Warsaw University of Technology

FACULTY OF
MATHEMATICS AND INFORMATION SCIENCE

Bachelor's diploma thesis

in the field of study Computer Science

Web Application To Connect Travelers

Rozalia Korycka

student record book number 288496

Rafał Bogdanowicz

student record book number 290543

thesis supervisor

Rafał Jóźwiak, PhD

WARSAW 2021

....

....

supervisor's signature authors' signatures

Abstract

Web Application To Connect Travelers

Most travelers have faced the difficulty of finding or persuading a friend to join them on their travels. It is often problematic to find someone with a similar vision for the trip and who also has the time and money to go through with the plan. Furthermore, finding or creating an entire group proves to be a difficult task if one does not already have friends or acquaintances that travel.

This work aims at solving this problem, it is meant to come as an aid for all wanting to travel with company. Whether it is to expand one's travel group or create a brand new one, this web application is here for that. Users can register, create their own profiles, and plan trips. Based on data inputted during trip creation the users are matched with other users and groups. It is then possible to message users, apply to groups, or create a group. Groups have their own private chat in which they can further plan out their trip.

This project is in form of a web application which was created using React framework for the front-end and Spring Boot framework for the back-end. The application's data is stored in a PostgreSQL relational database.

Keywords: web application, travel, social, ReactJS, Spring Boot, PostreSQL

Streszczenie

Aplikacja internetowa łącząca osoby podróżujące

Większość podróżników napotkała problem, jak znaleźć i przekonać swoich znajomych do wspólnych podróży. Często problematyczne jest znalezienie kogoś, kto ma podobną wizję wyjazdu, a także ma czas i pieniądze na realizację planu. Co więcej, znalezienie lub zorganizowanie całej grupy stanowi poważny problem, szczególnie jeśli dana osoba nie posiada znajomych, którzy chcieliby wspólnie podróżować.

Ta praca ma na celu rozwiązanie tego problemu. Została stworzona, żeby pomóc wszystkim osobom, które chcą podróżować w towarzystwie. Bez względu na to, czy chcemy powiększyć naszą istniejącą już grupę o kolejne osoby, czy stworzyć grupę od zera. Użytkownicy tej aplikacji mogą się zarejestrować, stworzyć własny profil i planować wycieczki. Na podstawie danych wprowadzonych do aplikacji podczas tworzenia wycieczki, użytkownicy są parowani z innymi użytkownikami oraz grupami. Aplikacja zapewnia możliwość komunikacji tekstowej, dołączania do grup i zakładania nowych. Grupy posiadają własne, prywatne czaty, w których ich członkowie mogą szczegółowo planować swoje wycieczki.

Projekt został stworzony w formie aplikacji webowej, której front-end został zbudowany przy użyciu frameworka React, a back-end przy użyciu frameworka Spring Boot. Dane aplikacji są przechowywane w relacyjnej bazie danych PostgreSQL.

Słowa kluczowe: aplikacja webowa, podróże, społeczność, ReactJS, Spring Boot, PostgreSQL

Warsaw, ...

Declaration

I hereby declare that my part of the Engineering thesis (according to the division of work described in the Section 6.1 of the thesis) entitled „Web Application To Connect Travelers", submitted for the Engineer degree, supervised by Rafał Jóźwiak, PhD, is entirely my original work apart from the recognized reference.

.....

.....

Contents

1. Introduction

This chapter covers the general description of the diploma thesis project. It specifies functional and non-functional requirements, presents some existing non-perfect solutions, and explains the choice of technologies used to complete the project.

1.1. General Description

People are social beings and they need the company of other people. However, it does not seem like anyone is looking into the fact, that many travelers have no one to travel with. Solo travelling is on the rise not only because people want to travel alone but also because most of them do not have anyone to travel with, who shares their interests or even anyone at all.

The aim of our work is to provide travelers with a tool that will assist them at their search for travel partners or groups. There are some other tools on the market, which some travelers try to use; however, none of them match the standard we feel is enough to satisfy the users. Thus, some simply give up on their search and either decide not to travel or to travel alone. The goal of our application is to provide a dedicated, fast, simple and user-friendly service to those who are looking to expand their travel party.

The biggest obstacles travelers encounter while looking for companions is that they have to:

- Be interested in traveling to the same place and for the same or similar amount of time.
- Be able to travel at the same time.
- Be in the right demographic.
- Be interested in similar activities.
- Have a similar budget.

And these obstacles become even more burdensome while looking for an entire group.

Therefore, to create a valuable application we concentrated on these issues. Firstly, users save time on their search simply because we have created a dedicated service for them. Meaning no time has to be wasted simply looking for other travelers. Furthermore, we created tools that will greatly assist in finding the right travel buddies. The paramount components are trips and groups, they are the backbone of this web application. Users are able to create and plan a trip by simply filling out a form. The form not only takes simple information like the dates of the trip and location but also asks about the preferred group size, what age and gender the people should be, and of course, the activities of interest. This information is used to match users to each other and to groups. Saving users countless minutes or hours looking through travelers.

To further assist the process, we implemented a chat system into the web application allowing users to easily start conversations with matched users. On top of private conversations, we also created group chats. Each trip group has their own chat where all the trip details can be further discussed.

