Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Online Organisation of Trips

Bc. Daniel Koten

Supervisor: Ing. Petr Aubrecht, Ph.D.
Field of study: Open Informatics
Subfield: Software Engineering
January 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Koten Daniel**	Personal ID number:	**483828**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

Online Organization of Trips

Master’s thesis title in Czech:

Online organizace výletů

Guidelines:

For planning of a bigger and complex event, there are available multiple simple, freely available solutions, which concentrate only on one part of the problem. Collect all needs of organizers and how they can be solved using current tools.
Design your solution, which will cover the requirements. One user will create an event, invite other users as event members. Together, they work on checklists of items and actions required for the event. Consider reusability of final checklist as a template for future events.
1. Analyze the current state of online tools available for organizing complex group event, specifically checklist of required items and actions. What are the requirements and what approaches the tools use.
2. Design a solution, which will cover the requirements. It must include management of members, shared and private checklists of items and actions, the progress of preparation.
3. Implement your solution. The server-side should use Jakarta EE a PostgreSQL. The client side should provide a mobile application -- select the technology and explain your motivation.
4. Test your application from user's point of view. Implement functional tests -- REST API or UI tests.

Bibliography / sources:

[1] The Jakarta EE 8 Tutorial: https://eclipse-ee4j.github.io/jakartaee-tutorial/toc.html
[2] Jakarta EE Cookbook - Second Edition, https://www.packtpub.com/programming/jakarta-ee-cookbook-second-edition
[3] Patterns of Enterprise Application Architecture — Martin Fowler

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Name and workplace of master's thesis supervisor:
Ing. Petr Aubrecht, Ph.D. Department of Computer Science FEE
Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **28.08.2023** Deadline for master's thesis submission: ___________

Assignment valid until: **16.02.2025**

______________________	______________________	______________________
Ing. Petr Aubrecht, Ph.D.	Head of department's signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature		Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

______ . ______________	______________________________
Date of assignment receipt	Student's signature

Acknowledgements

I would like to thank Ing. Petr Aubrecht, Ph.D., for his guidance, helpfulness, and valuable advice. I would also like to thank all my family, my partner, and friends for their support during my studies and for their help in testing the application.

Declaration

I declare that I have prepared the submitted thesis independently and that I have listed all the information sources used in accordance with the Methodological Guidelines on the observance of ethical principles in the preparation of university final theses.

In Prague, 7. January 2024

....

Abstract

After a long period of restrictions due to the coronavirus, the world has started moving again, and people are looking for an application to organize their events. The purpose of this thesis, which focuses on checklist management and the social aspect, is to cover this area.

The goal is to design and implement a mobile application to give users essential online features to organize an event, focusing mainly on lists of items.

The whole solution should also focus on a cooperative aspect. Because of that, there is also a need for a backend part of the system, which brings many challenges and problems that need to be addressed. For example, the login process, user management, permissions, and performance.

Keywords: Event, Checklist, Flutter, Mobile Application, Jakarta EE, PostgreSQL, Auth0, Docker

Supervisor: Ing. Petr Aubrecht, Ph.D.

Abstrakt

Po dlouhém období spousty omezení v důsledku koronaviru se svět opět dal do pohybu a lidé potřebují aplikaci pro organizaci různých událostí. Záměrem této práce, která se zaměřuje zejména na správu seznamů položek a sociální aspekt, je pokrýt tuto oblast.

Cílem je navrhnout a implementovat mobilní aplikaci, která uživatelům poskytne potřebné online funkce pro organizaci akce, přičemž se bude soustředit především na seznamy položek.

Celé řešení by se mělo zaměřit také na kooperativní aspekt. Z tohoto důvodu je také potřeba vytvořit backendovou část systému, což přináší řadu výzev a problémů, které je třeba vzít v úvahu a řešit, například přihlašování, správu uživatelů, oprávnění a výkon.

Klíčová slova: Událost, Seznam položek, Flutter, Mobilní aplikace, Jakarta EE, PostgreSQL, Auth0, Docker

Překlad názvu: Online organizace výletů

Contents

Figures

Tables

Chapter 1

Introduction

Since the world has started traveling again after the covid lockdown, people have begun organizing trips, especially in groups. Such an event requires a significant amount of preparation and organization, which includes negotiating and agreeing on various aspects. And that is exactly the area I would like to pursue in my work.

A mobile application tailored for this purpose could ease this task. It allows people to cooperate asynchronously and add or improve ideas anytime and anywhere. Either sitting on a couch, walking a dog, or shopping. It is also worth noting that according to statistics, 85.74% of the world's population uses a smartphone.[1] Therefore, I see mobile phones as the right platform to target.

I decided to focus on checklists of items and the cooperative aspect. People need a personal list of items, and they also need to cooperate. Hence, they have to be able to share a part of those lists with other group members. Moreover, it would be valuable to implement a feature that enables users to gain inspiration. It could be either from each other, or the application itself could help by providing some ideas. Finally, it might be notably time-saving to have the ability to reuse past events in some way.

There are already some applications available that focus on this area. However, they do not completely address this specific use case – group checklists management. It should be a clean and easy-to-use application with a clearly defined purpose. Yet, it should provide users with features to support this. Therefore, I tried to stay focused primarily on the area of group checklists, but I also integrated it with multiple supportive features when designing a solution.

The goal of this work is to design and implement an application where users can create checklists for a group event that will provide them with all essential features. The most important one is managing multiple types of checklists, but there are more, like splitting group expenses or automated suggestions on what they should add to their checklists. Users' experience during events organization should be smooth. It should be a complete application to reduce the need to use multiple applications for these purposes, leading to fewer mistakes and contributing to the overall seamless organization of events.

Chapter 2

Related Work

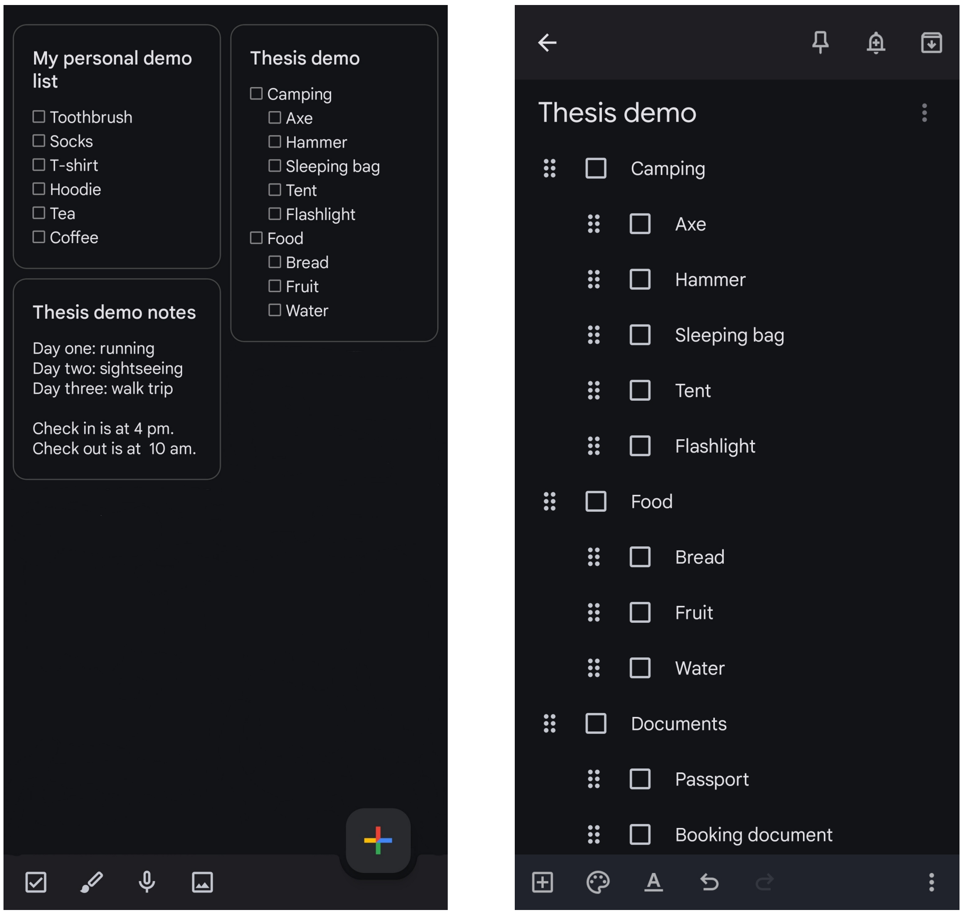
There are already several solutions for planning and preparation of trips and various group events. Despite this, there is still room for improvement and new ideas to make things easier for users.

2.1 Google Keep

One of the most popular applications for managing checklists is Google Keep[4]. It provides basic features like creating a new checklist, adding new items with up to two levels of hierarchy, and sharing with other users. The biggest advantage of this tool is simplicity. It is straightforward to start using it without any previous experience. There is only one screen with a list of notes, and all one has to do is open this note and start adding items, as can be seen in Figure 2.1.

Although, there are many limitations. The most significant one is that there is only one checklist for all users, and another one could be that there is no functionality over this data, such as splitting costs or assigning items to members etc. To summarize, this app is mainly for quick and simple notes and checklists without any advanced functionality.

I would like to take inspiration from this app's clean design. Namely, it uses material design. It follows the best practices for mobile app development and is a kind of standard for most mobile applications.



My personal demo list

- ☐ Toothbrush
- ☐ Socks
- ☐ T-shirt
- ☐ Hoodie
- ☐ Tea
- ☐ Coffee

Thesis demo notes

Day one: running
Day two: sightseeing
Day three: walk trip

Check in is at 4 pm.
Check out is at 10 am.

Thesis demo

- ☐ Camping
  - ☐ Axe
  - ☐ Hammer
  - ☐ Sleeping bag
  - ☐ Tent
  - ☐ Flashlight
- ☐ Food
  - ☐ Bread
  - ☐ Fruit
  - ☐ Water

Thesis demo

- ☐ Camping
  - ☐ Axe
  - ☐ Hammer
  - ☐ Sleeping bag
  - ☐ Tent
  - ☐ Flashlight
- ☐ Food
  - ☐ Bread
  - ☐ Fruit
  - ☐ Water
- ☐ Documents
  - ☐ Passport
  - ☐ Booking document

Figure 2.1: Google Keep

2.2 TripIt

Another solution could be TripIt: Travel Planner[5]. It is a basic app for planning a trip. Users can create an itinerary from predefined types like activities, meetings, restaurants, flights, etc. This can be seen in Figure 2.2. It allows users to add other trip members and, for example, assign them to an activity. This application focuses primarily on a map. Everything revolves around adding activities to the map. In other words, it is an excellent tool for itinerary planning. For instance, it can automatically import trip details from a linked email account. Although it is presented as a good feature, many users are wary of giving access to their email accounts to be scanned by this app for travel details as it could easily raise privacy concerns.[6]

The main problem is that there is no way to manage checklists. It is only possible to add text notes, which is not a satisfactory solution. There are also many limitations to a free version. Users have to pay to use all features.

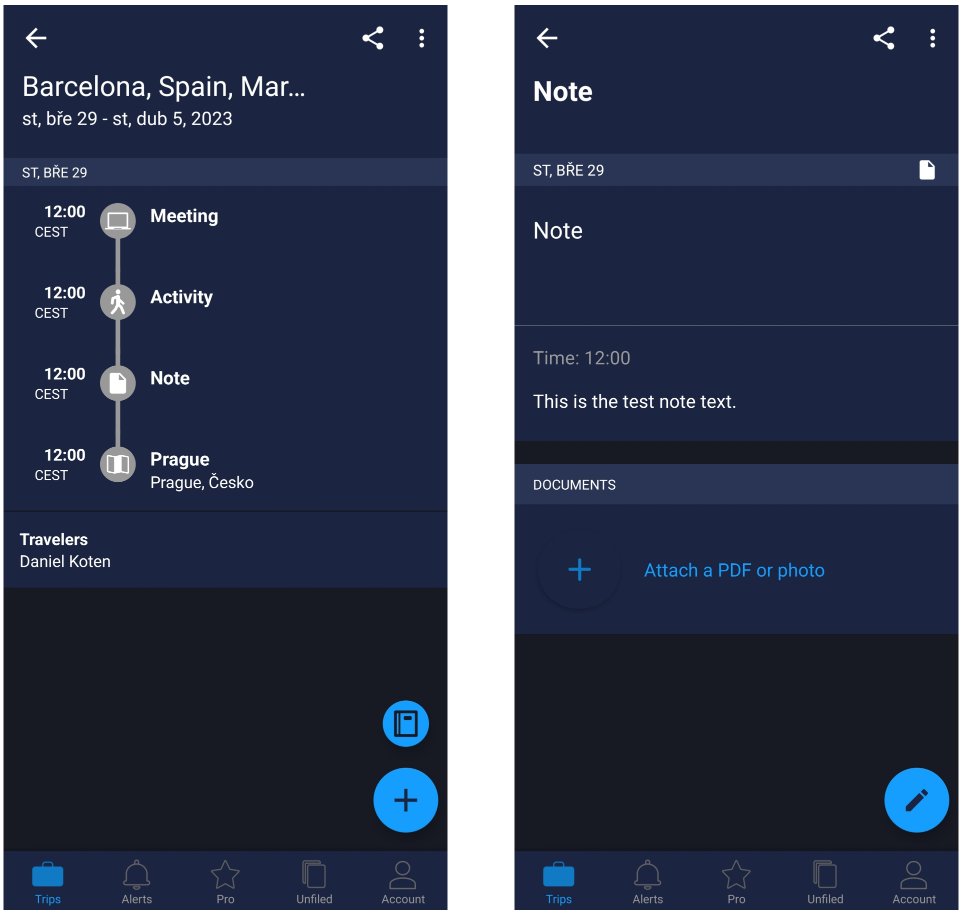


Figure 2.2: TripIt

This tool has good features for building an itinerary but completely misses other functionalities for planning a trip or event. Some examples include checklist management, task assignments, money calculations, and more. I see potential in how this application works with trip members' management. Each has a role (viewer or traveler) and permissions based on this role. I want to implement it similarly in my application.

2.3 Wanderlog

Wanderlog[7] is also an option for organizing events. The biggest problem here is the overwhelming complexity. The UI is full of different controls, and it could be more straightforward to do what the user wants. It also works with checklists in a too-simplified way. It does not allow users to create their private lists or shared ones. There is always just one global. It can be seen in Figure 2.3 where a simple global checklist and expense split screen are shown.

