned that those who are using devices with Android versions lower than 5.0 are most likely older people, who are not in the target audience of the application described in this thesis. Therefore, Android 5.0 was chosen as a minimum supported version.

2.4.2 Application architecture

Choosing a good architecture is a crucial decision that has to be made before development begins. I have decided to follow the architecture recommended by Google [4], which is shown in Figure 2.9. The biggest advantage of following the suggested approach is the separation of concerns, which is achieved by



ANDROID PLATFORM VERSION		API LEVEL	CUMULATIVE DISTRIBUTION
4.1	Jelly Bean	16	
4.2	Jelly Bean	17	99.8%
4.3	Jelly Bean	18	99.5%
4.4	KitKat	19	99.4%
5.0	Lollipop	21	98.0%
5.1	Lollipop	22	97.3%
6.0	Marshmallow	23	94.1%
7.0	Nougat	24	89.0%
7.1	Nougat	25	85.6%
8.0	Oreo	26	82.7%
8.1	Oreo	27	78.7%
9.0	Pie	28	69.0%
10.	Q	29	50.8%
11.	R	30	24.3%

Figure 2.8: Android versions distribution

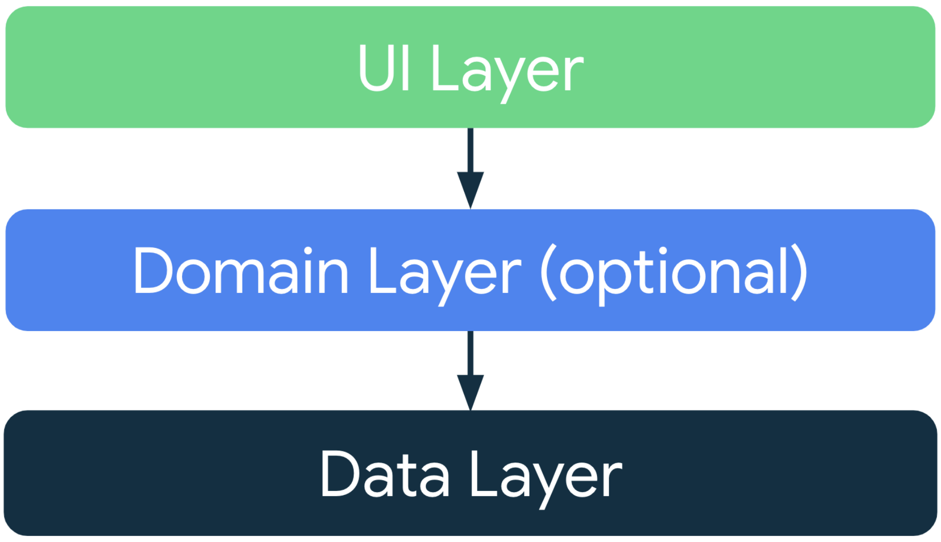


Figure 2.9: Recommended architecture by Google

splitting the user interface (UI), domain, and data layers. Due to the nature of the developed product, there is not much business logic in the client, since all such logic should be handled by the server, so the optional domain layer should only be present in case it is needed.

2.4.3 Android components

When designing the architecture, especially the UI layer which heavily relies on the Android framework, it is extremely important to take into account the way its components work.

One of the main components of the Android framework is called *Activity* [4]. An activity is a single thing that a user can do and usually represents a full-screen window. There exist multiple states and callbacks related to those states, which are typically referred to as lifecycle. It is important to note one mechanism of the lifecycle. When the application goes into the background, the system can kill the activity in order to free memory, and recreate it when a user comes back. The same thing happens when the user rotates their phone. When this happens, the state of the Activity is lost, therefore it is necessary to save the state elsewhere.

Another fundamental component is called *Fragment* [4]. It is a reusable part of the UI which has its own lifecycle, somewhat similar to the lifecycle of the activity. Fragments are also destroyed and recreated during the life of the application and do not preserve their state automatically.