Lastly, a budget tool was added. Allowing the group to easily plan expenses and keep track of them to avoid situations of miscommunication. Our system automatically calculates and displays the expected expenses per group member.

We believe we created a tool that allows users to effortlessly organize trips and form travel groups.

1.2. Requirements specification

1.2.1. Functional requirements

Before we started the development phase, we made a list of functional requirements. These are all functions seen by a user:

1. Create new account – it is the first functionality that will be used by a new user. User will input some personal information such as: name, e-mail address, date of birth, gender, a list of spoken languages, current country of residency, and a short description about himself. Apart from that the user will need to input a password which will be later used to log in into the service. In case of invalid data input the application should inform the user with an error message.

2. Log in – a user will be able to log in into the application using e-mail address and password linked with his account. In case of entering invalid credentials, the application should inform the user with an error message.

3. Log out – when a user will be logged in, he must have the option to log out.

4. Change password – a logged in user will have the possibility to change password linked to his account. Because of security reasons, before changing the password, the old password must be entered.

5. Edit profile information – user should be able to change personal information linked to his account such as: name, date of birth, gender, a list of spoken languages, current country of residency, and a short description.

6. Upload / change profile picture – there will be a possibility to upload a picture linked with the account.

7. Delete account – if a user decides to resign from using the application, there will be a possibility to remove the account. Because of security reasons, it should be confirmed using the current password. This function will remove all the user information from the database. After completion, there will be no way to recover the account back.

8. Display user's profile – a logged in user will be able to see other users' profiles. Personal information about the user will appear, including a profile picture.

9. Add a trip – the application will allow to add a trip to the account. Firstly, a user will need to specify some details such as: name, location, date, preferable activities, preferable group type (male / female only, mixed), group size and age range of its members.

10. Delete a trip – The application should allow to remove a saved trip as well. After removal there would be no way to recover the data back.

11. Browse trips – a feature that will translate users input into a query that will look for matching groups and users. It means that a user will be able to browse through existing groups and profiles which match to given criteria such as: location, date, preferable activities, preferable group type (male / female only, mixed), group size and age range of its members. We believe it will be the most important feature in the project since it would allow to find other travelers going to the same location at the same time.

12. Add / join a group – the application will allow to create groups for existing trips. A group will associate travelers who are planning to spend their vacation together. Each

group should have an owner who has some extra permissions such as: accepting / rejecting pending users and changing group's currency. Each member of a group should be able to leave it in any moment.

13. Messaging – the application will allow to send messages in a chat. There will be two kinds of chats: private one between two users and a group one. All members of a group will be able to send and receive messages in a group chat linked with that group. A user, who will want to to send a message to another user, should be able to do it using a button on the profile page of that person.

14. Add / remove a bill – a feature that will allow to add estimated costs to a group. The owner of the group will have permission to add and remove bills and change currency. Each member of the group will be able to see what the total estimated cost is and how much it is per person.

1.2.2. Non-functional requirements

The tool which we develop is a web application. The only software which will be needed to use it is a web browser. Users should be able to use all kinds of commonly used devices such as PCs, laptops, or tablets to connect to the application. We want to ensure that the user experience will be satisfying for everyone – which means it must have quick response time and be accessible through all popular browsers. Moreover, it must work correctly on all most popular web browsers.

The list of non-functional requirements is given below:

1. The application must be safe. Any data leak is unacceptable. Users will be authenticated using their e-mail address and password. They can not share their password publicly or it may result in their account being accessed by an unwanted person with malicious intentions.

2. The application must be responsive all the time. Potential shutdowns should be minimized.

3. The database must be able to store data of thousands of users.

4. Hundreds of users should be able to use the application simultaneously.

5. The application should be easy to maintain and extend in the future.

6. The application must be available in English language.

7. The application must have a user-friendly interface.

1.3. Existing solutions

The idea of this application was to create a simple tool for travelers, which would help them find companions. We wanted to make it possible for users to find other people with similar interests and expectations about their trip. There are popular web services and applications which travelers may choose to use. The most popular applications for travelers are listed below, together with their strong points and drawbacks.

1.3.1. Facebook traveling groups

We believe it is the most popular platform to search for travel companion. There are many groups which are dedicated to specific countries. One can add a post containing information about trip destination and departure date, asking if there is anyone else who would like to join the trip. Other members can ask questions about the offer in the comment section.

The problem with such groups is that usually they are chaotic. It is hard to find people at similar age, going to the same destination at the same time. Most people who publish such posts do not specify what kind of attractions they are interested in. Moreover, if someone prefers to travel in larger groups, it is almost impossible to plan such a trip.

We have already had some experience with finding trip buddies in this way and we think that this process could be improved.

1.3.2. Meetup.com

It is a platform for finding and building local communities. People use this application to meet new people and get out of their comfort zone. It is also often used by travelers who have the opportunity to find new acquaintances at their destination. Everyone can find a meeting according to their personal preference, since the service offers a wide range of activities, especially in big cities.

This platform is not ideal though. Meetup.com is rather used to make an appointment for one-time meeting (e.g., a city tour). It is hard to find others for the entire period of stay.