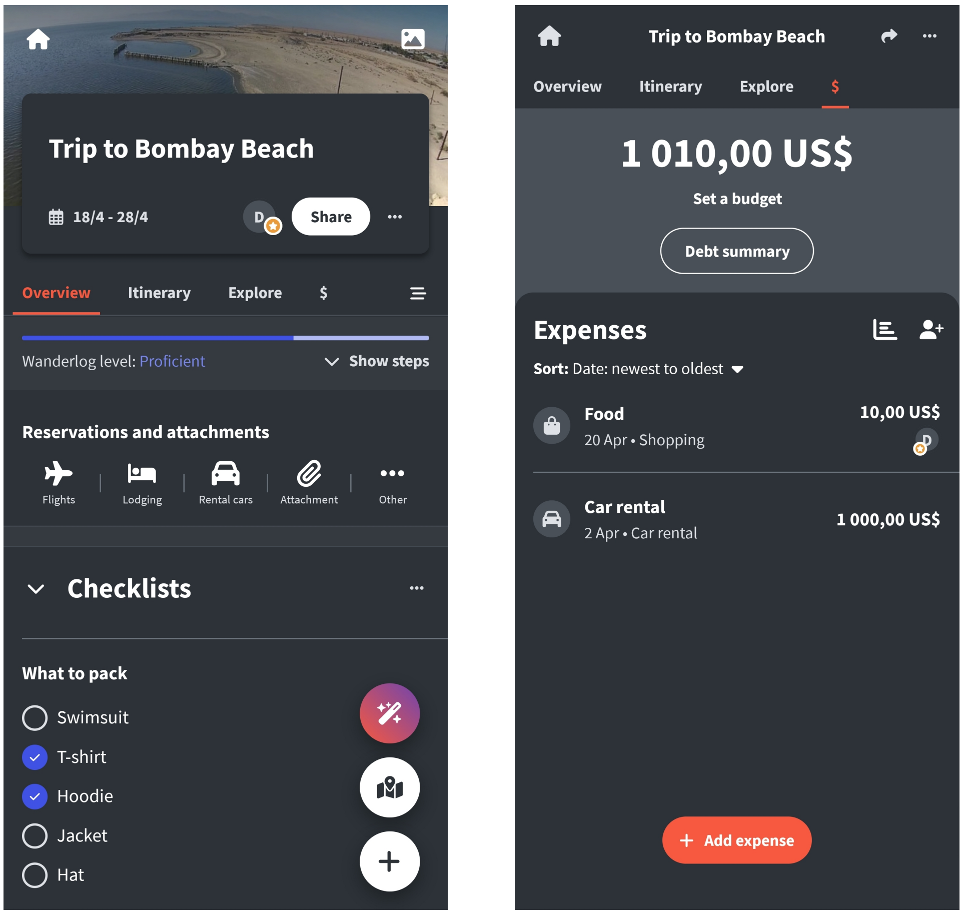


Figure 2.3: Wanderlog

This application is a suitable candidate to take inspiration from, but it needs to be done differently to make it easier for users. One problem is that it does not follow material design recommended practices. It is bad for users because they are used to it, and they expect mobile applications to behave in a similar way. For example, the home page mixes different information. I expect it to contain my trips, but there are many banners with propagation of the PRO version and other elements. The bottom navigation bar should be there most of the time. But it is used only at top-level screens, and most of the time, it is hidden using the app. Another problem is incompleteness. There are functionalities that redirect users to the web page because the app itself does not provide this functionality directly. Such application behavior is not user-friendly and expected.

If done properly, this app could be a good choice for trip and event planning. I would like to take some inspiration from it, especially in terms of budgeting and managing the events themselves – for example, assigning different rights to different members or calculating who owes whom with a minimum number of transactions.

2.4 Summary

Each of those apps offers unique features and insights that can inspire the development of a new app. Google Keep's simplicity and user-friendly design, TripIt's efficient trip member management and itinerary planning, and Wanderlog's potential in budgeting and event management are all aspects that can be incorporated into a new application.

On the other hand, all reviewed applications lack support for more cooperative checklists and functionalities over that data that could help users. They only offer simple global general lists of items where everyone can see and edit everything he wants. This is not sufficient, as it can easily turn into chaos. The aim is to create an app that addresses the limitations of these existing tools while enhancing user experience and functionality in trip and event planning.

Chapter 3

Proposed Solution

Based on the assignment, the analysis of existing applications, and my personal needs, the solution should provide a simple and straightforward way to manage group checklists with additional supportive features and functions over the data.

First of all, it will be a mobile application as that is exactly the suitable platform for this use case. One of the reasons for that is the fact that based on the statistics from Bankmycell [1], there are 6.92 Billion smartphone users in the world – that is 85.74% of the world's population. From Figure 3.1, there can also be seen a growing trend in smartphone ownership, which again supports this decision to make it a mobile app. It can be said that almost everyone has a sufficient phone for this kind of application.

Nevertheless, I decided to provide it as a single-page web application as well. The reason is that there can be situations where it is more comfortable to use the app on a computer. For example, when I work on my laptop, I do not want to take a phone in my hand. It might also be better when it comes to multitasking or copying items.

Cross-platform compatibility can be achieved thanks to Flutter, which I decided to use for the client side. In fact, it supports all the most used platforms, including Android, iOS, Windows, Linux, Mac OS, and Web. The only limitation I have encountered is that if I use a package, it has to have support for the platforms I want to use. Unfortunately, Auth0 SDK does not have full support for some of those platforms yet. I do not see it as a big problem as it runs perfectly in the browser, but if it is needed in the future, it is possible to use another service for authentication to avoid using Auth0 SDK.

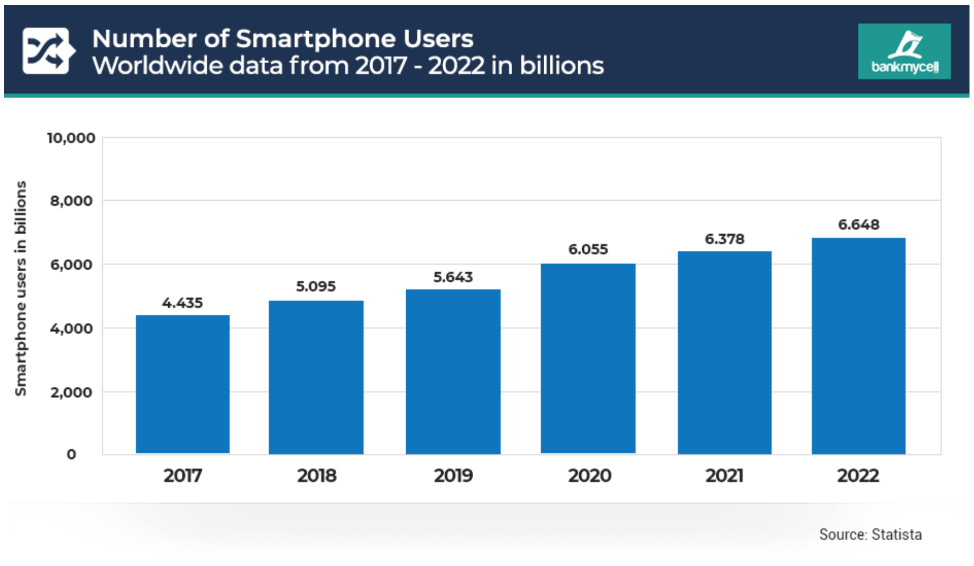


Figure 3.1: How many smartphones are in the world? [1]

The great thing is that it runs the same code completely, thanks to the cross-platform compatibility. This way, it can be accessible from any device equipped with a modern web browser and remains perfectly simple and maintainable.

In terms of functionality, I consider several use cases that I would like to cover and I will describe them in more detail in further paragraphs.

There should be multiple types of checklists as there are more use cases, and it would be nice to integrate them and allow users to use all of those as they need. Namely, there must be a simple personal checklist. This way, a user can use the app without any other members for personal purposes. Although this is not the main focus of this app, there can still be a reason for that. For instance, he can decide later that he wants to invite someone to start participating, and he can easily do it without the need to move all data to another application designed for groups. This leads to the idea that users on that trip should see the checklists of each other as inspiration. Looking at the friend's list and figuring out what I forgot can be extremely time-saving. At the same time, hiding some data from other members should be possible. I can imagine that someone might have some medicine on his list, and he does not want other members to be able to see it. When there are multiple members, it makes sense to have a group-shared list. It will be possible to add items directly to this shared list, but those items will also be visible in the owner's personal list so he can check for all his items, including those that are shared. This way, he has all his stuff in one place, but there is also a place where everyone can see what is shared. It can also be used the other way around – it aggregates all shared items from all members' personal checklists.

It should also be easily recognizable which type of checklist a user currently sees and what items there are. There will be a mechanism to show who is responsible for taking the concrete items, and it will be possible to check the preparation progress.

Considering that it will be possible to add prices to the items, it would be a shame not to use this data for some calculations and to make users' lives easier. Thus, the application will have the functionality to split costs between members. To be concrete, it will also be possible to choose which members are supposed to be involved in the payment. For example, there can be a situation where some part of the group will pay for the rent of a cottage, and the rest of the group will sleep in their tents, so they do not want to pay for the rent. It should also be possible to make changes after some transactions have been done. For instance, someone could add an item later, and the application must recalculate the expenses considering the previous transactions.

The application will also be able to suggest ideas for checklists automatically. It will be possible to generate suggestions based on the current context, such as event name, category, already added items, and eventually more. Adding these generated items to the right category will be very easy with just one click. It will also set the right flags if it is shared or not.

When the user has everything done, he will be able to copy the whole event to reuse it even with a different group. During this duplication, all the personal items will be copied, as well as all shared items from all group members. It is better to copy more items and remove some of them later with just a click instead of manually creating them if they were not copied.

Chapter 4

System Requirements

As the system requirements will serve as a basis for the design of the entire system, I divided them into functional and non-functional to make it more clear.

4.1 Functional Requirements

First of all, users must sign in to use the app. After a discussion with the supervisor, there should be multiple ways of authentication. As it is common to have a Google account nowadays, the app should allow users to log in using third-party identity providers – namely, Google OpenID. If they sign in for the first time, they must fill in the profile information.

Once they log in, they can create a new trip and begin adding items to their checklist. They can also add members to the trip or remove them. Finally, they can show the trip settings where they can edit the trip name and date and delete the whole trip.

There are three types of checklists. Every member has a personal checklist. He has full editing rights to this list. To add each item, he can specify several attributes, namely name, category, price, and flags if the item is private or shared. And that leads to the second type of list – shared list. This serves as a list of all shared items from all members displayed in one place. This is the only place where it is possible to edit or check other members' items. It can be useful when, for example, a family is preparing together, and one member wants to check that he or she already packed an item for the second member.

However, it is not possible to edit or check other members' personal items to avoid unwanted modifications. It could, for example, happen that one user would mark another user's item as packed even if it is not, and he or she will forget to pack it later. It will also be possible to see a specific member's entire checklist as an inspiration for what I could also pack, and that would be the third list type. The limitation is that it will only be possible to see items not marked as private because it is understandable that users might want other members not to be able to see some items like medicines, amount of money, etc.

The app should also work with item categories. It should be as flexible as possible but also user-friendly. To satisfy these two requirements, the user can manually specify a category name for each item if there is no such category. Alternatively, he can choose from categories that already exist in the trip.

Another feature to make adding items as easy as possible is automatic suggestion and generation of items for a given category. Users can generate suggestions of what to pack for a concrete category. It should consider the trip's name, category name, and already packed items. Based on this information, the application should be able to provide other item suggestions that might be relevant to pack up as well. This will speed up the whole process of adding items significantly as users will not always have to write the name of the item and they just click on the suggestion, and it will be automatically added to the correct category. It should be, however, possible to edit item details later – for example, shared or private flags, etc.

There can be a price set for items. It will be possible to split these item expenses among any number of members. Those members will then be able to pay these costs directly from the app, which will open their mobile banking app with all payment information filled in. Speaking primarily about the web version, generating a payment QR code for users to scan using their phones makes sense. When the transaction is done, they can mark it as completed, and it will move to the transaction history section.

FR-1 User Authentication

- **FR-1.1** Users must be able to sign up and sign in either by email and password or by their Google account.
- **FR-1.2** Users must be signed-in to use the app.
- **FR-1.3** Users must be able to sign-out.

FR-2 Trip Management

- **FR-2.1** Users must be able to create a new trip by filling in information like name and date.

FR-2.2 Users must be able to view a trip detail.

FR-2.3 Users must be able to edit trip information if they are in the admin role.

FR-2.4 Users must be able to delete a trip if they are in the admin role.

FR-3 Member Management

FR-3.1 Users must be able to add a new, not yet existing member to the trip by searching by the user's first or last name.

FR-3.2 Users must be able to delete a member from the trip if they are in the admin role.

FR-3.3 Users must be able to leave the trip if they are not in the admin role.

FR-4 Personal Checklist

FR-4.1 Users must be able to view their checklist.

FR-4.2 Users must be able to add new items to their checklist by specifying a name, category, private flag, shared flag, and optionally amount and price.

FR-4.3 Users must be able to edit the items in their checklist.

FR-4.4 Users must be able to delete their items from their checklist.

FR-5 Shared Checklist

FR-5.1 Users must be able to view all items from all trip members marked with a shared flag and not marked with a private flag in a shared checklist.

FR-5.2 Users must be able to add new items to the shared checklist by specifying a name, category, private flag as false, shared flag as true, and optionally, amount and price.

FR-5.3 Users must be able to edit all items in the shared checklist.

FR-5.4 Users must be able to delete all items from the shared checklist.

FR-6 Personal Checklists of Other Members

FR-6.1 Users must be able to view all non-private items in other members' checklists.

FR-7 Categories

FR-7.1 Users must be able to see items grouped in categories.

FR-7.2 Users must be able to choose from the list of already existing categories in a current trip when creating or editing an item.

FR-7.3 Users must be able to add a new category by typing a specific category name when creating or editing an item.

- **FR-7.4** Users must be able to remove a category from a trip by removing a given category from all items in the trip.
- **FR-7.5** Users must be able to add an item directly to a given category.

FR-8 Automatic Suggestions

- **FR-8.1** Users must be able to generate item suggestions for a concrete trip name, category name, and already added items.
- **FR-8.2** Users must be able to edit these items later.

FR-9 Expenses Calculation

- **FR-9.1** Users must be able to add price to the item.
- **FR-9.2** Users must be able to assign multiple members to participate in the payment for the item.
- **FR-9.3** Users must be able to see completed and future transactions of all members.
- **FR-9.4** Users must be able to generate a payment QR code.
- **FR-9.5** Users must be able to open a mobile banking application with preset payment details like bank account number, money amount, and payment description.

4.2 Non-functional Requirements

The estimation is that the app will be used by a small to medium group of people interested in organizing trips. The expected type of users are people of different ages with a positive attitude towards technology and people who like to keep things in order. Based on this assumption, the following can be defined.

The app must be easy to use. It should follow the recommended Material design guidelines, which most people are used to, so it would feel familiar to them. It should also target the fewest steps necessary for a given operation and make it as straightforward as possible so there should be only as much information as is actually needed in each screen. It should also respect the user's settings and preferences. For example, it should respect the theme set in the operating system – dark mode or light mode.

The next thing is performance. The goal is to handle hundreds of simultaneously active users. Each of them should be able to work with hundreds of trips and items. Considering that one of the primary purposes is to cooperate on the trip in a group, the server should be stable and reliable because the app will depend heavily on it as the data will have to be synced. However, no computationally intensive operations are expected to be done on the server.

It is critical to have data consistent across the whole system. Whenever a user makes a write operation, it must be persisted in a database immediately to prevent data loss. At the same time, every user should always see up-to-date data. So it is necessary to implement some refresh mechanism at the client.

Another just as important point is security. Users must log in to use the app. And then, no one can modify their data without permission. A server must check every API request to see if it is authorized and if it accesses the resources the user can access. On top of that, the system must handle user management securely. In particular, passwords must be stored appropriately to avoid leaks. Namely, they should be hashed and salted in the database. Finally, the app should also protect users against themselves, especially when it comes to unintended actions like delete operations. Those should always be protected by a confirmation dialog.

Maintainability is definitely part of the must-have non-functional requirements as well. Implementing and deploying new versions of the client app and back-end code must be possible easily and efficiently without data loss. It is not only to add new features but also to fix bugs and to apply security patches if there are any new vulnerabilities in the libraries used.

It must also be possible to run the application on more platforms with a single code base. Namely, it should run as an Android app and also as a Web application. The biggest advantage of it running as a web app is the fact that it can be run by any user on almost any device, like a phone, tablet, PC, etc. On top of that, it should be possible to easily add support for more platforms in the future, namely for Apple iOS devices.

NR-1 Ease of use

- **NR-1.1** Application must be self-explanatory even without documentation.
- **NR-1.2** Application design must follow Material design best practices.
- **NR-1.3** Application must target the fewest number of steps necessary for a given operation.

NR-2 Performance

- **NR-2.1** Application must be able to handle hundreds of trips for a given user and hundreds of items in each trip.
- **NR-2.2** System must be able to handle hundreds of simultaneously active users.

NR-3 Data Consistency

- **NR-3.1** Data must be consistent across users.
- **NR-3.2** Data must be written to the database after the change action on the client, and the current data must be updated on the other clients after the reload.

NR-4 Security

- **NR-4.1** Only a properly logged-in user can interact with the system.
- **NR-4.2** The system must check that the user has rights to the requested resources.
- **NR-4.3** User management must be handled in a secure manner so that, for example, passwords cannot be leaked.