2.4.4 UI layer

This paragraph is based on [4]

The recommended approach to creating a UI layer is MVVM, which stands for Model, View, ViewModel, and solves the issues with saving the state of Activities and Fragments. The state is represented using the models and is stored inside the ViewModels, while the view is kept in sync with the state of the ViewModels using the Observer pattern, creating a loose coupling between them. The implementation of the ViewModel class is provided by the Android framework and is able to survive configuration changes (rotation of the screen) and does not get destroyed as quickly as the UI components.

The implementation of the Observer pattern, which is used for the synchronization of the UI and the ViewModels, is also possible with the use of Android classes. *LiveData* [4] is an observable entity, which is also *lifecycle-aware*, meaning it respects the lifecycle of other components such as Activity and Fragment and triggers the observers only when they are in the active state. Without this feature, a situation where the UI is already destroyed, but the observer would try to update it would be possible, leading to crashes.

2.5 Server design

This section describes the design of the server part.

2.5.1 Architecture

The architecture of the server is very similar to the client's one. It is split into 3 layers:

Presentation layer This layer exposes the functionality of the server via REST API and consists of rest controllers, which handle communication over HTTP protocol.

Domain layer This layer contains the business logic of the server.

Data layer This layer is responsible for persisting the data. For the developed server, it is connected to an SQL database.

To keep the architecture clean, the presentation layer only communicates with and depends on the domain, and the domain depends and communicates with the data layer, there is no inverse dependency. This ensures that those layers are loosely coupled, and therefore are easy to change in the future.

CHAPTER 3

Implementation

3.1 Android implementation

This section describes all the important steps and decisions which were made during the implementation of the Android application.

3.1.1 Programming language

There are 2 main programming languages that can be used for Android development, Java and Kotlin. However, since 2019, Kotlin has become the main language for Android. Kotlin offers a lot of benefits when compared to Java, which are described below and are based on [4] and [5]

Kotlin advantages:

- *Compatible.* Kotlin is fully compatible with Java, which even allows mixing Kotlin and Java classes in one project if needed.
- *Concise.* Kotlin allows using less code to solve problems.
- *Safe.* Kotlin's type system is null-safe.
- *Coroutines.* Kotlin provides out-of-the-box support for coroutines, benefits of which are described in more detail in subsection 3.1.4.

Listing 3.1 demonstrates Kotlin's null-safe type system in comparison with Java. From it, it is visible that the same logic can be written with less code and in a more readable way in Kotlin, using the safe call ".?" operator. In addition to this, the listing shows one more feature of Kotlin that Java does not support called extension function [5], in this case "orEmpty()", which returns an empty list if the given list is null. Such extensions help to add functionality to existing classes while keeping the code easy to read.

To summarize, Kotlin is less verbose, requires less boilerplate code, is safer, and therefore is a clear choice for modern Android development.

Listing 3.1: Kotlin vs Java null safety

```
//Kotlin
fun searchList(
    list: List<String>?,
    term: String
): List<String> {
    return list?.filter { it.contains(term) }
        .orEmpty()
}

//Java
public List<String> searchList(
        List<String> list,
        String term
) {
    return list == null ? Collections.emptyList()
            : list.stream()
            .filter(item -> item.contains(term))
            .toList();
}
```

3.1.2 Kotlin DSL

Since building an Android project requires multiple steps such as lint analysis, annotation processing, compilation, packaging, and many others, it is necessary to use some build automation tools. For creating Android applications, Gradle is typically used. Based on [6], using Gradle brings a lot of benefits such as:

- Fast builds due to usage of build cache and other optimizations
- Ease of usage with support for custom tasks
- Great integration with IDE-s, such as Android Studio – the main tool for Android development

This paragraph is based on [7]

For writing Gradle scripts and configuration, a language called Groovy was typically used. The biggest problem with this language is that it is not widely used, and therefore most programmers do not know its syntax well. To solve this issue, support for writing scripts in Kotlin was added. The biggest benefit of choosing Kotlin domain-specific language (DSL) instead of Groovy is that Android developers already know it and are able to write the scripts and configuration for Gradle more easily, using the familiar syntax, and with auto-completion available in the Android Studio.