1.3.3. Couchsurfing.com

It is a community website where users can create profiles to tell fellow members about them, and then use that to: gain access to free accommodation in a local's home (as a traveler), host a traveler at home (as a host) or meet-up with people from around the world (as community members). It is a great application for travelers, but it does not offer a feature of finding like-

minded people like our application does. One can find or organize an event with other users of the website, but these are just single events, not entire trips.

1.3.4. Conclusions

None of the most popular applications and web services allow to get connected with like-minded travelers easily. A person who would like to find other people using these platforms would have to spend much time on searching. Moreover, the outcome may not be satisfactory. We have not found any tool which would meet given functional and non-functional requirements.

1.4. Overview of technologies used

We decided to do the front-end part in React.js library. There were many options to choose from, however, since we had no previous experience, we had a clean slate. We decided to go with JavaScript because it is very popular at the moment and learning it might prove benefitial for our future careers. It is possible to create a website just using HTML, CSS and JavaScript but making it using a framework or a library gives us much more structure and access to many tools. We could have also chosen to use Angular but decided on React because it uses virtual DOM. Real DOM, instead of changing just a particular section of the DOM that requires an update, updates the entire tree structure of HTML tables until it reaches the needed data. Virtual DOM allows us to update only specific sections of the Real DOM. This theoretically renders updates much faster and ensures fast performance – no matter of the apps' size.

The most popular technologies we had to choose from to build the web server were Java Spring Boot framework and C# ASP .NET Core. We decided for Java language because it is more popular in commercial use, and we wanted to gain new experience, since we have not used Spring framework before. Spring Boot has many advantages such as easy to use modules providing functionalities which were important to us (e.g., security or data module).

To store the application data, we decided to use a relational PostgreSQL database. We were also considering to use a non-relational MongoDB database but we decided to stick to the more standard solution.

2. Description of the solution

2.1. System architecture

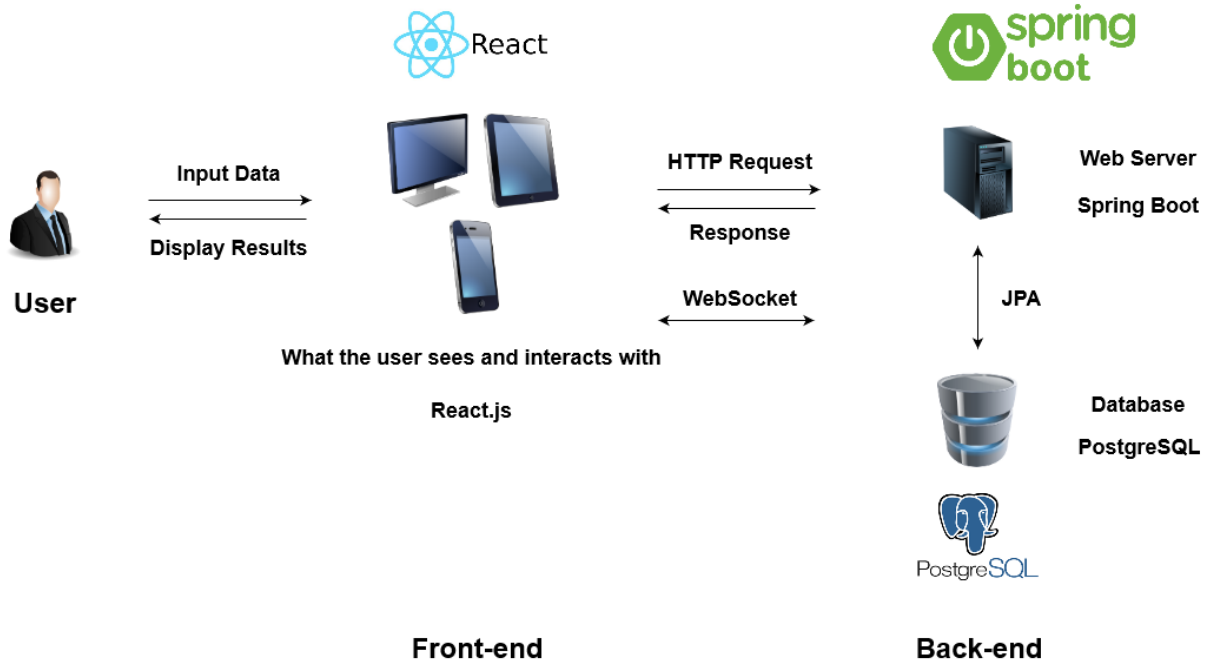

Figure 2.1: General system architecture of the project

The application is divided into two main components: front-end and back-end. The user interacts with the user interface built using React.js. React.js application processes the user input and communicates with the web server using HTTP requests. The server (which is built using Spring Boot framework) contains the logic of the application. If needed, the web server communicates and exchanges data with PostgreSQL database. Then, HTTP response is sent back to the front-end component and displayed to the user.

2.2. Spring Boot

2.2.1. General class schema

Spring Boot framework requires specific class schema for our main components. Following the schema helps organizing the source code in a clean and easy to understand form. There are 4 main types of classes which are described below:

- Model – classes of this type work like containers that contain the data of the application. They store information how the main objects in the application (such as trips, groups, users) are built.
- Repository – these classes specify what kinds of database operations will be necessary. Spring Boot Data JPA framework takes care of low-level database communication, so it is not necessary to manually write SQL queries.
- Service – classes of this type contain implementations of methods which allow us to get actual instances of objects, whose data is stored in the database. These service classes contain public methods and can be used in any part of the code. They are responsible for business logic of the application.
- Controller – these classes define publicly available endpoints which can be used to retrieve necessary information from the web server. They are responsible for handling incoming requests.