NR-5 Maintainability

- **NR-5.1** The whole system must be efficiently maintainable.
- **NR-5.2** New version deployment must be automated.
- **NR-5.3** Database update must be automated and without data loss.

NR-6 Multi-platform

- **NR-6.1** It must be possible to run the application on Android OS.
- **NR-6.2** It must be possible to run the application as a web application.

NR-7 Compatibility

- **NR-7.1** Application must be compatible with the last two major versions of Android OS.
- **NR-7.2** Web application must be compatible with the last two major versions of Google Chrome browser.

NR-8 Unintended Actions Protection

- **NR-8.1** Application must protect the user against unintended actions, especially when deleting items, for example, with a confirmation dialog.

Chapter 5

Technology Stack

There were some technologies like Jakarta EE 8 and PostgreSQL given in the assignment. When I was selecting the remaining technologies, I chose them to be as fit for purpose as possible. Another very important factor was how well-documented the technologies were.

5.1 Application Server

After discussing with the supervisor, I chose the Payara server[8] as an application server. It is a stable, well-tested solution for production-ready systems and offers a free trier.

One of the reasons was also the fact that it has excellent support for Auth0 integration. It is possible to set up the complete integration using only annotations and a configuration file. An official docker image is also available for the free tier.

The interesting thing was the migration from Jakarta EE 8 to Jakarta EE 10. Based on the assignment, I started with version 8 which was sufficient at that moment but during the development, there was a need to migrate data types for IDs in the database to UUIDs and it was not possible in that version to make them auto-generated. After a discussion with a supervisor, we decided to upgrade the Jakarta EE version. This is also related to the Payara version which was also upgraded from version 5.2022.5-jdk11 to 6.2023.11-jdk17. It can be noticed that the Java version was upgraded from version 11 to version 17 as well. This was exactly the point where the usage of Docker helped a lot. The upgrade of the Payara server was as easy as changing a line with a version in the Dockerfile because all server configuration is done using CLI so there is no need to make anything manually. As for the migration of the Jakarta version, I used the following script which took care of the whole process.

```
mvn package fish.payara.transformer:fish.payara.transformer.
    maven:0.2.11:run -DselectedSource=src -DselectedTarget=src10
```

Listing 5.1: Jakarta EE 8 to Jakarta EE 10 upgrade script

5.2 PostgreSQL

I was supposed to use a PostgreSQL database. It is open-source, well-tested, and stable, which is crucial for a database. *"PostgreSQL has been fully ACID-compliant since 2001 and implements multiversion concurrency control to ensure that data remains consistent."*[9] It may be worth describing the ACID in a bit more detail. Those letters stand for Atomicity (all operation must be completed successfully or everything is rollbacked), Consistency (a database must be in a consistent state both before and after a transaction), Isolation (result of a transaction is visible to other transactions after it has been committed) and Durability (a result of a committed transaction must be saved permanently) and those properties are achieved thanks to the transactions which is a tool to ensure them as Martin Fowler describes in his book. [10]

5.3 Flutter

The fundamental decision was about the technology for implementing the client application. When I was reading various texts [11], I found that Flutter is considered a very good framework according to feedback. Namely, it is supposed to be a very well-designed, robust framework for developing multi-platform applications with a reasonably close-to-native performance. According to the official documentation, *"Flutter is an open source framework by Google for building beautiful, natively compiled, multi-platform applications from a single codebase."*[12]

Currently, I am targeting Android devices and it can even run in a browser as a web application. But it is possible to make it run as a native application even on iOS devices, Windows, Mac OS, and Linux in the future with minimal modifications. Although Flutter supports all these platforms, it is worth noting that the limiting factor might be the packages used. It is important to check that the packages also support those platforms. I have come across that the Auth0 package only supports Android, iOS, and the web. It is not a limiting factor for me as the web support currently meets my needs, as it allows me to reach the vast majority of potential users. However, there could be challenges if there is a specific demand for example for a native Linux application, which is not currently supported. If it were desired to target these platforms as well, it would be necessary to use packages that allow this.

Another great thing about Flutter is that it uses a reactive programming paradigm. That helps in the separation of concerns and hence cleanliness and maintainability of the code. It also makes such a project very easy to scale.

In terms of performance, it is very close to the real platform native code. It is because the rendering is not done using the translation of the components to the native components but they are rendered directly on the canvas as Eric Windmill mentions in his book. *"Flutter compiles directly to native code and uses Skia, the same graphics engine that Chrome relies on, for rendering. This eliminates the need for translating Dart code at runtime, ensuring that applications maintain optimal performance and efficiency on a user's device."*[13]

I considered also other options.

If only performance is a factor, pure native would be a better choice. But in this case, the benefits of cross-platform development are more likely to prevail. It should also be added that if some part of the application needs to be written in native, for example, to access sensors, it is possible to integrate this native part into the Flutter application.

Another candidate that I considered was React Native. This is also a great choice for the development of cross-platform mobile applications. One of the helpful facts is that there are really many packages for this technology and it is very well established. Despite these advantages, I decided to give Flutter a chance. Although it is still a new framework and not so widespread yet, it offers very impressive statistics in terms of both performance and multi-platform support. Some studies show that Flutter can outperform React Native by tens of percent in terms of both CPU and memory usage. [14] Another reason was the fact that I find it more comfortable to write code in Dart rather than in JavaScript or TypeScript.

5.4 Auth0

I decided to use Auth0 for authentication and authorization. It is a popular solution that handles user management very well, is flexible, and allows multiple login methods. The basic one is using a username and password. Logging in using various third-party providers such as Google, Facebook, and so on is also possible.

It is possible to solve all this from scratch. Still, this solution is preferable, especially because, as with libraries and frameworks, it is usually better to use existing tested solutions rather than writing a custom framework.

There are indeed other solutions. One of the highly used is Keycloak which is an open-source identity and access management solution. [15] I choose Auth0 over Keycloak mainly because Auth0 has way better integration support for Flutter. At the moment, it is also better for me to use the free tier on Auth0 than to pay for custom resources to run a custom Keycloak instance. Another candidate could have been Google's Firebase Authentication [16] which has excellent support for Flutter. Despite the slightly more complex setup, I decided to go with Auth0 because of its greater flexibility and customization options. For instance, I used Auth0 Actions to add user roles directly into the ID token. This approach is efficiently handled on my server side, where Payara offers excellent support. It allows for resource restrictions based on the roles specified in the ID token, enabling a more refined and secure user management system.

Chapter 6

Design and Architecture

This chapter revolves around the design of the whole system and its individual parts. It is also about putting those technologies defined in Chapter 5 together even with other services like reverse proxy, etc. During the design of the whole solution, the previously defined functional and especially non-functional requirements were taken into account as they directly influence some decisions, such as the choice of the technologies used.

6.1 System Architecture

I chose client-server[17] as the basis for the whole architecture. The simplified deployment diagram can be seen in Figure 6.1. This decision is based mainly on the NR-2.2 requirement that the system must handle a load of several hundred users. It also has a clear separation of concerns between client applications and backend logic, which contributes to the overall maintainability.

The server environment is orchestrated using Docker Compose, which allows for easier deployment and re-usability of the containers. In terms of those containers, there is a Nginx server in the role of a reverse proxy. This server proxies requests to the Payara server running my application. According to the assignment and the NR-3.1 requirement that clients must have consistent data with each other, I used a PostgreSQL database that meets the ACID properties for this purpose. To manage database schema changes during the development, I utilized Liquibase which enables version-controlled changes – so-called migrations.

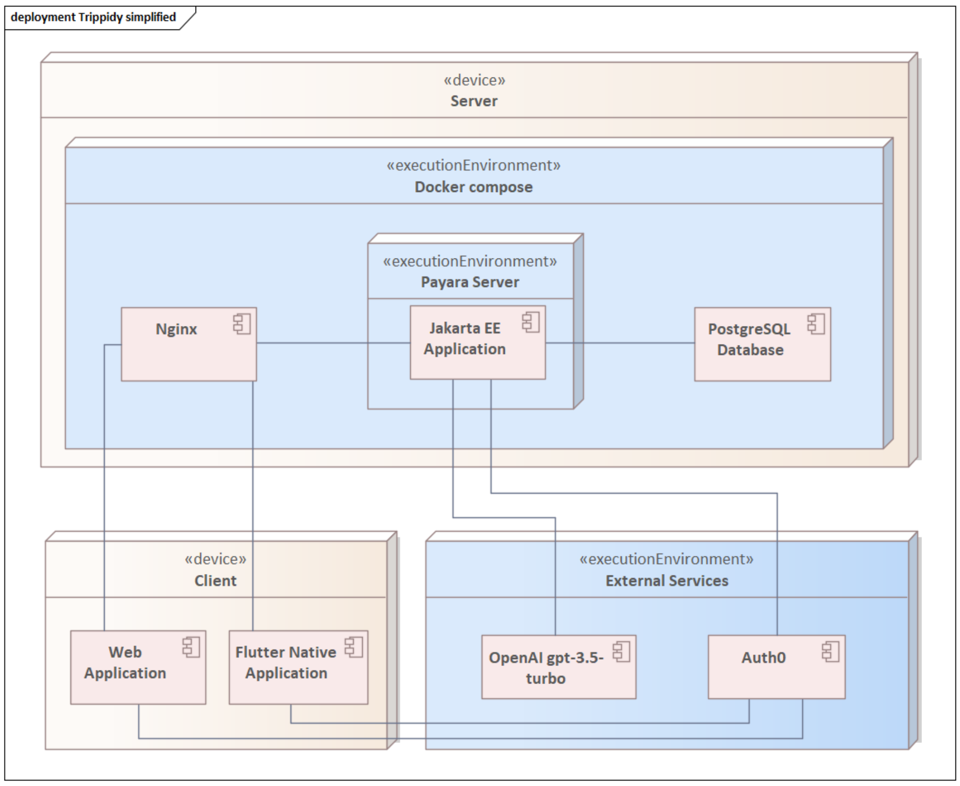


Figure 6.1: Deployment diagram – overview

On the client side, there can be either a Flutter native application or a web application running in the browser. This way, the system can be accessed from almost any user device.

As for the external services, there is Auth0 for the authentication and authorization process. It is communicating with both the client and server. Moreover, OpenAI gpt-3.5-turbo service is utilized. It is used by the server for automatic item suggestions. It must not be called directly from the client application because it would be insecure to store API keys directly on the client as it could be decompiled and those keys could be stolen.

There are also other services in this system. I do not show them here yet, but I will show them in the deployment section to keep this architecture overview simple.

6.2 Database Design

A database schema is a very important aspect of the system and can often influence how the architecture should look like. That is why I decided to describe it right in the second section of this chapter. Its schema can be seen in Figure 6.2.

The central point is the Member entity. Essentially, it is a combination of a concrete user profile and a concrete trip. It also has a role that specifies what actions the user can take on the current trip – for instance, delete the trip and so on. There is also a flag accepted. It specifies if the user has already accepted the invitation and can therefore access the trip or if it is still an invitation and the user can not see the content of the trip yet. Each member can also have multiple items, completed transactions, and future transactions. The usage of an Item entity is obvious. The completed Transaction entity is used to save who has already paid how much to whom. As for the Future Transaction entity, it stores the information about which members should participate in paying for the concrete items – because, according to the FR-9.2, the application should allow users to specify only a subset of members to pay for the item. Finally, there is a Category entity that allows effective work with item categories and their reusability.

6.3 Dockerized Backend

I decided to use Docker for the backend part of the system as it brings a lot of advantages, such as isolation of individual services, independence from the system where the service runs, and more. In simple words, it is sufficient to have Docker and Docker Compose installed on the computer, and then running the whole backend infrastructure is as easy as running a simple command. The following containers are currently running.

6.3.1 Server

The central unit is the server. Originally, I chose the Jakarta EE 8 server as required by the assignment, but during the development, I had to upgrade the version to Jakarta EE 10 – it was, of course, after the discussion with the supervisor. As a concrete application server implementation, I choose the Payara server as described in Chapter 5 in more detail. It is a very important part as it communicates with a majority of other services.

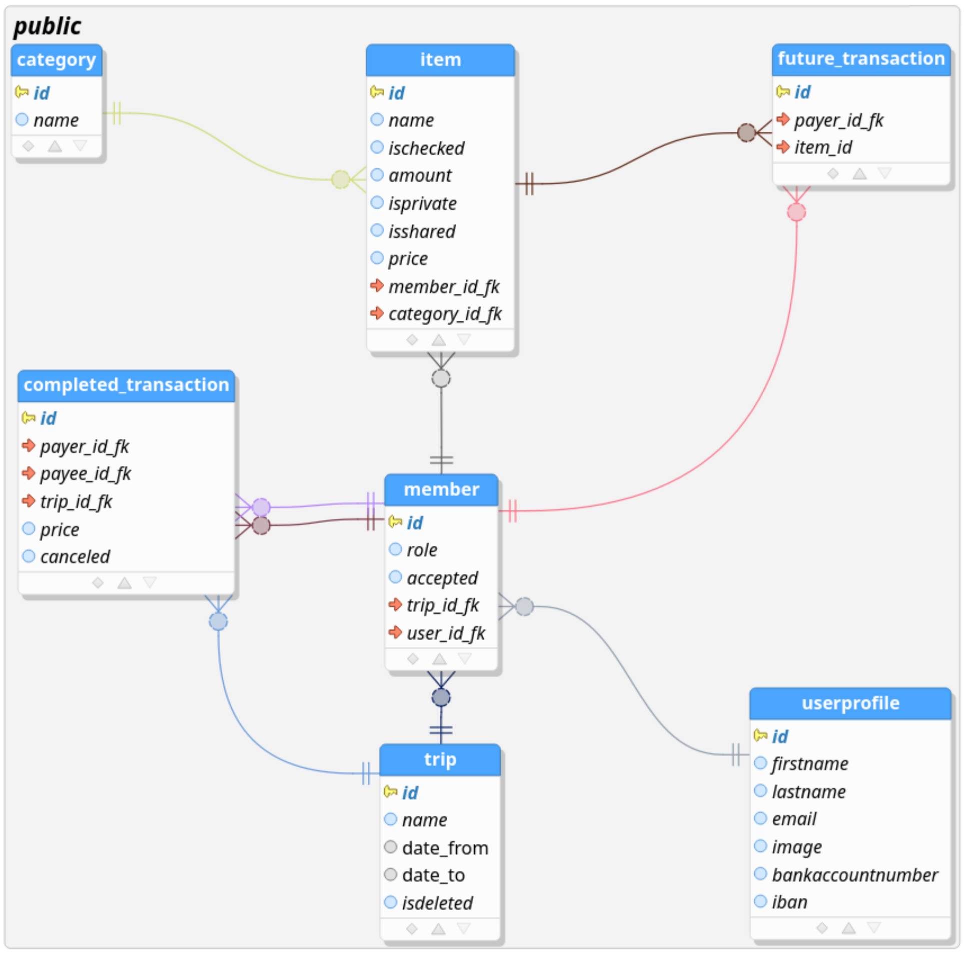


Figure 6.2: Database schema

6.3.2 Database

According to the assignment, I chose the PostgreSQL database for data storage. It also exactly fits the data consistency and security requirements as per the requirements of NR-3 and NR-4.

As for the interaction with the database, it is not directly exposed to the internet. It only communicates with a server and a Liquibase service within the local network. This is the best practice to minimize potential vulnerabilities and provide additional security.

6.3.3 Liquibase

As Martin Fowler describes in his book Refactoring: Improving the Design of Existing Code [18], the database structure needs to be maintained as the project evolves. It is definitely a challenging task, especially in a production environment where any data loss would cause a huge problem. The issue is so important that Fowler describes it even in more detail on concrete scenarios in his article Evolutionary Database Design.[19], which is why I incorporated Liquibase into my system as well.

Liquibase is a service that manages so-called database migrations. Those are incremental changes to both the database schema and content that are applied to keep the database schema updated and compatible with the current code. Thanks to this, the entire system, including the database, can be further modified without the need to manually make the necessary changes to the database. This brings the way better maintainability and work efficiency. It also allows us to make changes in the database without data loss when used properly. Finally, all this is fully automated and works perfectly as a part of the continuous integration & continuous deployment process, which will be described later in more detail.