3.1.3 Modularization

This subsection is based on [8]

Although is it possible to write the whole project in one module, it is not a good approach since it does not scale well, and the build time significantly increases with the growth of the codebase. For this reason, I have decided to use multiple modules to develop the application.

The most widely spread way to modularize an application is using the so-called *feature modules*. When using this way, each module contains all the code related to a specific feature. I have split the client into the following modules:

app This is the main module of the application, the purpose of which is to connect all the feature modules.

auth This module contains all the screens related to the process of authentication, such as registration, login, and password recovery.

authapi Since the authentication logic is used by multiple other modules, I have decided to put it into a separate module. The authentication is done using the Amplify SDK, so this is mostly a wrapper around the SDK, which provides an extra level of abstraction and loose coupling.

core This is a special module, which is not related to a specific feature but instead contains helper functions, extensions, and other auxiliary code which is used by multiple feature modules.

images The logic of uploading an image to the remote storage is used in multiple places, therefore I have extracted it into this module.

location This module provides the functionality related to locations, more specifically models, utility functions, and a screen where a user is able to pick a location.

mainapi This module handles all the communication with the back end.

posts Everything related to the posts is located in this module.

trips All features connected to trips are implemented in this module.

3.1.4 Coroutines

This subsection is based on [5]

Naturally, when developing an Android application, blocking the main thread to load data, thus freezing the user interface, is an extremely bad user experience and is not acceptable. Therefore, some approach to handle asynchronous programming has to be selected. Since it is an old and widespread issue, there are many existing techniques to solve it.

Threads:

- Multiple threads are used to avoid blocking the application
- Synchronizing the threads and avoiding race conditions can be tricky

Callbacks:

- A special callback function is called when a long-running method finishes
- Nested callbacks are hard to read, and error handling becomes complicated

Reactive streams:

- All data is represented using observable streams which react to changes in the upstream data sources
- It is a completely different approach from traditional ones and requires a big change in thinking

All of the listed solutions are usable, however, Kotlin introduces a different method to write asynchronous code called coroutines. At the center of this approach is an idea of a suspendable computation, in other words, a function that is able to suspend its execution and resume at a later point. This is done by marking a function with keyword *suspend*. This allows to write asynchronous code in a synchronous way, meaning the resulting code is easy to read and understand, and developers do not need to change their programming style unlike with reactive streams.

Each coroutine has its context, including a dispatcher, which helps decide on which thread a coroutine should be executed. There exist special dispatchers optimized for Input/output (IO) or computationally intensive tasks.

3.1.5 Image storage

Following the analysis, it is necessary for the application to store images in a cloud. There are multiple options of how to do that. One of the options is Firebase Cloudstore [9], a service provided by Google, which offers cloud file storage with an SDK that is easy to use. Firebase SDK has to be integrated into the application regardless of this decision because its other services such as Crashlytics[3] are used in the application.

Another product that could be used is AWS S3 [10], which is integrated into Amplify SDK, already used within the project.

Looking at the options, we can see that they are completely equivalent in both features and usage. However, having previous experience with Firebase Cloudstore, I have decided to choose it.

3.1.6 Libraries

A number of libraries were used to simplify the development. The most notable of them are described below.

3.1.6.1 Navigation

As described in subsection 2.4.3, the main UI components on Android are *Activity* and *Fragment*. Currently, the recommended approach to creating an application by Google is to use a single Activity and represent different screens using Fragments. This is supported by the *Navigation component* [4] library, which handles complex navigation, arguments, and deep links. Each screen is called a destination, and possible navigation directions and arguments are described using *navigation graphs*.