2.2.2. Spring Boot features

Spring Boot is an extension of the existing Spring framework and it has some specific features which make it easier to work in the development ecosystem. These include pre-configurable starter dependencies that simplify the initial configuration of the application which can get complicated and time-consuming for other Spring projects. Spring Boot has a few features that make it easy to quickly develop a Java application from scratch, such as auto-configuration, health checks, and numerous dependencies. In our application we take advantage of some Spring Boot features, such as:

- Standalone application – it allows to simply build the application executable file and run the application with no need to customize the deployment. The only environment that needs to be installed is Java JDK. In our case the only extra module which is added on the back-end side is PostgreSQL database.

- Embedded server – Spring Boot comes with several prebuilt application servers which do not require further installation to use. It also provides faster more efficient deployments resulting to shorting restart times. Our solution uses Apache Tomcat/9.0.39 servlet engine.

- Auto-Configuration - Spring Boot takes care of auto-configuration and it comes with so called annotations to control it. It attempts to automatically configure the Spring application based on the dependencies which have been added. This feature helps a lot, since a developer does not have to worry about manual configuration of the web server.

- Starter dependencies – Spring Boot provides dependencies designed to simplify the build configuration. It also provides complete build tool flexibility (such as Maven and Gradle). In our project we use Maven as a build automation tool. It allows to easily add necessary dependencies, perform unit and integration tests and build the whole project into one executable file.

2.2.3. Spring Boot dependencies

One of the advantages of Spring Boot framework is the fact, that is offers numerous dependencies and manages them automatically. Each release of Spring Boot provides a list of dependencies that is supported. The list below shows the most important ones which have been used in our application and explains why we decided to use them:

- spring-boot-starter-web – it is one of many starter dependencies available in Spring Boot framework. It is used for collecting all the dependencies needed for a web application to be running. This starter uses tools like Spring MVC (a Java framework which is used to build web applications, that follows the Model-View-Controller design pattern), REST architectural style and Tomcat as default embedded server.

- spring-boot-starter-security – a starter which adds Spring Boot Security, which is a powerful and highly customizable authentication and access-control framework. We decided to use this framework for security since it is a standard for securing Spring-based applications. It focuses on providing both authentication and authorization to Java applications and can be easily extended to meet custom requirements.

- spring-boot-starter-data-jpa – a starter which is used to efficiently connect the Spring application with relational database. The standard, old-fashioned approach of storing data in a database would involve writing SQL queries and setting up type and size of columns manually. In our application we decided to use more efficient approach which

uses Spring Data JPA and Hibernate.

The advantages of using JPA include reduced boilerplate code and auto-generated queries. Simple database queries can be automatically generated based on the method name. For example, we can define a method on our repository interface with the name that starts with findByEmail. When we define such a method, Spring parses its name and creates a query for it. In this case the result of such a query would be an entity with a given e-mail address.

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findById(Integer id);

    Optional<User> findByEmail(String email);