In the context of the Docker Compose, this service is different from other services as it does not run all the time. It is only ever started once, for example, when a new version is deployed, and its purpose is to bring the database state up to the required state through the migrations, whether it is modifying the database schema itself or manipulating data.

6.3.4 Nginx

When dealing with any data that might be perceived as private, it is necessary to use encrypted communication. In my system, I work with many types of private data like authentication tokens, user items, and more. That is why I used TLS for communication. According to NIST, TLS is *“a security protocol providing privacy and data integrity between two communicating applications.”*[20]

Setting TLS communication directly on the Payara server would be possible, but I decided to go differently. I utilized Nginx as a reverse proxy server, which has a TLS certificate and provides secure connections to clients. It proxies all requests matching the specified pattern to the Payara server. The important thing is that the communication between Nginx and Payara does not have to be secured anymore because it is running in the same virtual environment, and the requests do not travel over the public internet. There is also a redirect from HTTP to HTTPS protocol, so users are forced to use the secured connection.

Another purpose of this reverse proxy server is to add CORS headers to the responses of external services called from the clients. Namely, there was a problem that a web browser running the web version of the application blocked a load of images from external websites in some cases because those websites returned the wrong CORS headers in a response. Therefore, I decided to fetch all images from external services through the Nginx reverse proxy.

Last but not least, this brings another advantage, and it is caching. Namely, I set caching for all requests for images as those are seldom changed, improving response times and the whole system's performance. Those resources are cached for 1 hour or until they take up over 1 GB.

Finally, this server can also work as a load balancer in the future if more performance is needed. However, it is worth noting that this would bring even more complexity as it would bring a need for data synchronization across various servers after write operations.

6.3.5 Certbot

In the previous subsection 6.3.4, I mentioned that the Nginx server has a TLS certificate so it can provide secured communication. The problem here is the fact that the certificate from the Let's Encrypt certification authority is valid only for 90 days. It is, therefore, necessary to renew it before its expiration. For sure, it would be possible to do it manually, but it would not be very wise or effective.

That is why I employed the Certbot. According to official documentation, *"Certbot is a free, open-source software tool for automatically using Let's Encrypt certificates on manually-administrated websites to enable HTTPS."*[21] In other words, it automatically renews the certificate every 60 days.

6.4 External Services

In addition to the actual services running in the docker, the system also interacts with external services that already exist for the purpose. The majority of them are called from the backend, but for example, Auth0 is called even from the client.

6.4.1 Auth0

As the NR-4 security requirement implies, user login and related API security need to be addressed accordingly. For these purposes, I decided to integrate the Auth0 solution, which addresses this area, into the system. According to the official documentation, *"Auth0 is a flexible, drop-in solution to add authentication and authorization services to your applications."*[22]

I used this product because it provides a robust and secure user authentication and authorization solution. It is also possible to use many third-party identity providers with Auth0 – for example, Google, which I used. Another great feature is the so-called Actions. They allow for custom JavaScript code to be run in various situations like user login workflow and many more. As a result, it is possible to call custom triggers, modify the response token, etc. In terms of my usage, I utilized this to add user roles and claims required by the Payara server. The custom Actions code can be seen in the Listing 6.1.

```
exports.onExecutePostLogin = async (event, api) => {
  const roles = ['user'];

  if (roles) {
    api.idToken.setCustomClaim('https://payara.fish/mp-jwt/
   groups', roles);
    api.idToken.setCustomClaim('https://payara.fish/mp-jwt/jti',
     require('uuid').v4());
  }
};
```

Listing 6.1: Auth0 Actions script

On the server side, each request from the client app must contain a token signed by Auth0. This is used to authenticate the user and based on the roles specified in it, the user can be then either authorized to do a given action or not. However, the token signature must be verified. A configuration file can be seen in the Listing 6.2. It is used by the Payara server to verify the signature of the token. If it is valid, it means that the user provided the correct credentials to Auth0 and obtained this valid token from it. In this case, the token is used for user authentication and authorization.

```
mp.jwt.verify.publickey.location=https://trippidy.eu.auth0.com/.
    well-known/jwks.json
mp.jwt.verify.issuer=https://trippidy.eu.auth0.com/
```

Listing 6.2: Auth0 server configuration for token signature verification

6.4.2 OpenAI GPT API

There has been a huge demand for AI features recently due to its revolutionary capabilities in natural language processing, decision-making, and much more. That is one of the reasons why I decided to integrate OpenAI's GPT-3.5-turbo into my application.

I choose specifically the GPT-3.5-turbo model for its cutting-edge performance in reading and generating texts in natural language combined with reasonable pricing and speed. There are other even more performant models like GPT-4, but they are slower and more expensive. In addition, I do not need its even bigger context size and other capabilities. For my use-case of generating item suggestions based on the context, it is sufficient to use a bit less advanced GPT-3.5-turbo as it, on the other hand, offers more reasonable pricing and faster response times, which is very important from the user experience perspective. Users usually do not want to wait tens of seconds to get a response if it is just a list of items.

As shown in Figure 6.1, the GTP-3.5-turbo is called from the server and not directly from the client applications. It has several reasons. One of them has already been mentioned, and it is security because if it were called from the client app, the API key would have to be stored directly on the client, and it could be stolen. But there are other reasons. For instance, daily limits can be set for each user to prevent overuse. In a production scenario, this feature would have to be limited so that the application is not in the red financially. Speaking about financial sustainability, the solution could be that users would get some free points to use this feature for each advertisement watched.

Another reason to call it from the server is logging. It is always a good idea to log at least the errors and exceptions so the problems that were not discovered during testing can be identified and fixed. Additionally, this way of using the API through the custom server can prevent various kinds of misuse. There could be users trying to generate inappropriate content, and if it were called directly from the client, there would be no way to filter this.

6.4.3 Let's Encrypt

When it comes to encrypted communication, there is a need for a certificate from the Certification authority to verify that the client communicates with the correct server and not with the potential attacker. I choose Let's Encrypt CA for this purpose. According to the official site, *"Let's Encrypt is a free, automated, and open certificate authority (CA), run for the public's benefit. It is a service provided by the Internet Security Research Group (ISRG)."*

To sum up, it offers a fully automatic process of both obtaining new certificates and renewals, and all of this is provided for free.

6.5 Frontend

Based on the reasons mentioned in Chapter 5, I used Flutter for the client applications. In addition to Flutter itself, I also used several packages mentioned below.

6.5.1 Flutter Riverpod 2.0

According to the official documentation, *"Riverpod is a reactive caching and data-binding framework that was born as an evolution of the Provider package."*[23] I use it for state management in the whole application, and it brings many benefits. The most important one is that it perfectly works with reactive programming, forcing the programmer to write clean and maintainable code.

6.5.2 Auth0 SDK

It would be possible to use Auth0 just using HTTP requests, but it would be unnecessarily complicated. Instead, I decided to use Auth0 SDK for Flutter, which provides all necessary functionalities – most importantly, WebAuthentication. One of the significant advantages of using such an SDK is that it handles key exchange and other cryptography issues. It can be seen in Figure 6.3 how the whole process works. The important thing here is that no secret key needs to be kept on the client because it is not safe to store it

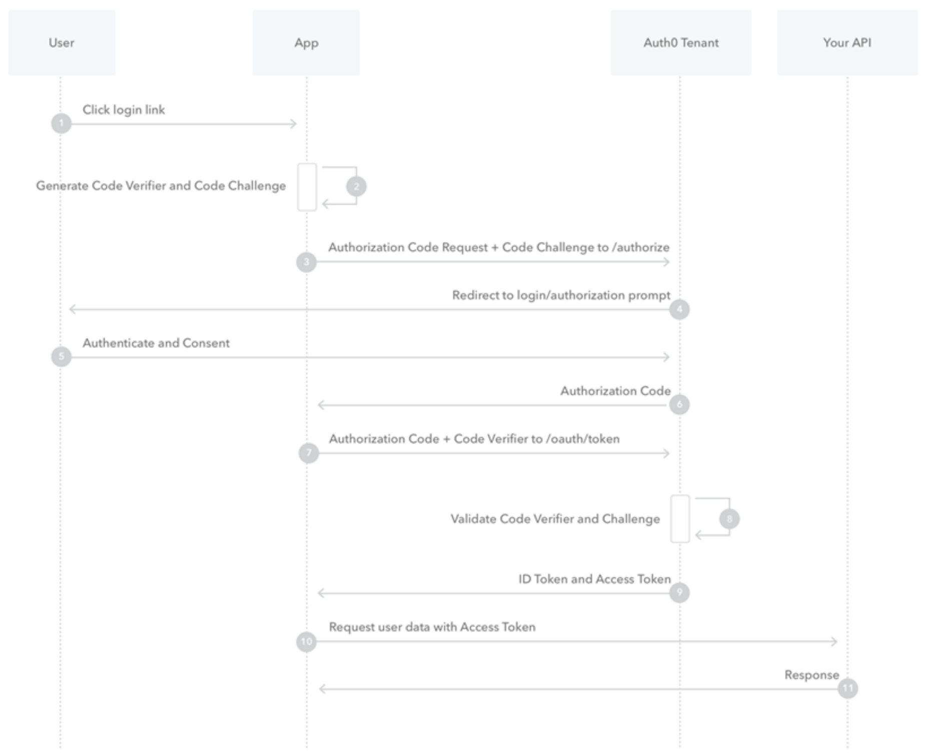

User
App
Auth0 Tenant
Your API
Click login link
Generate Code Verifier and Code Challenge
Authorization Code Request + Code Challenge to /authorize
Redirect to login/authorization prompt
Authenticate and Consent
Authorization Code
Authorization Code + Code Verifier to /oauth/token
Validate Code Verifier and Challenge
ID Token and Access Token
Request user data with Access Token
Response

Figure 6.3: Authorization Code Flow with Proof Key for Code Exchange (PKCE) – retrieved from the official documentation [2]

on the client's device – for example, there is a possible risk of decompiling the application and misusing it.

6.6 Mobile Application Design

I first created a low-fidelity prototype of the screens shown in Figure 6.4 to determine how the mobile app works. Of course, the final application looks a bit different, but it is expected. This low-fidelity prototype's purpose is to sketch, very generally, how the application will work, how the screens and components on them will be roughly arranged, and how the features defined in Chapter 4 will be implemented.

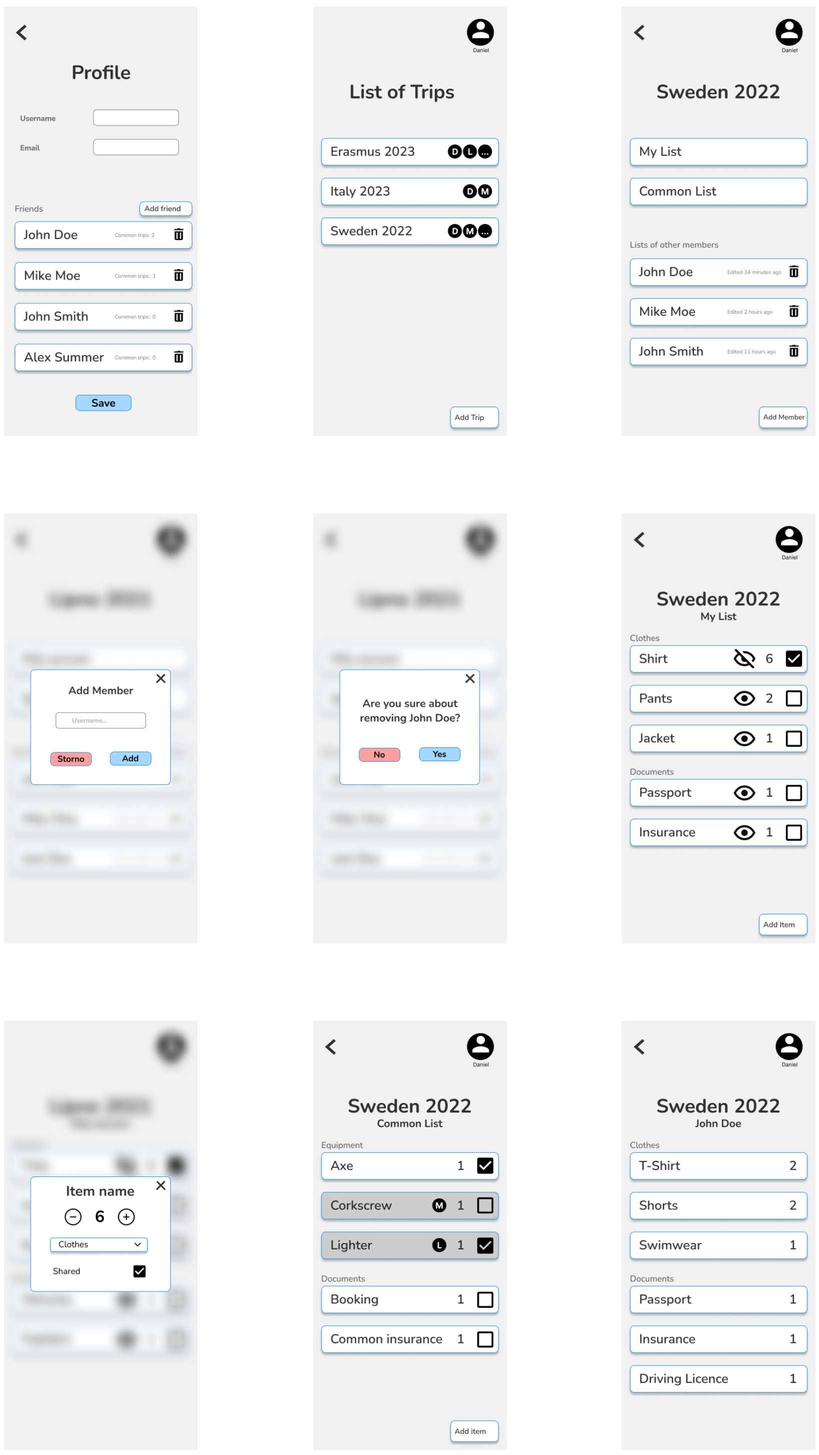


Figure 6.4: Low fidelity prototype screens

6.7 Integration and Deployment

I also set up so-called continuous integration & continuous delivery. Rossel nicely describes this concept in his book simply as an automation.

"If Continuous Integration, Delivery, and Deployment could be summarized with one word, it would be Automation. All three practices are about automating the process of testing and deploying, minimizing (or completely eliminating) the need for human intervention, minimizing the risk of errors, and making building and deploying software easier up to the point where every developer in the team can do it (so you can still release your software when that one developer is on vacation or crashes into a tree). Automation, automation, automation, automation..." [24]

6.7.1 Jenkins

This tool was often recommended during the lectures in software engineering-related courses, and it has its reasons. It is a very robust and powerful build server that can be even expanded using various plugins. Namely, I used a plugin for GitHub integration, allowing me to seamlessly trigger a build based on the GitHub notifications.

Specifically, using the Jenkins server, it is possible to automatically build a new version and then deploy it to a test environment, making the development very efficient. It is unnecessary to run this whole process manually as it is started automatically whenever a new version is pushed to the master branch in git. When it happens, GitHub automatically notifies the Jenkins build server and it pulls the new changes from the remote repository, builds the whole solution, and deploys it to the testing environment.

The problem that I encountered was the situation that I wanted to run Jenkins in Docker as well as the rest of the system. The issue here is that Jenkins needs to call Docker commands on the host machine to deploy a new version of the Docker containers, but Jenkins itself is running as an isolated Docker container. Hence, it does not have access to the host machine.