3.1.6.2 Hilt

In modern application development, it is crucial to use dependency injection (DI), preferably an automatic one, using a library. One of the existing DI libraries for Android is called Hilt [4], and it is recommended by Google. Hilt was built over an older library, Dagger, with the goal of simplifying the Dagger infrastructure, improving readability, and offering better integration with the Android framework.

The key component of Hilt is a module. A module is a collection of bindings, each one of them defines how to create a dependency of a certain type. Those definitions are later used by Hilt for automatic injection. Each module has to be installed in a certain component, which defines which Android components it will be used in. A binding can have a scope, which determines when a certain dependency is created and destroyed, and how many instances

[3]A service for tracking application's crashes in production.

of it can exist. A notable scope is Singleton, which allows only 1 instance of a given dependency.

3.1.6.3 Retrofit

Retrofit [11] is a type-safe HTTP client, which supports all HTTP methods and handles such things as parsing of requests and responses into data objects (using a JSON parser called Moshi). All requests are defined as methods of an interface, the implementation of which is generated by Retrofit and can be called like usual Kotlin functions. In addition to this, it natively supports suspending functions, which makes making network calls very easy.

3.1.6.4 Firebase

As already mentioned in subsection 3.1.5, Firebase needs to be integrated into the project for multiple purposes.

One of the purposes is to upload and retrieve users' images. The SDK of Firebase makes the process very easy by providing Kotlin functions and support for coroutines.

The second goal of this library is to enable Crashlytics [12], a service that helps track crashes in production. A crash report, which is generated and sent to Firebase's servers, includes helpful information such as the Android OS version, device name, and a stack trace, which helps identify the cause of the issue and later fix it.

3.1.7 Layers

The application follows the architecture which was introduced during the design and is split into layers accordingly.

The UI layer uses the MVVM architecture. The View-Models depend on domain and data layer only, those dependencies are injected using Hilt.

The Domain layer consists of classes that represent specific use cases, and are only used where more complex business logic is present. To give an example, the class "SearchTripsUseCase" has a single public method that provides the logic of filtering a list of trips based on a search query.

The Data layer consists of classes that provide data from various sources such as SDK-s, API-s, and local storage.

3.2 Server implementation

3.2.1 Server framework

When choosing the framework for developing the back end side, I have formulated the following criteria:

1. It has to support Kotlin as a development language in order to develop the whole system, including server and client, using one language.

2. The framework has to have a rich selection of libraries, including dependency injection and security, which will decrease the development time.

After researching available options, I have selected 2 possible candidates: a well-known framework called Spring, and a completely new product called Ktor.

3.2.2 Spring

Spring Framework is very popular for developing Java applications. One of its purposes is to simplify the development of servers. Being first released in 2004 [13], it offers a lot of pre-made solutions for most of the common problems such as security, dependency injection, and testing. However, this can also be a disadvantage, as the whole framework is extremely big and therefore is hard to fully learn.

Advantages:

- Mature framework with ready solutions for most problems
- Kotlin support
- Big community

Disadvantages:

- Massive code base which is hard to learn
- Although Kotlin is supported, Spring is Java-first. As a result, it does not utilize the advantages of Kotlin's syntax and lacks support for features such as coroutines

3.2.3 Ktor

Ktor is a new framework for creating servers that, at the time of writing, is in active development. It was created as Kotlin first and utilizes all the modern features of this language such as concise syntax and coroutines. On the other hand, being completely new, the number of available plugins and libraries is smaller, and some problems may not have a solution by the framework.

Advantages:

- Young, Kotlin-first framework with modern design
- Lightweight, relatively easy to learn

Disadvantages:

- Framework's active development may introduce a lot of breaking changes in later versions
- Relatively small community
- Not all use cases may be supported

3.2.4 Final decision

As shown above, both frameworks have their advantages and disadvantages. Spring offers the benefits of a very mature product, whereas Ktor is new and modern. For the development of the application described in this thesis, I have decided that possibility of implementing all the common use cases with the usage of framework tools is more important, so I have chosen to work with Spring.