    Boolean existsByEmail(String email);
}
```

Figure 2.2: Sample repository interface for the User class

The Figure 2.2 shows a sample repository interface for the User class. We extend JpaRepository which uses two generics: User and Long. The User is the entity which is to be managed and Long is the data type of primary key. We define three methods inside the code of the repository. The methods findById and findByEmail will return an object of type User or null in case there is no matching entity in the database. The method existsByEmail will check if an entity with a given e-mail address exists and will return a proper Boolean value. Hibernate (which is a framework implementing JPA specification) will generate proper queries automatically.

- spring-boot-starter-websocket – another starter which allows to use web sockets. It is needed since our application uses them for the real-time messaging between users. Section 2.7 is dedicated to the instant messaging feature.

- jjwt – a dependency which is responsible for parsing a JWT token.

- postgresql – a dependency which allows to connect to a PostgreSQL database.

- gson – a dependency which allows to use GSON, which is an open-source Java library to serialize and deserialize Java objects to JSON.

2.2.4. Controller classes

Controller classes are responsible for processing incoming REST API requests. They have some public methods which control the data flow into model objects in the application. They execute application logic and then send a response to the client, containing some requested data. There are several controller classes in our Spring Boot application. They are listed and briefly described below:

- AuthController – a class which is responsible for user authentication. It contains methods needed for a user to register new account, log in or change password.
- UserController – a class containing methods which allow to get information about a user with given id number or change some profile information. There is also a method to delete user's account from the system.
- TripController – a class which is responsible for handling requests related to trips. It contains methods to get all trips of a given user, add a new trip and delete one.
- GroupController – a class which is responsible for handling requests related to groups. It contains a few methods such as getting a group by its id number, adding a group to a trip, changing the currency used by the group, applying and joining to the group, accepting and rejecting a pending user, leaving and deleting the group by a user.
- ChatController – a class containing two methods related to chats: getting all chats of a given user and adding a private chat between two users.
- ChatMessageController – a class responsible for handling requests related to messages in a chat. Contains a method for getting all messages of a given chat.
- LocationController – a class containing a method for getting a list of all countries on a given continent.
- BillController – a class responsible for handing requests related to bills. Contains methods such as getting bills of a given group, adding a new bill to a group, removing a bill and getting the sum of values of all bills in a given group.
- WebSocketHandler – this class handles the WebSockets events for the chat functionality. It handles text message inputs, WebSocket sessions and text message outputs. The class contains methods called when a client establishes a connection, closes the connection and when a message is received from a client.

2.3. React

2.3.1. React features

React.js is a JavaScript library that has a layer on top of the vanilla JavaScript. Unlike frameworks like Angular and Vue, React uses HTML inside JavaScript with JSX, a syntax extension to JavaScript. JSX opens many doors for us and produces React elements, also called components. We can create components as small as just a single button which we can then combine with different components and styles to form pages. Because React operates on a Virtual DOM we can create an efficient Single Page Application (SPA) using these components. SPA means that when a user opens our website, instead of receiving just one HTML document with all the content of the page, the user receives an almost empty HTML file, along with all the JavaScript content that is necessary not only for this particular page, but for the entire web application. Which of course, might mean that the initial load might take more time, however, this ensures that everything after the initial load will be much faster. In server-side rendering, any change to the page would result in a new connection to the server with a request for an updated page. The server would render and send a new page, even if the change was small, the whole page would be re-rendered. In SPAs in combination with Virtual DOM we gain a big efficiency advantage. Everything is already in the user's memory, thus no requests to the server are necessary when updating the page. Furthermore, because of Virtual DOM we only update the components that need updating instead of re-rendering the entire HTML Tree. In turn we are able to create a very interactive site while still keeping the load times short.

2.3.2. React dependencies

React.js is a library and not a framework, meaning it gives us great flexibility. A framework would keep us bound by its rules and functionalities. React gives us the freedom to choose from a wide variety of different tools and packages. And due to the fact that it is maintained by many developers and companies, most notably by Facebook, there are many valuable tools at our disposal. Using npm, a JavaScript package manager, we gain access to these open-source packages. In the list below we write about some of the most important ones:

- Babel - in short, a compiler. In our case it allows us to use JSX, since it transforms our code that uses JSX into a backwards-compatible JavaScript. This firstly, allows us to use JSX in the first place. Secondly, ensures that our application will run in older browsers by converting our code that takes advantage of newer syntax into something that older

browsers will understand.

- Axios – a very backward compatible promise based HTTP client. Axios is used for all HTTP requests to the back-end server. Since it is promise based we can make asynchronous calls, in certain situations resulting in a decreased loading time. A big advantage of axios is its automatic JSON data transformation, meaning we have one less thing to worry about.

- React Router - a library used to create routing in our application. It is used to display multiple views in our SPA. We define "routes" and we enter a specific URL that matches a "route" we defined, the user is redirected to that route, which usually simply means we show a different "page" to the user.

- WebSocket - a very important dependency for our application, more precisely, for our chat service. It is a communication protocol. We use WebSockets to establish a bidirectional communication between the client and the server. Having a two-way communication is crucial for a chat service. Instead of asking the server x number of times for new messages, we first establish a connection and then simply listen for updates from the server.

- React Testing Library(RTL) and Jest-DOM - used to write test for our application. The tests written with this library test the actual DOM nodes, mocking a user. Jest is a framework which we use along with RTL, together they provide the necessary tools to write tests.

- Bootstrap (Reactstrap) – Bootstrap is a HTML, CSS, JavaScript framework. It is used for its many features, among others it comes with many prebuilt components, Sass variables, and most importantly in our case, responsive grid system. Thanks to it we were able to make our web application usable on other devices, such as mobile phones. However, for React applications, Reactstrap is used because unlike Bootstrap it has no dependency on jQuery. Reactstrap provides a React implementation of Bootstrap.

2.4. PostgreSQL database

This section shows examples of how the information is stored in the database. Since we took advantage of Spring Boot JPA and Hibernate, we did not have to specify the names of tables, data types and the size of columns manually. Spring framework did this for us based on the information stored in the model classes.

An example of a model class ChatMessage is shown in the Figure 2.3. It contains variables and annotations specifying how the chat message objects should be built. The objects contain information such as ID number, ID number of a chat, ID number of a sender, timestamp and the text message itself. The Figure 2.4 shows how this information is stored in the database. Seven messages from one group chat are visible. We can see that the messages come from three senders (user IDs 1, 3, and 4).

Another example, shown in the Figure 2.5, is a table of users. It contains many columns; some of them are more interesting than others from the technical point of view.

For example the password is encrypted because of the security reasons. Even if the database gets hacked, the attackers will not be able to know the passwords of users.

Another interesting column is the one storing profile pictures of the users. They are stored as byte arrays but later are transferred between the front-end and the web server using Base64 encoding.

Some of the columns store only null values but the Spring framework requires them by default.

Figure 2.6 shows the information stored in the table of trips. Some columns store data in form of a list where elements are separated from each other using ';' character. We have added special classes to the Spring Boot application (i.e., IntegerListConverter and StringListConverter) which translate the list from the database column into a Java list object and vice versa.