In general, there are two main solutions to this problem. One of them is to run Docker-in-Docker. However, this might cause unexpected problems. Therefore, I decided to go another way. It means setting up an SSH server on the host and then connecting to it from within the Jenkins running as a Docker container. I found this method in the article by Abhishek Attri. [25]

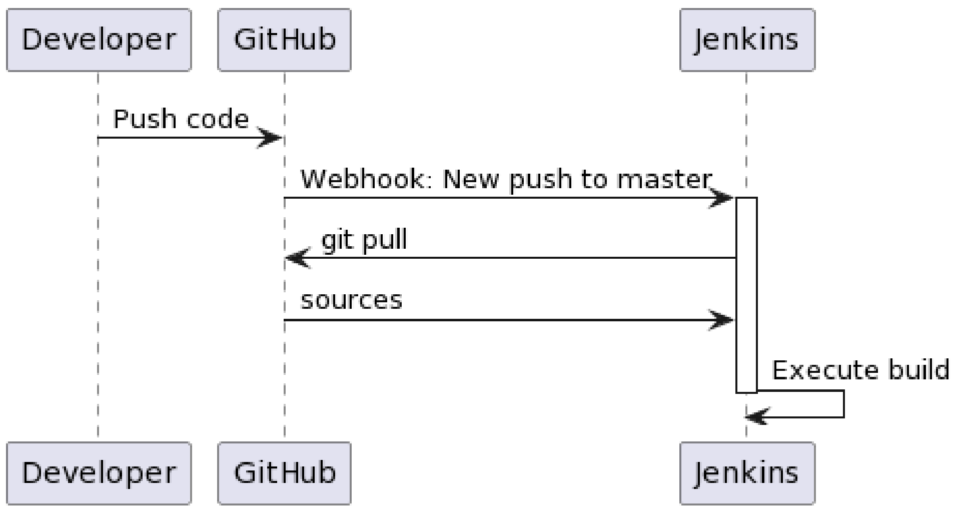


Figure 6.5: Jenkins build using GitHub Webhooks - sequential diagram

This SSH server and communication could also be depicted in the deployment diagram of the whole architecture shown in Figure 6.6, but I did not want to over-complicate this diagram for readability reasons, so I omitted this fact in that diagram.

6.7.2 GitHub

It is also worth mentioning that I utilized GitHub Webhooks to automatically notify the Jenkins build server whenever a new version is pushed to the master branch so it can immediately start the deployment process. This is faster and way more effective than if Jenkins would have to ask GitHub whether there is anything new periodically. The process can be seen in a sequential diagram in Figure 6.5.

6.8 Complete System Overview

For completeness, I show the extended deployment diagram in this section – Figure 6.6. Compared to the simplified one, Figure 6.1, shown in the first section of this Chapter, this extended one also contains the services responsible for this CI/CD and even services like Certbot, etc., described in previous sections. This complete diagram can serve as a whole system overview as it contains all important services and how they are connected.

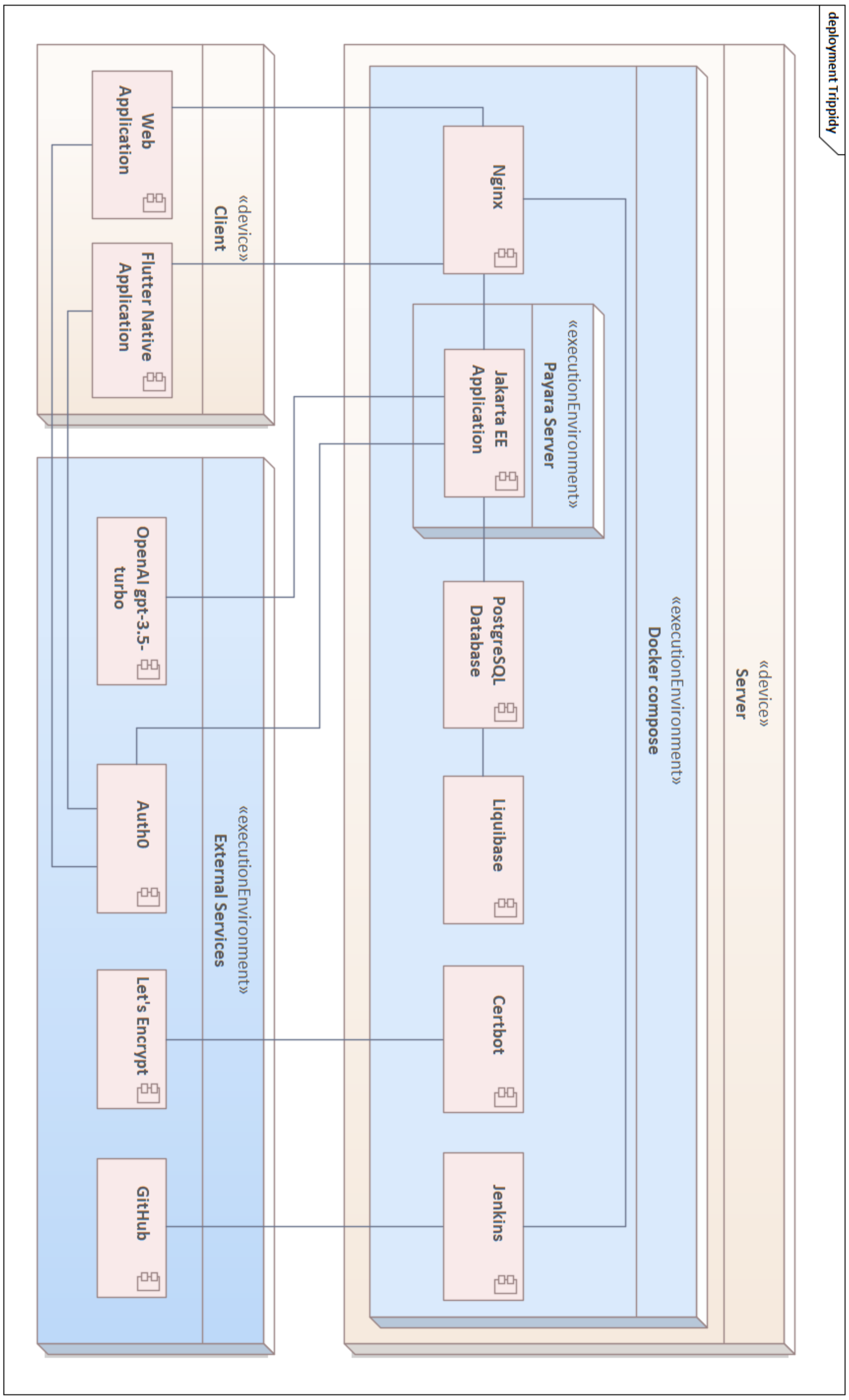


Figure 6.6: Deployment diagram

Chapter 7

Implementation

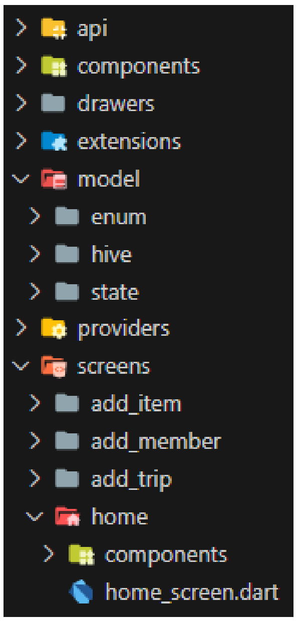
There are a couple of components of the whole system that deserve to be described in a bit more detail. Those are frontend and backend implementations and also the final application.

7.1 Mobile Application Implementation

The common problem with frontend frameworks is that there are usually no strict project structure guidelines, which may lead to so-called spaghetti code. Of course, there are many different recommendations, but ultimately, it is up to the developer to choose the appropriate way. So I decided in the end, after analyzing different ways, to stick to the project structure shown in Figure 7.1. The sample is very simplified for demonstration purposes only. It is based on the *“package by layer”*[26] approach, which is usually appropriate for medium size projects.

In terms of a model layer, I decided to use the same model as is used by the server DTO model. It greatly improves the process of integration with the backend. When the application starts, it downloads all necessary data, and then this data model is changed only by partial updates to avoid unnecessarily wasting internet resources. This also makes further requests faster, greatly improving user experience. In some cases, additional requests are not even needed.

Everything is a widget (a building block of the UI) in Flutter, and these widgets are combined into more complex ones, resulting in a widget tree. As



```
> api
> components
> drawers
> extensions
v model
  > enum
  > hive
  > state
> providers
v screens
  > add_item
  > add_member
  > add_trip
  v home
    > components
      home_screen.dart
```

Figure 7.1: Client project structure (simplified for demonstration purposes)

Flutter uses the reactive programming paradigm, fields in those widgets are reactively updated based on the underlying state objects. This separation of concerns makes the whole application more maintainable and scalable.

As already mentioned in Chapter 6, I decided to use Riverpod for state management. In terms of usage of this package, I simply create a provider class for each important entity in the application – a good practice is one provider per screen. This provider holds the state of a given type. It can be, for example, a list of trips. In addition, this provider also contains code to work with this data.

To give an example, there is a login process in the app. Once the user successfully logs in, the trips provider gets notified and loads the trips for the current user from the server. While those trips are loaded, a progress circle is shown, and once they are loaded, a screen (view) is notified and rebuilds itself. To sum up, there is no code to load data from the server in the view. Instead, it watches for changes in the provider and rebuilds on any data change.

As for the integration with my backend, I use REST API. I utilized the Retrofit package for this purpose. It supports endpoint definition using annotations, which greatly simplifies code. Moreover, I use an interceptor both to add a bearer token to all requests and to handle error responses. This error handling is useful, especially when it comes to expired tokens. In this situation, I try to automatically refresh them using Auth0 SDK and when it succeeds, the new request is resent again. Only when the automatic refresh is impossible is a user redirected to the login screen. This way, the user is not required to login so often, as it is mostly possible to refresh the token automatically.

7.2 Backend Implementation

According to the assignment, I chose Jakarta EE 8 for the server side (I upgraded to version 10 as already mentioned in Chapter 5). It is a stable, well-tested, and production-ready platform for enterprise applications and information systems.[27]

Similar to the client application, I used a *"package by layer"*[26] approach to structuring the project. This proven method is suitable for smaller to medium-sized applications, which exactly fits the requirements. This way, the project contains a REST layer for exposing APIs, a service layer for logic and work with a database, and an essential model layer.

In terms of a model layer, I decided to use both JPA entities for the database and DTOs for the communication with client applications. This more robust approach contributes to cleaner code, higher security, and more flexible integration with clients. When it comes to mapping fields between those two models, I used the mapper from the Mapstruct package, which simplifies the whole mapping process significantly.

As I already mentioned, I used JPA in my implementation. According to Jakarta EE Cookbook, *"Jakarta Persistence (formerly JPA) is a specification that describes an interface for managing relational databases using Jakarta EE."* [28] In simple words, it enables object-relational mapping (ORM). Hence, it is possible to work with SQL database tables as if they were objects.

When it comes to database transactions, there is a great way to work with them in the Jakarta EE environment, as Moraes describes in his book Jakarta EE Cookbook.[28]. At first, I had to add the following line to the persistence.xml configuration file as shown in Listing 7.1.

```
<persistence-unit name="trippidy" transaction-type="JTA">
```

Listing 7.1: Transactions configuration

After that, all transactions are made automatically in EJB annotated classes. Therefore, I annotated my services as `@Stateless`, and every method of this service is executed as a single transaction. It means that all database changes that were done inside that method are either flushed to the database after return or roll-backed if there was any exception thrown.

For communication with a client, I created a REST API. I drew information from an official Jakarta EE tutorial.[29] Various annotations that are used to build this API are explained there.

7.3 Final Application

In this section, I show the screenshots of the final application. They were taken on a physical Android phone running the Flutter native application. As for the web version, it looks exactly the same, so I decided not to include it here. I also skipped many screens that were not the most important, like the edit profile screen, trip invitations screen, and more.

In the first pair of images (Figure 7.2), there is a list of trips where each tile directly shows a list of members and how many items that are related to the current user are already checked. This contributes to the possibility of seeing the progress of preparation. In addition, there are notifications in the top right corner. At the moment, those are trip invitations but there could be multiple types of them in the future. It could, for example, notify users about updates on their items and more.

In Figure 7.3, there is a personal list of items. The important thing here is that there are all items of the current user – even shared ones. It can be noticed that those shared items are also present in the shared list in Figure 7.4. This is a nice feature that allows a concrete user to check all of his items including those that are meant to be shared and also there is a possibility to see all shared items aggregated from all members in the shared list.

Individual tiles are organized into collapsible categories. Moreover, users can choose between two views. Either a full view with all categories in one screen or a tab view where they can select the desired category in a tab bar. This second one is supposed to be useful for bigger lists that would be too long for only one screen.

In addition, when using the tab view, a suggestions generator button appears in the top right corner. This generates suggestions for a category based on the trip name, category name, and already added items using the GPT-3.5-turbo model. Generated ideas can be added simply by clicking on them.

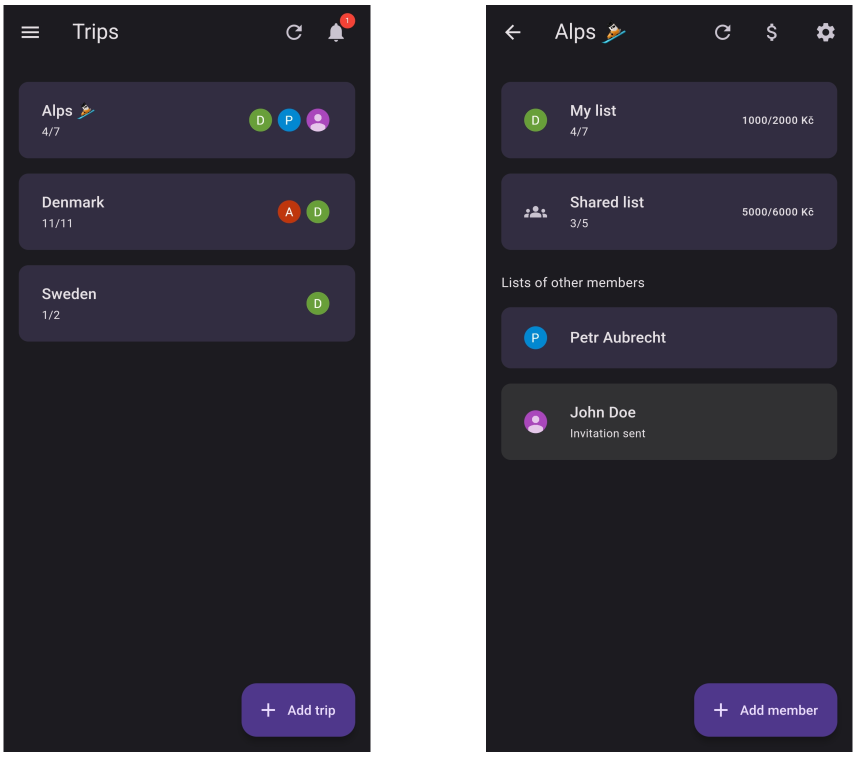


Figure 7.2: Final application – trip list and detail

In terms of items, apart from checked flags, they also show important data. In the personal list, if there is no icon, it means that it is a regular item. A group icon is for a shared one and the crossed eye represents a private item that cannot be seen by anyone else.

When tapped on an item tile, an edit screen, shown in Figure 7.5, is displayed. It is important here that it cannot be shared and private at the same time because it does not make sense to have shared items that nobody can see. In the bottom part, there is a price and a list of members who are supposed to participate in the payment.

This edit screen looks the same way for the Add Item button. The only differences are the prefilled values and different labels. This contributes to the overall consistency across the application. It also applies to different types of lists. There are some changes between them but they are more about limiting or adding features and the overall user interface stays similar. Thanks to this, switching between those lists feels very smooth.

Driven by my personal needs and those of my friends, I decided to include expense-splitting functionality. It can be seen in Figures 7.5 and 7.6. It also considers completed transactions. This ensures it works correctly even if the price of an item being split changes or if new items are added or removed.