3.2.5 Libraries

Similar to the client, some external libraries were used during the development, they are described below.

3.2.5.1 Spring Boot

Spring Boot is the framework that was used to develop the server. It includes multiple modules, such as starter, which provides the core features of Spring Boot, security, which handles authentication of users, and Jakarta Persistence (JPA) module used to connect and work with databases. Another notable module is called the actuator. It automatically creates utility endpoints, which can be used to get information about the state of the server. One of the most important ones is the "health" endpoint, which does not require any authorization, and returns HTTP code 200 when the server is up. This endpoint must be present in order for the cloud provider to know that server is successfully running.

3.2.5.2 Auth0

The authorization is done using JSON Web Tokens (JWT), which are issued by the account management service of AWS. The signature on the tokens is validated by the server using the public keys. As a result, it was necessary to add libraries to work with those tokens. The first of these libraries is Auth0 Java-JWT, which allows to parse and validate the signature on the given token. The second one is Auth0 JWKS-RSA, which simplifies the fetching of public keys from Cognito, using which the signature is verified.

3.2.5.3 MockK

MockK is a mocking library used for writing unit tests. Typically, a different and older library called Mockito is used, however, MockK was specifically designed to work with Kotlin, and therefore features better syntax and easier usage.

3.2.6 Project structure

The server was developed using the architecture specified during the design, with the presentation, domain, and data layers.

The project is split into *feature-packages*, meaning each sub-package contains code related to a specific feature. Those sub-packages are structured with a separation-by-layers approach. This allows to find the necessary classes quickly and keeps the structure clean.

3.3 Deployment

3.3.1 Cloud infrastructure

In order to deploy the server into the production environment, a cloud infrastructure needed to be configured. To visualize it, I have created a diagram which is shown in Figure 3.1. It mostly follows the design architecture with a couple of new services.

One of them is the load balancer, which acts as an intermediary between a user and a server. Since the environment where the server is hosted can be scaled by running multiple instances of the server, a load balancer is needed to distribute the incoming requests between those instances.

The second new service is Route 53, which acts as a Domain Name System (DNS) server. The reason why DNS is needed is described in the subsection 3.3.4.

3.3.2 AWS Cognito

This paragraph is based on [14]

Cognito is an identity management solution provided by AWS. It stores user profiles in so-called “user pools” and provides all features that are necessary for modern applications with user profiles such as registration with email verification using a verification link, multi-factor authentication, and secure password recovery.

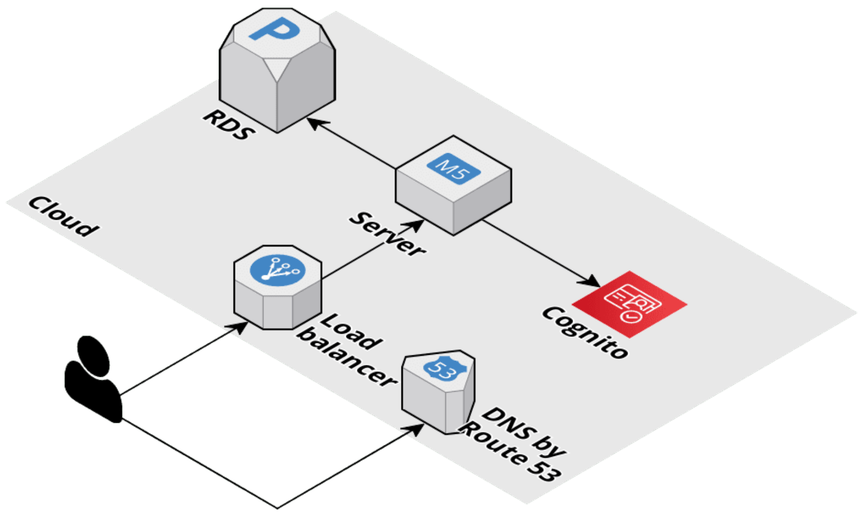


Figure 3.1: Cloud infrastructure

In order to work with Cognito, I have created and configured a user pool according to the requirements and use cases. To fulfill the use case UC1, which mentions that only verified users can log in, I have enabled verification of users by verification link, which they receive in their email after registering. In addition to this, to implement UC3, which requests the user to verify themselves as the email owner, I have configured Cognito to send a security code to the user's email. The user then needs to enter it in the application in order to set the new password.