```
@Entity
@Table(name = "messages", uniqueConstraints = {
        @UniqueConstraint(columnNames = "id")
})
public class ChatMessage {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @JsonProperty("id")
    private Integer id;

    @JsonProperty("chatId")
    private Integer chatId;

    @JsonProperty("senderId")
    private Integer senderId;

    @JsonProperty("timestamp")
    @JsonFormat(pattern="d-M-yyyy H:m:s")
    private LocalDateTime timestamp;

    @Column(name = "message", length = 4096)
    @JsonProperty("message")
    private String message;

    *
    * *** methods here***
    *

}
```

Figure 2.3: Part of the ChatMessage model class

	id [PK] integer	chat_id integer	message character varying (4096)	sender_id integer	timestamp timestamp without time zone
1	1	3	Hi!	3	2021-01-11 14:12:18.033156
2	2	3	Hey!	1	2021-01-11 14:14:10.800471
3	3	3	How are you?	1	2021-01-11 14:14:16.316701
4	4	3	Hi, I just joined	4	2021-01-11 14:14:49.029582
5	5	3	how is everyone doing	4	2021-01-11 14:14:58.312163
6	6	3	I'm doing well. I'm glad we can f...	1	2021-01-11 14:18:51.380752
7	7	3	So what are your ideas for the t...	1	2021-01-11 14:19:00.46912

Figure 2.4: Messages table in the database

	id [PK] integer	account_non_expired character varying (255)	account_non_locked character varying (255)	age integer	authorities character varying (255)	birth_date character varying (255)
1	1	[null]	[null]	24	[null]	8-10-1996
2	2	[null]	[null]	23	[null]	1-2-1997
3	3	[null]	[null]	22	[null]	4-5-1998
4	4	[null]	[null]	23	[null]	12-8-1997
5	5	[null]	[null]	24	[null]	1-2-1996

	chats_ids character varying (255)	connected_chat_id integer	country character varying (255)	credentials_non_expired character varying (255)	description character varying (255)
1	3	[null]	German	[null]	I'm a nice guy from Germany and wo...
2	[null]	[null]	Poland	[null]	I want to meet new people and travel ...
3	[null]	[null]	German	[null]	Looking for a group of friends who I c...
4	3	[null]	France	[null]	Hey, I'm Blanche from southern Franc...
5	[null]	[null]	Canada	[null]	I'm a travel photographer for a living, ...

	email character varying (255)	enabled character varying (255)	first_name character varying (255)	gender character varying (255)	languages character varying (1024)
1	user4@gmail.com	[null]	Christian	Male	German;English
2	user1@gmail.com	[null]	Jadwiga	Female	Polish;English;Spanish
3	user5@gmail.com	[null]	Krista	Female	German;English
4	user6@gmail.com	[null]	Blanche	Female	French;English
5	user2@gmail.com	[null]	Stefan	Male	Polish;English;French

	last_name character varying (255)	password character varying (255)	pic_byte bytea	pic_type character varying (255)