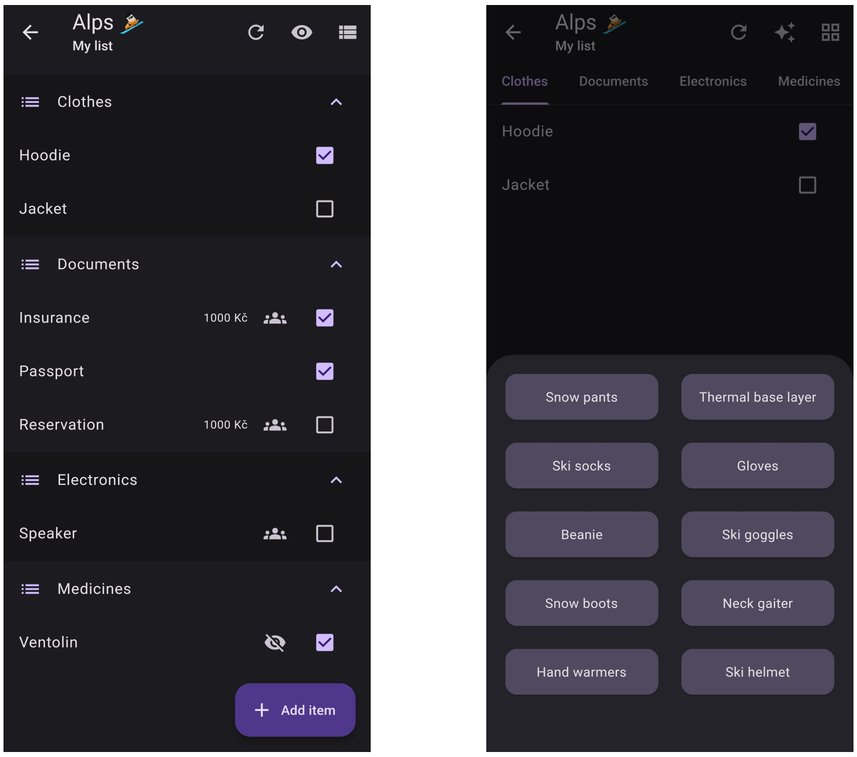
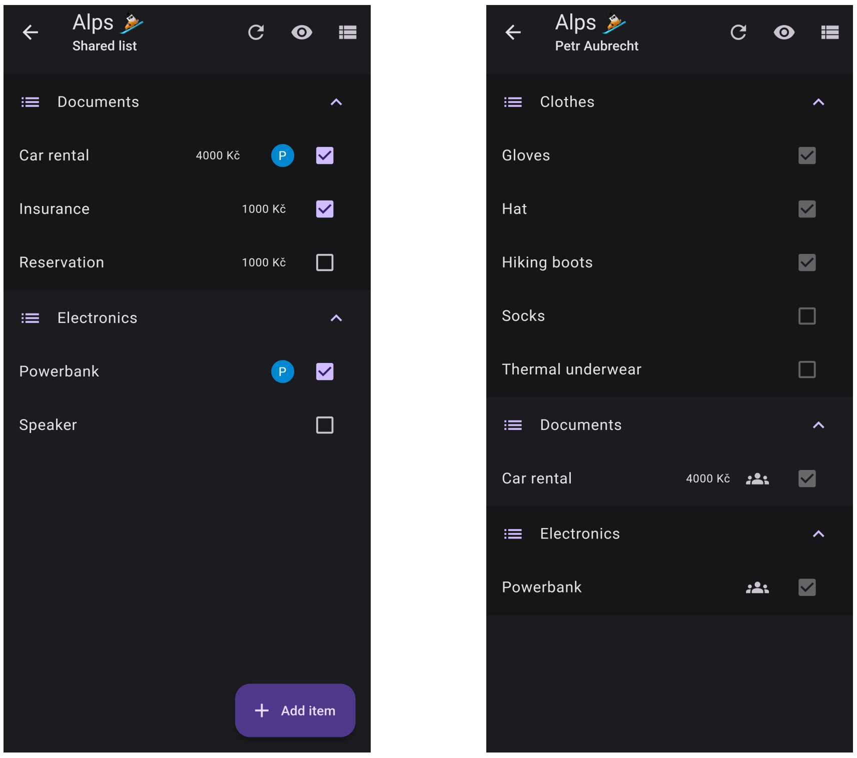


Figure 7.3: Final application – personal list with AI suggestions

The QR code payment feature is also worth mentioning. It uses a bank account number of the user profile to calculate the IBAN and generate payment details in the Short Payment Descriptor (SPAYD) format defined at QR Platba.[30] This string is then encoded into the QR code and is rendered. There are two ways to use it. Either it can be scanned by a phone camera when displayed on another device like a laptop or it can be used simply by tapping it. This tap allows one to open a mobile banking application and automatically prefills the required fields.

As the last screen, I included an Auth0 login page which is also consistent across all supported platforms. It is worth noting that all of those elements can be customized which might be beneficial in some cases.


Alps
Shared list

Documents
Car rental 4000 Kč P ✓
Insurance 1000 Kč ✓
Reservation 1000 Kč ☐

Electronics
Powerbank P ✓
Speaker ☐

Add item

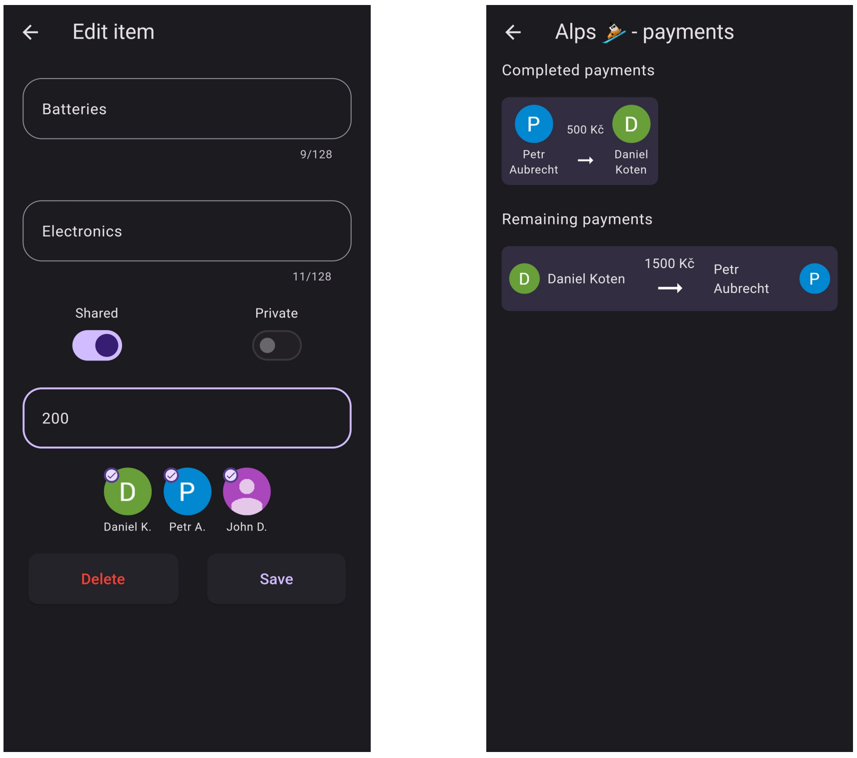
Alps
Petr Aubrecht

Clothes
Gloves ✓
Hat ✓
Hiking boots ✓
Socks ☐
Thermal underwear ☐

Documents
Car rental 4000 Kč ✓

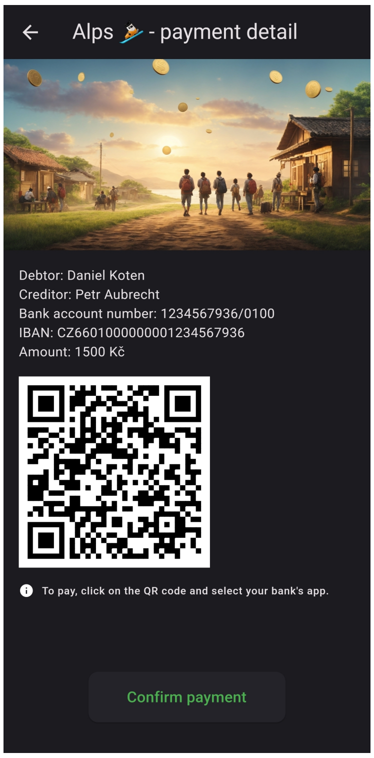
Electronics
Powerbank ✓

Figure 7.4: Final application – shared list and member's list


Edit item
Batteries
9/128
Electronics
11/128
Shared
Private
200
Daniel K. Petr A. John D.
Delete
Save

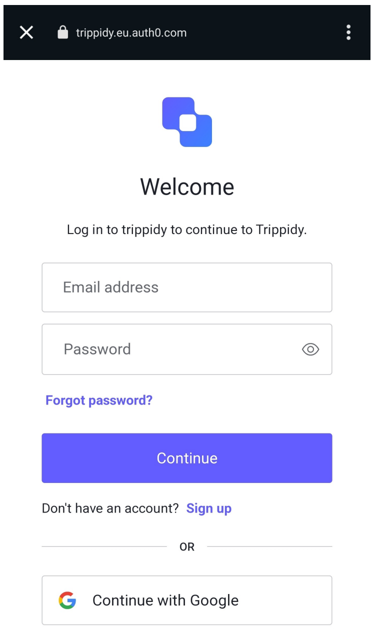
Alps - payments
Completed payments
Petr Aubrecht 500 Kč → Daniel Koten
Remaining payments
Daniel Koten 1500 Kč → Petr Aubrecht

Figure 7.5: Final application – item edit screen and payments


Alps 🏂 - payment detail

Debtor: Daniel Koten
Creditor: Petr Aubrecht
Bank account number: 1234567936/0100
IBAN: CZ6601000000001234567936
Amount: 1500 Kč

To pay, click on the QR code and select your bank's app.



Welcome

Log in to trippidy to continue to Trippidy.

Figure 7.6: Final application – payment detail and login screen

Chapter 8

Testing and Evaluation

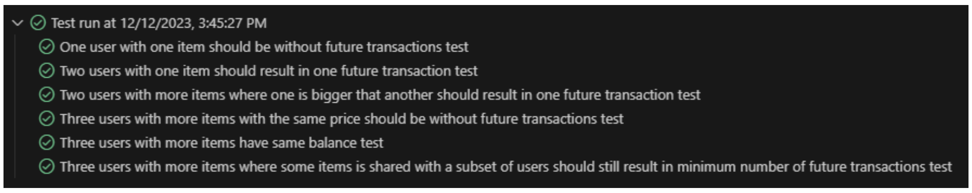
During and after the development process, I also dedicated a lot of effort to testing. We established a routine of weekly meetings with my supervisor where I always showed progress which I did. The important thing is that these consultations also served as mini-testing phases. Thanks to the evaluation of the progress in those smaller increments, it was possible to identify and address potential issues early. This approach convinced me that iterative development and continuous feedback are crucial in software engineering.

Apart from the continuous manual testing, I also implemented various types of automated tests. They mostly serve as so-called regression tests. According to Axelrod, *"Regression tests are tests that verify that a functionality, which previously worked as expected, is still working as expected."*[31]

8.1 Unit Testing

Firstly, I implemented unit tests. Namely, it seemed most important to me to test if the expenses splitting calculator works correctly. Hence, I added those tests to the client project. A nice thing about Flutter is the fact that it has a testing framework already prepared to use. All I had to do was to include a flutter_test dependency.

As for the tests itself, I created a dummy Trip object will all necessary data. This served as a template for objects used in individual tests. In each test, I created a deep copy of that template object using a copyWith method. This allows cloning the object with the possibility to change some of its fields.



```
✓ Test run at 12/12/2023, 3:45:27 PM
  ✓ One user with one item should be without future transactions test
  ✓ Two users with one item should result in one future transaction test
  ✓ Two users with more items where one is bigger that another should result in one future transaction test
  ✓ Three users with more items with the same price should be without future transactions test
  ✓ Three users with more items have same balance test
  ✓ Three users with more items where some items is shared with a subset of users should still result in minimum number of future transactions test
```

Figure 8.1: Unit tests results

This approach helped me a lot because I needed slightly different objects for every test, and this way I did not have to always create it from scratch.

All tests are also absolutely independent of each other and on the rest of the application. They always test only the things they are supposed to. For illustration, I included a screenshot of the test results in Figure 8.1

8.2 Integration Testing

Another type of test that I implemented were the integration tests. In my particular use case, they test the REST API. And since REST is the only method I use to communicate with the backend, those tests essentially test the backend.

These tests are very important for several reasons. The first thing is that they also serve mainly as regression tests, the same as the unit tests. But the critical thing is that they can help to identify where the problem might be when some bug appears during development. To be more specific, imagine a situation where there is a new feature added or updated. It was needed to edit both backend and client code and now the app does not work, and it is unclear if the problem is on the server side or in the client application. Thanks to the integration tests, it can be spotted if the backend part is the problem or probably not.

I decided to create another Maven project for this. It is because I do not want those tests to be run during the same Maven life cycle as the main project because it is not as easy to run the whole system. There are many services that have to run like database, Liquibase, etc. Therefore, I prefer to have a separate project that I can run whenever I want, and it is perfectly isolated.

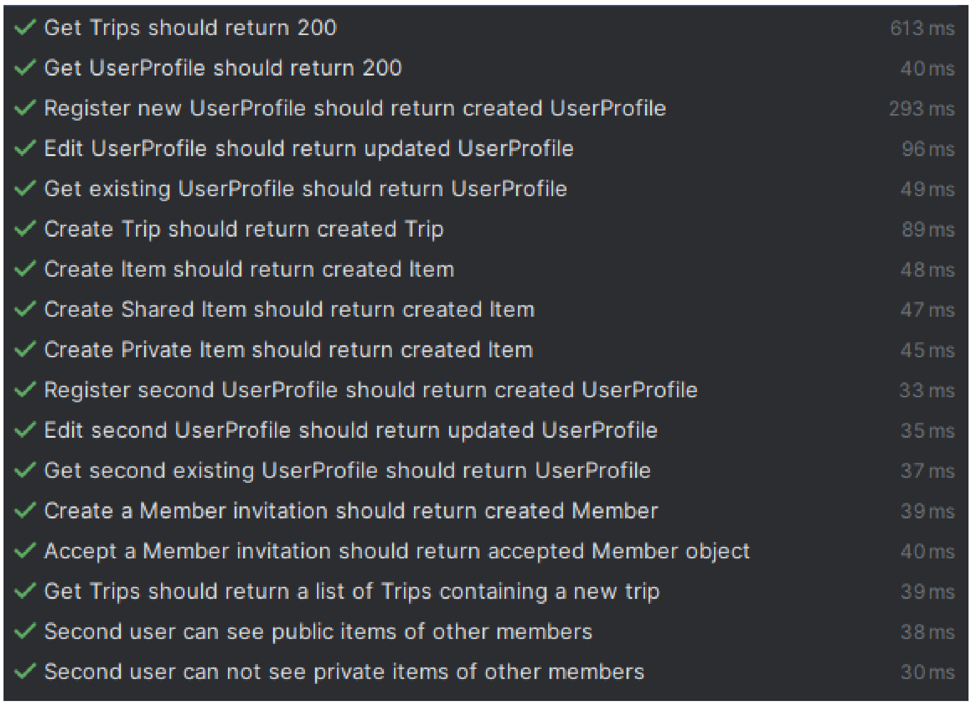
In terms of implementation, I utilized the JUnit 5 framework. As stated on the official website, *"JUnit 5 is the current generation of the JUnit testing framework, which provides a modern foundation for developer-side testing on* *the JVM. This includes focusing on Java 8 and above, as well as enabling many different styles of testing."*[32] In addition, I used REST Assured for much simpler and readable writing of tests and Hamcrest for more advanced matchers. This combination allowed me to write tests in a more clean and maintainable way. An example of a test can be seen in Listing 8.1. The precondition of this test is that there is such a user present in the database. The test then calls the appropriate API endpoint with the correct token of the logged-in user, and it should receive a response with status code 200 OK and a JSON body containing the correct ID and email. If there was anything different, the test would fail.