3.3.3 AWS RDS

This paragraph is based on [15]

Next, a database of the project has to be set up. Fortunately, a service called RDS (Relational Database Service) is available on AWS. It is a fully managed cloud database that is extremely easy to set up, operate and scale. It supports various database engines such as MySQL, Oracle, and PostgreSQL, although only the MySQL engine is included in the free tier plan of AWS usage, so that is the one I have chosen.

I have configured the database to automatically create snapshots of itself every day, with those snapshots being stored for 10 days. This ensures the safety of users' data even in case of database corruption by allowing restoration of the data from the snapshot.

3.3.4 Secure communication

By this point, the communication between the server and the client was done via the insecure HTTP protocol, which is suitable for development purposes but has to be changed before deployment to avoid exposing user data in unencrypted format.

In order to make the communication secure, Hypertext Transfer Protocol Secure (HTTPS) needs to be configured and used. To do that, it is necessary to get a Secure Sockets Layer (SSL) certificate issued for the domain name that is used by the server. Since it is not possible to issue such a certificate for a domain provided by AWS, I had to purchase my own domain name.

After the domain was purchased, the DNS had to be configured to forward users to the server's location. For this purpose, Route 53 was used. First, I have configured my domain to use the name servers of AWS, and after that, I have set up Route 53 to route traffic to my server.

As the last step, an SSL certificate had to be issued. This is possible to do directly through AWS, using a service Certificate Manager. Once the certificate was issued, I assigned it to the load balancer. The communication between the balancer and the server is then done via HTTP, which happens inside the AWS infrastructure and thus is secure even without encryption.

3.3.5 Android deployment

Since the application is not publicly distributed yet, the deployment is done by creating an installation package and distributing it manually. Through a Gradle script, I have configured the build process of the installation package, called Android Package (APK). I have set the "versionCode", which is a number that represents the current version of the application, to use the size of the git log (the total amount of commits), which ensures that the version automatically increases with each new commit. The "versionName" parameter, which is a human-readable name of the version, is set and updated manually when preparing a new release. I have also set up the name of the produced package to include the name of the application, the version code, and the version name, which allows testers to easily identify the version they are using.

The APK is assembled by running a Gradle task. There are 2 versions of the application that could be built:

Debug Such a version is meant for development purposes. The resulting application allows to use a debugger to debug itself, prints logs into the console, skips optimization steps during the build, and is not signed.

Release This version is ready for release. It is optimized, its resources are shrunk and code is obfuscated to prevent reverse-engineering. In addition to this, the resulting APK file is signed with a private key, which prevents unauthorized updates of the application by unknown parties.

Chapter 4

Testing

This chapter describes the process of testing the implemented system.

4.1 Unit tests

To verify that the code works as expected on the lowest level, unit tests are used. When writing such tests, a class is isolated, its dependencies are mocked, and the behavior of each method is checked, including boundary cases. Such an approach allows to ensure that components work correctly in isolation, however, this does not verify the correctness of interaction between those parts.

In the server, the unit tests cover the domain layer. The implementation of the presentation and data layers heavily relies on the external libraries, which are tested by their creators, and does not contain any business logic. They were tested manually during the process of development.

In the client, the domain layer and all the ViewModels are covered by unit tests, which verify all the business logic of the application. The proper work of the UI and the data layer was tested manually.