1	Hartmut	$2a$10$ywM6ww2yTVWVZ/...	[binary data]	image/jpeg
2	Kowalska	$2a$10$IYwge5KADcPxBweH...	[binary data]	image/jpeg
3	Eleonore	$2a$10$M5lR5pWCDv9osd3j...	[binary data]	image/jpeg
4	Soraya	$2a$10$539oPiFdp9NPPcu8v...	[binary data]	image/jpeg
5	Zielony	$2a$10$BpMOoX.ssOlkZIPPL...	[binary data]	image/jpeg

Figure 2.5: Users table in the database

2.5. User Experience and User Interface

A crucial part of any application is its User Interface (UI) and User Experience (UX). This is the bridge between the user and everything else. Having a user friendly interface can have a massive impact on the popularity of the service. As far as features go, it can have everything that a user might want out of it, but if it is difficult to navigate, displeasing to the eye, or slow,

id [PK] integer	activities character varying (255)	age_range_from integer	age_range_to integer	city character varying (255)	continent character varying (255)
1	1	Extreme Tourism;Agritourism;...	18	30	South America
2	2	Backpacking	18	32	South America
3	3	Extreme Tourism;Bungee jum...	18	30	South America
4	5	Bungee jumping;Road trip;Bac...	18	30	South America
5	6	Backpacking;Wine Tourism	18	38	South America
6	8	Backpacking;Clubbing	18	30	South America
7	9	Backpacking;Clubbing	18	30	South America
8	10	Backpacking;Clubbing	18	30	South America

country character varying (255)	date_from date	date_to date	group_id integer	group_size_from integer	group_size_to integer	group_type character varying (255)
1	Brazil	2021-02-18	2021-04-11	[null]	2	8	Mixed Group
2	Brazil	2021-03-01	2021-04-30	[null]	2	8	Mixed Group
3	Brazil	2021-01-11	2021-03-11	1	2	8	Mixed Group
4	Brazil	2021-01-11	2021-03-11	[null]	2	8	Mixed Group
5	Brazil	2021-02-20	2021-03-16	[null]	2	8	Mixed Group
6	Brazil	2021-03-01	2021-03-30	3	2	8	Mixed Group
7	Brazil	2021-03-01	2021-03-30	3	2	8	Mixed Group
8	Brazil	2021-03-01	2021-03-30	3	2	8	Mixed Group

location character varying (255)	name character varying (255)	user_id integer
1	South America;Brazil;	Brazil Trip	2
2	South America;Brazil;	South America - Brazil trip	4
3	South America;Brazil;	Extreme Brazil Trip	5
4	South America;Brazil;	Extreme Brazil Trip	5
5	South America;Brazil;	Brasilien Reise	1
6	South America;Brazil;	Brazil in March	3
7	South America;Brazil;	Brazil in March	1
8	South America;Brazil;	Brazil in March	4

Figure 2.6: Trips table in the database

it can very quickly drive possible clients away. There is a correlation between how quickly a new user becomes proficient at using the application and their satisfaction. If an application feels like a chore to use then it will not keep their users engaged for long. Due to this fact, a great amount of time was spent on planning user interaction paths and at times, even sketching graphical layouts of the application.