As all endpoints are secured and available only for logged users, I had to get a token somehow. I originally generated and changed the token manually, but it was not too efficient, so I decided to use the Auth0 package for Java, which allows me to get a token for a concrete user using his credentials. It also offers an API for manipulation with JWT tokens, which I used.

```java
@DisplayName("Get existing UserProfile should return it")
@Test
void getExistingUserProfileShouldReturnUserProfileTest() {
    RestAssured.given()
        .auth()
        .oauth2(idToken1)
        .when()
        .get("/my/userProfile")
        .then()
        .statusCode(200)
        .body("id", equalTo(userId1))
        .body("email", equalTo(userEmail1));
}
```

Listing 8.1: Integration test using REST Assured

In contrast to the unit tests, these ones are not completely independent as some of them create new data, which is then required by others. Nevertheless, all of those tests can be run repeatedly as the IDs are uniquely generated. It is also worth mentioning that even if it was required to run the tests against a clean database, it is possible to simply remove the volume from the database docker container, and it can start with a fresh state. Similarly to the unit tests, I included an illustrative image of the test results in Figure 8.2.



```
✓ Get Trips should return 200                                          613 ms
✓ Get UserProfile should return 200                                     40 ms
✓ Register new UserProfile should return created UserProfile           293 ms
✓ Edit UserProfile should return updated UserProfile                    96 ms
✓ Get existing UserProfile should return UserProfile                    49 ms
✓ Create Trip should return created Trip                                89 ms
✓ Create Item should return created Item                                48 ms
✓ Create Shared Item should return created Item                         47 ms
✓ Create Private Item should return created Item                        45 ms
✓ Register second UserProfile should return created UserProfile         33 ms
✓ Edit second UserProfile should return updated UserProfile             35 ms
✓ Get second existing UserProfile should return UserProfile             37 ms
✓ Create a Member invitation should return created Member               39 ms
✓ Accept a Member invitation should return accepted Member object       40 ms
✓ Get Trips should return a list of Trips containing a new trip         39 ms
✓ Second user can see public items of other members                     38 ms
✓ Second user can not see private items of other members                30 ms
```

Figure 8.2: Integration tests results

8.3 Performance Testing

It is crucial to know what load the system can hold. As Molyneaux (2015) points out, *"Non-performant (i.e., badly performing) applications generally don't deliver their intended benefit to an organization; they create a net cost of time and money, and a loss of kudos from the application users, and therefore can't be considered reliable assets. If a software application is not delivering its intended service in a performant and highly available manner, regardless of causation, this reflects badly on the architects, designers, coders, and testers"*[33]

Since this is an application that is supposed to enable collaboration in organizing a trip for multiple users at the same time, it is inherently very server-dependent. For this reason, speaking about the overall performance of the system, the biggest issues might be on the server side during the peak workload. Therefore, I focused on performance testing of the REST API.

During the process, I paid special attention to the number of simultaneously connected users, the number of requests they generated, and the response time in that situation. It was also important to test how the server behaves

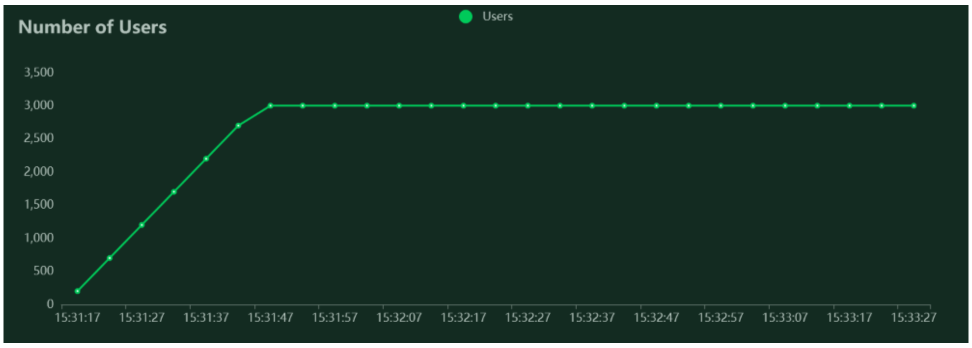


Figure 8.3: Number of Users

when it is overloaded. It must not crash even in that case. Rather it should reject the connection if it is not able to accept more requests.

I decided to use Locust as the testing tool for this purpose. It was mainly because it is a highly scalable tool with an official docker image, is well-documented, and uses Python for test scripts, which makes it easy to use. It also provides nice visualizations and statistics of the tests that were performed.

In terms of concrete setup, I used different machines for server hosting and Locust. Those machines were on the same network so the test results were not affected by my internet connection speed – in production use, it would run on a cloud or VPS platform, which provides a high-speed internet connection. Both server and Locust ran in Docker containers. As for the Locust configuration, there was 1 master node and 10 worker nodes used to generate requests seamlessly. The server container fully utilized 5 assigned CPU cores and 5 GB of RAM. Based on my experiments, it was not able to use more CPU cores so if more performance is needed, there would have to be created more server copies, and the workload would need to be distributed among them. Based on the NR-2 (that the system must be able to handle hundreds of simultaneously active users), I tested with 3000 users – it can be seen in Figure 8.3. Those users were generating approximately 1500 requests per second (RPS), as can be seen in Figure 8.4. The requests consisted of both read and write operations at a ratio of 3:1.

The results show that the system can handle such a workload without any problem. As can be seen in Figure 8.5, only the response time increases as the number of requests grows. Nevertheless, even during the peak workload of 3000 simultaneously connected users, the response time is about 2 seconds in the worst case. The important thing is that there was no single failure during the testing. All requests were handled successfully. As for the comparison of read and write operations, the response times for them were essentially the same – it is shown in Table 8.1 and Table 8.2. This is good because users

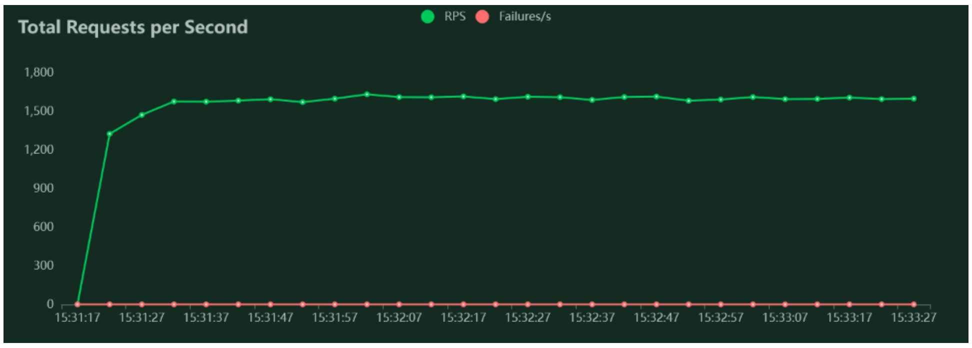


Figure 8.4: Total Requests per Second

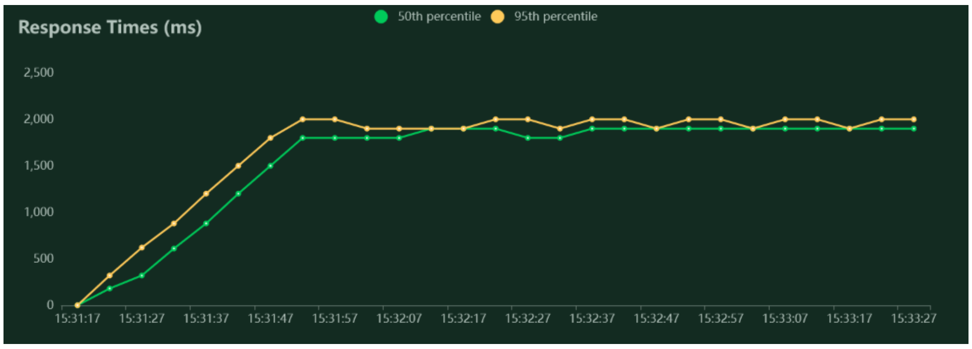


Figure 8.5: Response Times (ms)

will occasionally perform write operations, like adding new items, and those should also be executed fast.

To be sure how the system will behave during extreme workload, I also tried to test it with the number of users it is not capable of handling anymore. Concretely, I tried to generate requests from 20.000 simultaneously connected users. In that case, the server handled as many requests as it could, and it simply rejected the rest of them by ConnectionRefusedError(111, 'Connection refused'). This is the expected and correct behavior for me. It would be bad if the server crashed or if there were any data loss due to overload. The plotted results can be seen in Figure 8.6.

Would it be necessary in the future, it is eventually possible to add more server instances and balance the load using the Nginx reverse proxy server – it would also serve as a so-called load balancer in this case. However, this would bring new challenges to be solved. Namely, it would probably be needed to synchronize data like JPA caches between servers. After adding more Payara instances, it is likely that the database server will become the new bottleneck. It might be necessary to add more database servers and again synchronize data, which could be a much more complex problem. Finally,

Method	Name	Requests	Fails	Avg	Min	Max	Size	RPS
PUT	/api/v1/my/item	53213	0	1659	18	2064	268	399.4
GET	/api/v1/my/trip	53204	0	1661	28	2068	61857	399.4
GET	/api/v1/my/trip/ebf9...	53165	0	1658	28	2063	46951	399.1
GET	/api/v1/my/userProfile	53343	0	1655	21	2067	8181	400.4
Aggregated		212925	0	1658	18	2068	29295	1598.3

Table 8.1: Request Statistics

Method	Name	50% (ms)	90% (ms)	99% (ms)	100% (ms)
PUT	/api/v1/my/item	1800	1900	2000	2100
GET	/api/v1/my/trip	1800	1900	2000	2100
GET	/api/v1/my/trip/ebf9...	1800	1900	2000	2100
GET	/api/v1/my/userProfile	1800	1900	2000	2100
Aggregated		1800	1900	2000	2100

Table 8.2: Response Time Statistics

even the load balancer can, of course, become a bottleneck. There are some techniques for this situation, too. It would probably lead to a Round-robin DNS load-balancing technique. It means that there would be multiple IP addresses set for a single domain, and each of those IPs would go to a different load balancer. It can be seen in a diagram shown in Figure 8.7 from an AWS article.[3] This approach has more benefits. For example, it may solve the single point of failure problem – if only one loadbalancer was used and it crashed, then the whole system was unavailable. However, if DNS load balancing is utilized, other load balancers are probably still running. Another benefit could be balancing based on the location. For instance, the requests would go to the Europe servers for European users and to US servers for US users. The good thing is that the communication would be faster this way. On the other hand, consistency probably could no longer be optimal due to the CAP theorem.

"The CAP theorem states that in any massive distributed data management system, only two of the three properties consistency, availability, and partition tolerance can be ensured."[34]

To sum up, it is definitely possible to scale the system up in the future by adding more server instances or utilizing DNS load balancing. However, based on the CAP theorem, this would worsen consistency, which is an important requirement in my application, as stated in NR-3 in Chapter 4.

8.4 User Testing

The most detailed testing I did was user testing. It involved creating test scenarios simulating the expected real usage behavior. As it was more complex to organize, I arranged it only once in a later phase of development.

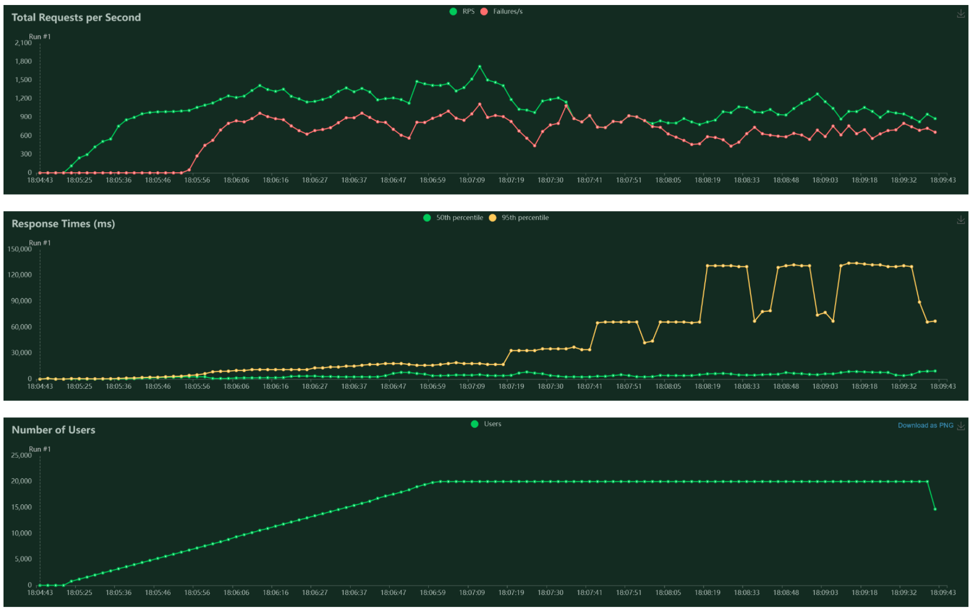


Figure 8.6: The performance test for too many users

Nevertheless, I personally conducted smoke tests that were similar to the given scenarios during the development process. Even though it required additional time to develop new features, it proved to be more beneficial in the long run, as fixing it at a later stage would have been more difficult and could have potentially impacted other parts of the application.

8.4.1 Testing Scenarios

In the following, I want to revisit and elaborate in greater detail on the testing scenarios previously mentioned. When designing the scenarios, the goal was to cover the requirements specified in Chapter 4 and to check if a user can figure out how to do a certain action by himself without too detailed instructions. I also instructed testers to feel free to share their thoughts and ideas about the app's functionalities.

First Login

As a new user, open a freshly installed app and register using email and password. Try to enter an invalid email and a weak password. After that, try to enter valid values based on the validator. Once successfully registered, fill in the additional required user information to start using the app. Log

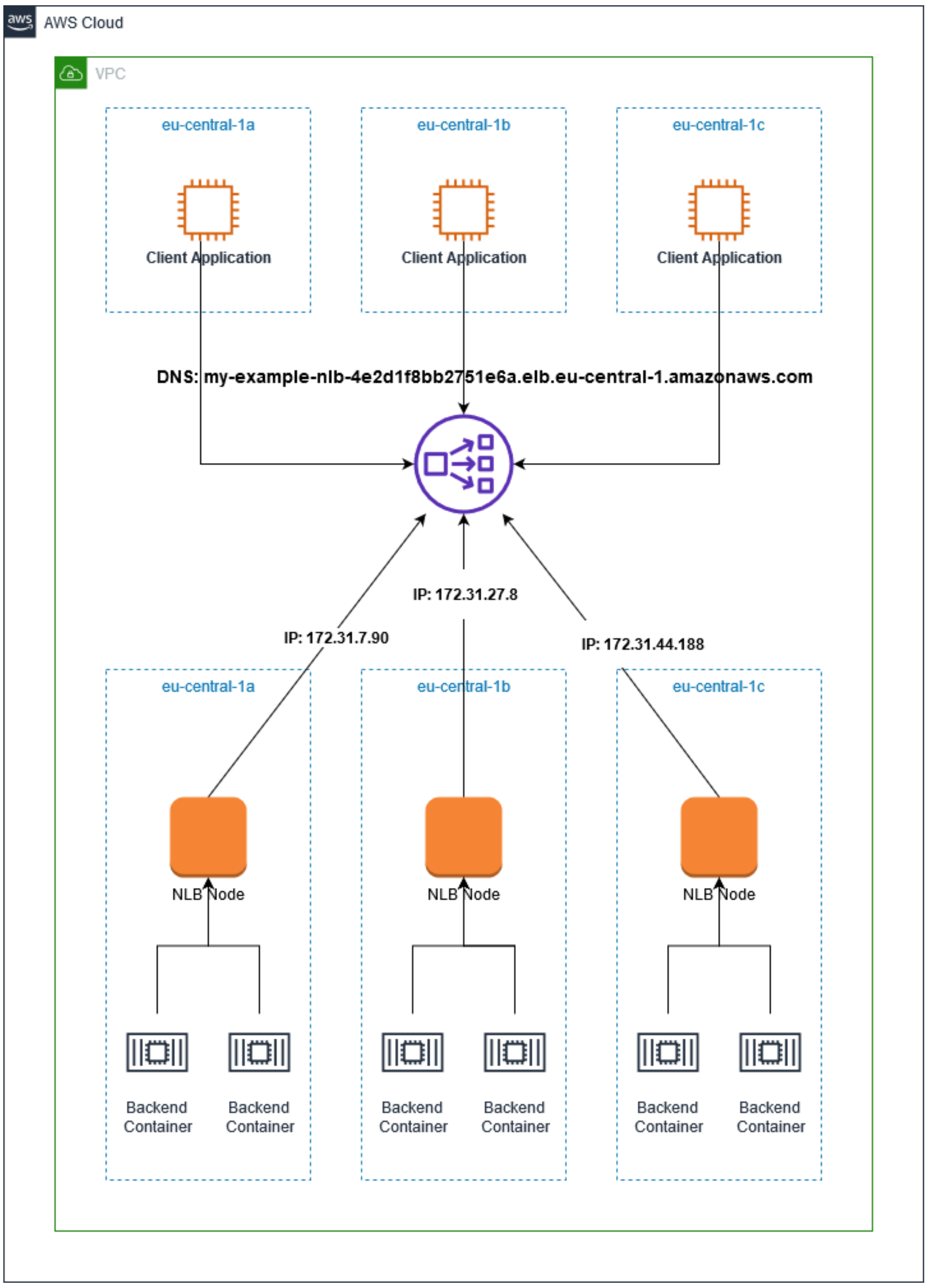


Figure 8.7: DNS Load Balancing – taken from AWS article Resolve DNS names of Network Load Balancer nodes to limit cross-Zone traffic.[3]

out from this account and create another one. Try to use the Google account now. Fill user information again, but this time use different values. Once redirected to the homepage, create a first trip.