4.2 User tests

Finally, to verify how real people would interact with the client and to find possible problems and future improvements, I have conducted user tests. I have selected users from people who, in my opinion, would be interested in using the application. I also tried to choose testers with different backgrounds and experiences in using Android devices to get feedback from various perspectives.

When performing a test, I gave a user a phone with the installed application, account credentials, and a test scenario with actions that they had to complete. The provided account was set up beforehand with testing data.

During the test I observed the user and did not interfere, to simulate a real user experience. The scenario was designed to cover all the common use cases and is shown below.

1. Open the application and log in using the provided credentials.
2. Try to update the picture of your profile and change your name.
3. Try to create a post.
4. Try to find a trip you could join, imagine you want to provide a boat. Request to join this trip.
5. Open your trip where you are the owner. Review and reject the pending join request, imagine you want to allow the person to send new requests.
6. Try to create a trip.

After the user was done with the scenario, I have asked these control questions:

1. Was it easy to navigate through the application and find where to complete the requested actions?
2. Did all UI elements behave as you would expect, was there anything confusing?
3. What were the 2 things in your experience that you liked the most?
4. What were the 2 things in your experience that you did not like?

The purpose of the first two questions was to evaluate the implemented design and find any possible flaws in it. The last two questions are product-oriented and are designed to detect what users value the most in the created product, and what they think is wrong with it.

The results of the testing were the following:

User 1

1. I was able to find all the requested actions, however, the structure of the application is somewhat unclear. I would expect to be able to create a trip from the "My trips" screen.
2. Yes, the behavior of the elements was predictable.
3. Clean design and good responsiveness.
4. Only the structure, which was unclear in some places.

User 2

1. Most of the actions were clear, however, I had trouble finding how to create a post. Also, I was missing the ability to create trips from “My trips”.

2. Yes, I did not see anything confusing.

3. The idea of the application, a big number of features.

4. The user experience (UX) could be improved. In some places, the design could be better as well.

User 3

1. Mostly yes, but I think it could be improved.

2. Yes, the behavior is standard for Android.

3. Features of the application, design.

4. The lack of a way to communicate through the application. Also, there is space for improvements in the UX.

As seen from the experience of these users, the UX of the application could be improved by bringing a more clear structure, and it would be my highest priority for future improvements. A good thing to note is that all of the users were able to complete the requested actions, which means that even in its current form the application is usable, and the detected problems can be solved gradually, with more feedback coming from the users.

2 users mentioned that they lack the ability to create a trip from the “My trips” screen, and 1 user complained that it was hard for them to create a post. Based on this feedback, I have added the requested button to the “My trips” screen and a button to create a post from the home screen.

4.3 Future improvements

Based on the testing, there is space for future improvements. They include:

UX As discovered during the testing, the structure of the application can be unclear. Therefore, it could be improved by conducting more user tests and discovering which UI and UX would meet users’ expectations and needs best.

In-app communication For now, the suggested way for users to communicate is to use an external service such as social media or a chat. As mentioned by a tester, it would be beneficial to instead have an in-app way for users to communicate with each other as it would keep the user from leaving the application, thus improving retention.

Conclusion

The goal of this thesis was to develop a system that would help people find co-travelers with support for equipment requirements and posts. The accomplishment of this goal started by analyzing the existing solutions and formulating detailed requirements and use cases.

After that, based on the analysis, the application was designed. That included a specification of the architecture of the client, server, and system as a whole. The architectures were created with best practices and modern approaches in mind.

Following the design, the system was implemented with the use of suitable libraries. All analyzed requirements and use cases, such as user authentication, creating trips, and searching for trips based on requirements and posts, were successfully implemented. An advanced setup on AWS was performed in order to ensure that the server is scalable and to allow secure communication over HTTPS.

Lastly, the correctness and usability of the final product were tested by automatic and user testing. The result showed that the application mostly worked as expected, with the biggest concern being UX issues, which would be the priority during future development. In addition to this, there are possible technical improvements that were skipped during the development due to lower priority, but could be done in the future:

Paging Currently, the data returned by the server is not paged. At this stage of the project, it is not a big issue, however, it will become a problem with a growing number of users. Specifically, a list of posts and trips should use paging.