Expected outcome:

- A user is not able to create a profile using too weak password.
- A user is able to switch accounts.
- A user can create new events.

Add Various Types of Items

Add some items to the personal list and the shared one as well. Try to add an item to the shared list in two different ways, respectively, from different places. There should be at least two item categories used. Try to add some items directly to the already existing category. Mark a few already existing items as private so future members cannot see them.

Expected outcome:

- A user is able to add new items of various types.
- A user is able to utilize multiple ways to add items.

Invite More Members

Now, add a user created at the beginning of the trip. Log out and log in as the first user. Accept the invitation and take a look at the trip details. Confirm that it is not allowed to delete the trip by a current user, and it is only possible to leave the trip.

Expected outcome:

- A user is able to add existing members by their firstname or lastname.
- A user is not able to delete a trip if he is not an owner.

Check the Correct Visibility of Items

Verify that there are already added items in both the shared list and another member's personal list. Verify that this user does not see any private items of the second user.

Expected outcome:

- A user is able to see all shared items from all members in the shared list
- A user is not able to see the private items of other members.

Generate Some Suggestions

Try to generate some item suggestions and add a few of them to the list.

Expected outcome:

- A user finds out that he has to switch a view type to be able to generate suggestions.
- A user is able to add generated items to the correct category.

Set Prices to Some Items

Mark some items as shared, set a price for them, and choose who is participating in paying for them. Try to set a different amount of payment participants for at least two items.

Expected outcome:

- A user is able to set prices for items.
- A user is able to select multiple members to participate in the payment.

Check the Calculated Costs Split

Find a section where the calculated expenses spit is shown. Verify that the future transactions are correctly calculated.

Expected outcome:

- A user manages to find a place where the future transactions are listed.

Pay the Dept and Check the Costs Split

For completeness, fill in the bank account number of the payee user profile so it is possible to generate a payment QR code. Log in as the second user and try to use the QR code to open the banking application and verify that the filled payment information is correct. Settle the debt by marking the transaction as completed. Finally, check the list of future transactions again and confirm that the transaction is moved to the completed ones.

Expected outcome:

- A user is able to add a bank account number to his user profile.
- A user is able to use the QR code to realize the payment.
- The transaction is successfully moved to the completed state.

Copy the Trip

Copy the current trip and open its details. Check all the types of checklists and verify that there are only items that were owned by the current user, plus shared items from all members. There must not be the private items of other members. Finally, only the current user should be listed as a member.

Expected outcome:

- A user is able to copy a trip.
- Only current user's items and shared items of all members are copied.
- The current user is the only member in the copied trip.

8.4.2 Participants

The initial question that I encountered was about how many testers would be required. I read that in most cases, 5 is the optimal amount. This number of testers usually finds most of the problems, and it is not cost-effective to use more people. Of course, it depends on the project being tested. If it is an online banking system, it should definitely be tested more. I drew from the article Why You Only Need to Test with 5 Users.[35]

As for the concrete people, the testing group consisted of my family and friends. Their experience with mobile apps varied a lot. While on the one hand, one of the testers knows almost nothing about mobile apps, on the other hand, there are a few of them that have a lot of experience with it. This diversity was good because it led to varied test results thanks to it.

8.4.3 Results Evaluation

The user testing brought multiple insights into how the application performs regarding usability. Testers also shared their preferences on how they would like to use the application, which inspired me to reconsider it from a different perspective and slightly adjust the app's behavior.

For some participants, it was surprisingly not as straightforward to switch accounts. All of them managed to register a new profile, but one of them had problems finding a logout button which is in the navigation drawer, and he tried to log out by closing and reopening the app, but the currently logged user is automatically logged in this case. In my opinion, this might be caused by a lack of experience as the user does not use mobile applications so much. Nevertheless, it might be worth considering adding a simple tutorial.

A second user is stuck on Google login. There was a problem with multi-factor authentication (MFA) as she could not obtain a code from the Google Authenticator app. However, this is not a problem of a tested application and also that is the reason why I wanted to support classic username and password access as well. To be able to finish the rest of the scenarios, she created another user account the same way she created the first one.

Other features were more challenging to find for less experienced users. Namely, it is an automatic items suggestions generator and setting the bank account number in the user profile. As for the first one, it was sufficient to give advice to switch a view type from a list to a tab view. Then, all users

Scenario	User 1	User 2	User 3	User 4	User 5
First Login	On own	With help	With help	On own	On own
Add Various Types of Items	On own	On own	On own	On own	On own
Invite More Members	On own	With help	On own	On own	On own
Check the Correct Visibility of Items	On own	Suggestion	On own	On own	Bug
Generate Some Suggestions	With help	With help	On own	On own	On own
Set Prices to Some Items	On own	On own	On own	On own	On own
Check the Calculated Costs Split	On own	On own	On own	On own	On own
Pay the Dept and Check the Costs Split	On own	With help	On own	On own	On own
Copy the Trip	On own	On own	On own	On own	On own

Table 8.3: User testing results

were able to find a button and use it. Speaking about the user profile update, it is related to the navigation drawer again. However, when he found this, he managed to complete the rest easily.

Apart from usability problems, there was also a suggestion that editing shared items of other members should be possible. I disabled it because I thought it was a correct behavior, but based on the feedback, there are some use cases where it makes sense to allow it. For example, there can be a family group, and a parent wants to mark the children's item as checked. After this, I reconsidered my opinion and allowed all members to edit all shared items.

Finally, there was one bug discovered. It was about a pull-to-refresh feature that did not load new data. The problem was found because the tester tested simultaneously both Android and web applications. It allowed him to see the difference between those two running instances, which would not be possible on a single instance and switching accounts. After this, I fixed the problem and verified that it worked correctly by following the same scenario as the tester did.

Chapter 9

Conclusion

The outcome of this thesis is a functional mobile and web application for an easy organization of events with a focus on checklists management and the cooperative aspect. The resulting solution was tailored to the given use case rather than trying to make a checklist app for a general purpose. It uses a custom backend with multiple services integrated.

As part of the cooperative aspect, I have dealt with the issue of item sharing and their visibility. In the end, the app contains three types of lists. The most important one is a personal checklist where users can add items. Furthermore, there is a shared list where users can see the aggregated shared things from all members. And finally, users can see the public entities of every member for inspiration.

The application also provides useful time-saving features like expense splitting with support to open a banking application with all payment details directly from the app. Furthermore, it can generate item suggestions based on the context using the GPT-3.5-turbo model. Past trips can also be reused as a template for a new one. And finally, it is possible to see the progress of preparation.

The whole system shown in Figure 6.6 on the page 36 is deployed on a test server and is working. The primary component is the Flutter mobile application, which also runs as a web application. As for the Android application, it is deployed on Google Play in the internal testing phase. It means it is already digitally signed and ready to be installed on testers' devices.

During the development, I utilized and integrated many different technologies like Flutter for a multi-platform client, Jakarta EE 10 running on the Payara server, PostgreSQL in combination with JPA for data persistence, Liquibase for managing the database schema, Nginx as a reverse proxy, Certbot for automatic certificate renewals. In addition, all this is orchestrated using Docker Compose. Finally, the whole deployment process is automated using the Jenkins build server, making it perfectly maintainable and expandable.

There are also several external services integrated. One of them is OpenAI GPT API for automatic suggestions, and another one is Auth0 for authentication and authorization. On top of that, there is a communication channel with Let's Encrypt CA to keep the certificates updated, and finally, GitHub communicates with a build server to make the deployment fully automatic.

In terms of testing, I combined multiple types of tests. The first one is Unit tests. They focus on the computational part, which is expense splitting. Secondly, integration tests check the REST API and serve mainly as regression tests. Moreover, I conducted performance tests to verify how many users the system is capable of handling and how it will behave when this limit is exceeded. On top of that, I organized user testing with five users with various levels of experience. They were given the test scenarios. I observed them and subsequently made an evaluation based on their performance and the problems found.

Finally, there are still ideas for future work. Speaking about the functionalities, it would be good to implement some polls for voting. It could even be used for brainstorming ideas for activities on a trip. When it comes to non-functional requirements, more performance may be needed if the app becomes popular. In this case, it will have to be scaled up, and the data synchronization will have to be addressed as discussed in Chapter 8.3.

Bibliography

[1] How many smartphones are in the world? https://www.bankmycell.com/blog/how-many-phones-are-in-the-world. Accessed: 2023-11-20.

[2] Authorization code flow with proof key for code exchange (pkce). https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-proof-key-for-code-exchange-pkce. Accessed: 2023-04-05.

[3] Resolve dns names of network load balancer nodes to limit cross-zone traffic. https://aws.amazon.com/blogs/networking-and-content-delivery/resolve-dns-names-of-network-load-balancer-nodes-to-limit-cross-zone-traffic/. Accessed: 2023-11-20.

[4] Google keep. https://play.google.com/store/apps/details?id=com.google.android.keep. Accessed: 2023-05-10.

[5] Tripit: Travel planner. https://play.google.com/store/apps/details?id=com.tripit. Accessed: 2023-05-11.

[6] How much privacy does one give up by using tripit? https://www.quora.com/How-much-privacy-does-one-give-up-by-using-TripIt-1. Accessed: 2023-10-29.

[7] Wanderlog – trip planner app. https://play.google.com/store/apps/details?id=com.wanderlog.android. Accessed: 2023-05-11.

[8] Payara server. https://docs.payara.fish/community/docs/Overview.html. Accessed: 2023-03-01.

[9] Sqlite vs mysql vs postgresql: A comparison of relational database management systems. https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems. Accessed: 2023-02-27.

[10] Martin Fowler and David Rice. *Patterns of enterprise application architecture*. Addison-Wesley, Boston, 2003.

[11] Flutter vs. react native vs. native: A comprehensive comparison for mobile app development. https://medium.com/@samra.sajjad0001/flutter-vs-react-native-vs-native-a-comprehensive-comparison-for-mobile-app-development-601b09e2fa56. Accessed: 2023-09-10.

[12] Flutter. https://docs.flutter.dev/. Accessed: 2023-05-10.

[13] Eric Windmill and Ray Rischpater. *No JavaScript bridge*. Manning Publications Co, 2020.

[14] React native vs. flutter: A performance comparison between cross-platform mobile application development frameworks. https://www.diva-portal.org/smash/get/diva2:1768521/FULLTEXT01.pdf. Accessed: 2023-09-10.

[15] Keycloak. https://www.keycloak.org/documentation. Accessed: 2023-09-11.

[16] Firebase auth. https://firebase.google.com/docs/auth. Accessed: 2023-05-29.

[17] Haroon Shakirat Oluwatosin. Client-server model. https://www.researchgate.net/profile/Shakirat-Sulyman/publication/271295146_Client-Server_Model/links/5864e11308ae8fce490c1b01/Client-Server-Model.pdf. Accessed: 2023-05-20.

[18] Martin Clay Fowler and Kent Beck. *Databases*, page 73–74. Addison-Wesley, 2019.

[19] Evolutionary database design. https://martinfowler.com/articles/evodb.html. Accessed: 2023-10-20.

[20] Transport layer security (tls). https://csrc.nist.gov/glossary/term/Transport_Layer_Security. Accessed: 2023-10-10.

[21] What's certbot? https://certbot.eff.org/pages/about. Accessed: 2023-10-14.

[22] Auth0. https://auth0.com/docs/get-started/auth0-overview. Accessed: 2023-05-23.

[23] Flutter riverpod 2.0: The ultimate guide. https://codewithandrea.com/articles/flutter-state-management-riverpod/. Accessed: 2023-05-28.

[24] Sander Rossel. *Continuous Integration, Delivery, and Deployment*. Packt Publishing, 2017.

[25] Configuring ssh connection to a remote host in jenkins (ssh-plugin). https://medium.com/cloudera-devops-beyond/configuring-ssh-connection-to-remote-host-in-jenkins-ssh-plugin-e2e9a00559f1. Accessed: 2023-10-16.

[26] Package by type, -by layer, -by feature vs "package by layered feature". https://proandroiddev.com/package-by-type-by-layer-by-feature-vs-package-by-layered-feature-e59921a4dffa. Accessed: 2023-03-02.

[27] Antonio Goncalves. *Java EE 7 at a Glance*, pages 1–22. Apress, Berkeley, CA, 2013.

[28] Elder Moraes. *Jakarta EE Cookbook Practical Recipes for Enterprise Java developers to deliver large scale applications with Jakarta EE.* Packt, 2020.

[29] The jakarta® ee tutorial. https://eclipse-ee4j.github.io/jakartaee-tutorial/. Accessed: 2023-08-10.

[30] Popis formátu qr platba. https://qr-platba.cz/pro-vyvojare/specifikace-formatu/. Accessed: 2023-10-20.

[31] Arnon Axelrod and ProQuest Ebook Central (online služba). *Complete guide to test automation: techniques, practices, and patterns for building and maintaining effective software projects.* Apress, New York, 1 edition, 2018.

[32] Junit 5. https://junit.org/junit5/docs/current/user-guide/. Accessed: 2023-10-25.

[33] Ian Molyneaux. *Why Performance Test?* O'Reilly, 2015.

[34] Andreas Meier. *SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management.* Springer Nature, Wiesbaden, 1st edition, 2019.

[35] Why you only need to test with 5 users. https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/. Accessed: 2023-10-28.

Appendix A

Installation

Installation is a bit more complex due to the certificates, domains, required SDKs, etc. Please follow the instructions in the Readme files located in the git repositories for more detailed installation steps. Further steps in this document are rather for illustration purposes.

```
Backend

$ git clone https://github.com/koty10/trippidy-server.git
$ mvn clean package
$ docker-compose up

Flutter application

$ git clone https://github.com/koty10/trippidy.git
$ flutter pub get
$ dart run build\_runner build --delete-conflicting-outputs
$ flutter build lib/main.dart
install an .apk file from build/app/outputs/flutter-apk

Tests

$ git clone https://github.com/koty10/trippidy-server-test.git
Integration tests can be run using IDE
Performance tests can be run using docker-compose up
```

Appendix B

List of Abbreviations

The following list describes the abbreviations used in the paper.

API	Application Programming Interface
BE	Backend – system layer that mediates communication between clients and other layers, such as the database
CI/CD	Continuous Integration & Continuous Delivery
DNS	Domain Name System
DTO	Data Transfer Object
FE	Frontend – the presentation layer of the system, in this case the mobile and web applications
FR	Functional requirement
JPA	Java Persistence API (Jakarta Persistence)
JWT	JSON Web Token
NFR	Non-functional requirement – quality requirement
ORM	Object–relational mapping – a technique for working with tables in a relational database as if they were objects
REST	Representational State Transfer
TLS	Transport Layer Security
VPS	Virtual Private Server