Error codes Since there are few cases where a logical error can be returned, the implemented server does not have specific HTTP error codes defined for the REST API and generic errors are used instead. Therefore, the client is only able to display that a request failed, without giving a specific reason. With the future growth of the system, more possible errors will appear, and it will be necessary to handle them by returning pre-defined error codes.

Bibliography

[1] Swagger. Best Practices in API Design. [cit. 2022-04-25]. Available from: https://swagger.io/resources/articles/best-practices-in-api-design/

[2] Wickramasinghe, S. AWS vs Azure vs GCP: Comparing The Big 3 Cloud Platforms. Oct 2021. Available from: https://www.bmc.com/blogs/aws-vs-azure-vs-google-cloud-platforms/

[3] Li, A. Android 11 on nearly a quarter of all devices, but it's not the most used version of Google's OS. Nov 2021. Available from: https://9to5google.com/2021/11/22/android-2021-distribution-numbers/

[4] Google. Android Developers. [cit. 2022-04-25]. Available from: https://developer.android.com/

[5] JetBrains. Kotlin Programming Language. [cit. 2022-04-25]. Available from: https://kotlinlang.org/

[6] Gradle. What is Gradle? [cit. 2022-04-25]. Available from: https://docs.gradle.org/current/userguide/what_is_gradle.html

[7] Beams, C. Kotlin Meets Gradle. May 2016. Available from: https://blog.gradle.org/kotlin-meets-gradle

[8] Beryukhov, A. Modularization of Android Applications in 2021. Feb 2021. Available from: https://proandroiddev.com/modularization-of-android-applications-in-2021-a79a590d5e5b

[9] Google. Cloud Firestore. [cit. 2022-04-25]. Available from: https://firebase.google.com/docs/firestore

[10] AWS. Amazon S3. [cit. 2022-04-25]. Available from: https://aws.amazon.com/s3/

[11] Square. Retrofit Introduction. [cit. 2022-04-25]. Available from: https://square.github.io/retrofit/

[12] Google. Firebase Crashlytics. [cit. 2022-04-25]. Available from: https://firebase.google.com/docs/crashlytics

[13] Risberg, T. Spring Framework 1.0 Final Released. Mar 2004. Available from: https://spring.io/blog/2004/03/24/spring-framework-1-0-final-released

[14] AWS. Amazon Cognito. [cit. 2022-04-25]. Available from: https://aws.amazon.com/cognito/

[15] AWS. Amazon RDS. [cit. 2022-04-25]. Available from: https://aws.amazon.com/rds/

Appendix A

Acronyms

API	Application Programming Interface
APK	Android Package
AWS	Amazon Web Services
DI	Dependency Injection
DNS	Domain Name System
DSL	Domain-specific Language
GCP	Google Cloud Platform
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ID	Identification
IDE	Integrated Development Environment
JDBC	Java Database Connectivity
JPA	Jakarta Persistence
JSON	JavaScript Object Notation
JWKS	JSON Web Key Set
JWT	JSON Web Token
MVVM	Model–view–viewmodel
OS	Operating System
RDS	Relational Database Service

REST Representational State Transfer

RSA Rivest–Shamir–Adlema

SDK Software Development Kit

SQL Structured Query Language

SSL Secure Sockets Layer

UI User Interface

UX User Experience

Appendix B

Contents of enclosed CD

`exe`	the directory with executables
`TravelMates-Server-v1.0.jar`	executable of the server
`TravelMates-v1.0.apk`	installation package of the application
`implementation`	source codes of implementation
`travelmates-client`	source code of Android application
`travelmates-server`	source code of the server
`thesis`	the thesis text directory
`thesis-sources`	source code of the thesis
`thesis.pdf`	the thesis text in PDF format