2.5.1. User Experience

User Experience design focuses on how a user will interact with the application and their experiences of it. The application should be designed as to decrease the initial learning curve. This means that the design should be simple and intuitive. Our approach was to use common practices and limit the number of any unique and creative design choices. This means that when a new user enters our application, most of the layout and functionalities will seem familiar and intuitive to use.

Included in this section are a few diagrams showing our planned structures for certain pages.

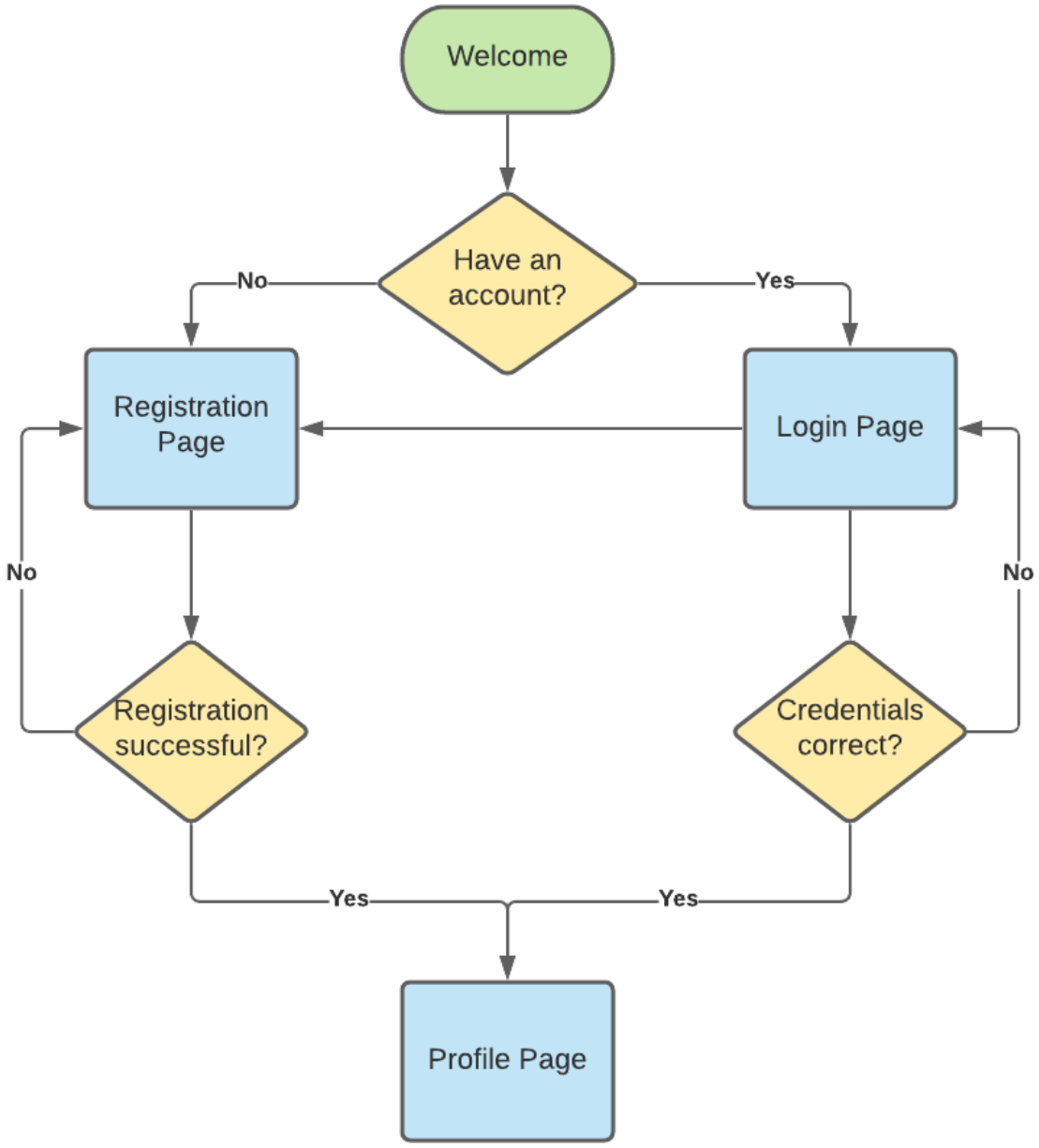

Figure 2.7: Flow diagram representing the flow and structure of the Welcome page

The Welcome page is the home page for all unauthenticated users. From here there are only two possible paths of action (Figure 2.7). Either a user may choose to register or log in. After either is successful the user is redirected to their Profile Page, which becomes their home page. Every user that has been authenticated has their own profile page as a home page.

The MyTrips page contains user's saved trips. This page is not as straightforward as the Welcome page, despite that, using conventional layouts we designed the page to be intuitive (Figure 2.8). Users with previous experience in other applications, will very quickly understand the page's structure and how to use it. If a user does not have any trips saved, the page simply

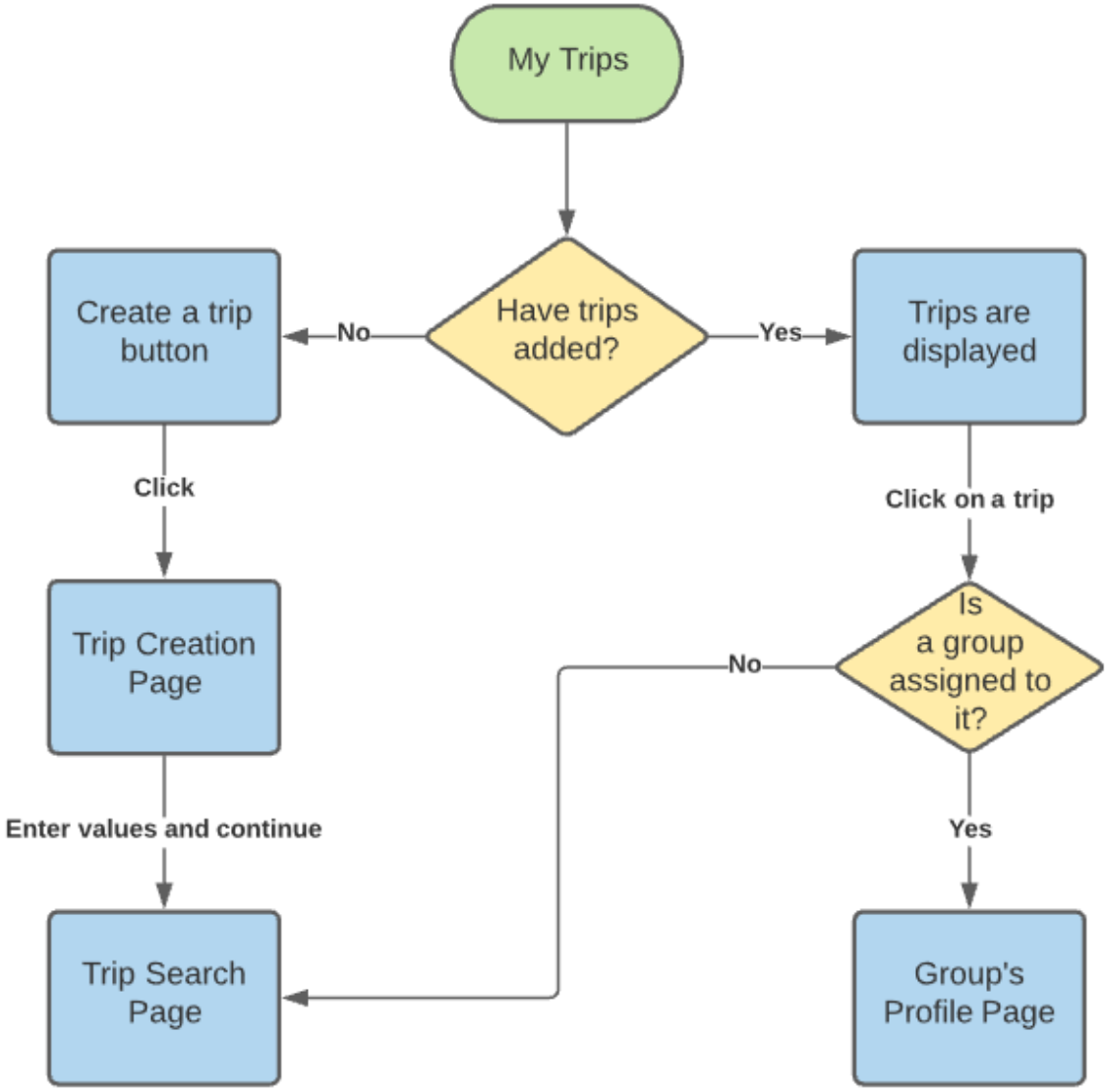

Figure 2.8: Flow diagram representing the flow and structure of the MyTrips page

contains information of this fact, and encourages the user to follow a link and start creating a new trip. On the other hand, if there are trips saved by the user, then they will be displayed on the page. Giving the user quick access to their trips and displaying basic information of each. Furthermore, the user can click on the trip's card and will be redirected to a different page. If the trip is assigned to a group then the user will be redirected to the group's profile page. Otherwise the Trip Search Page will be presented with the search form filled out with data from the saved trip that was clicked. We designed this as a shortcut, making it easier for users to go back and see if the search results changed, possibly seeing new matching groups. This increases the chances that users will form groups, since there is no need to repeatedly fill out the search from, thus making the service additionally convenient for users.

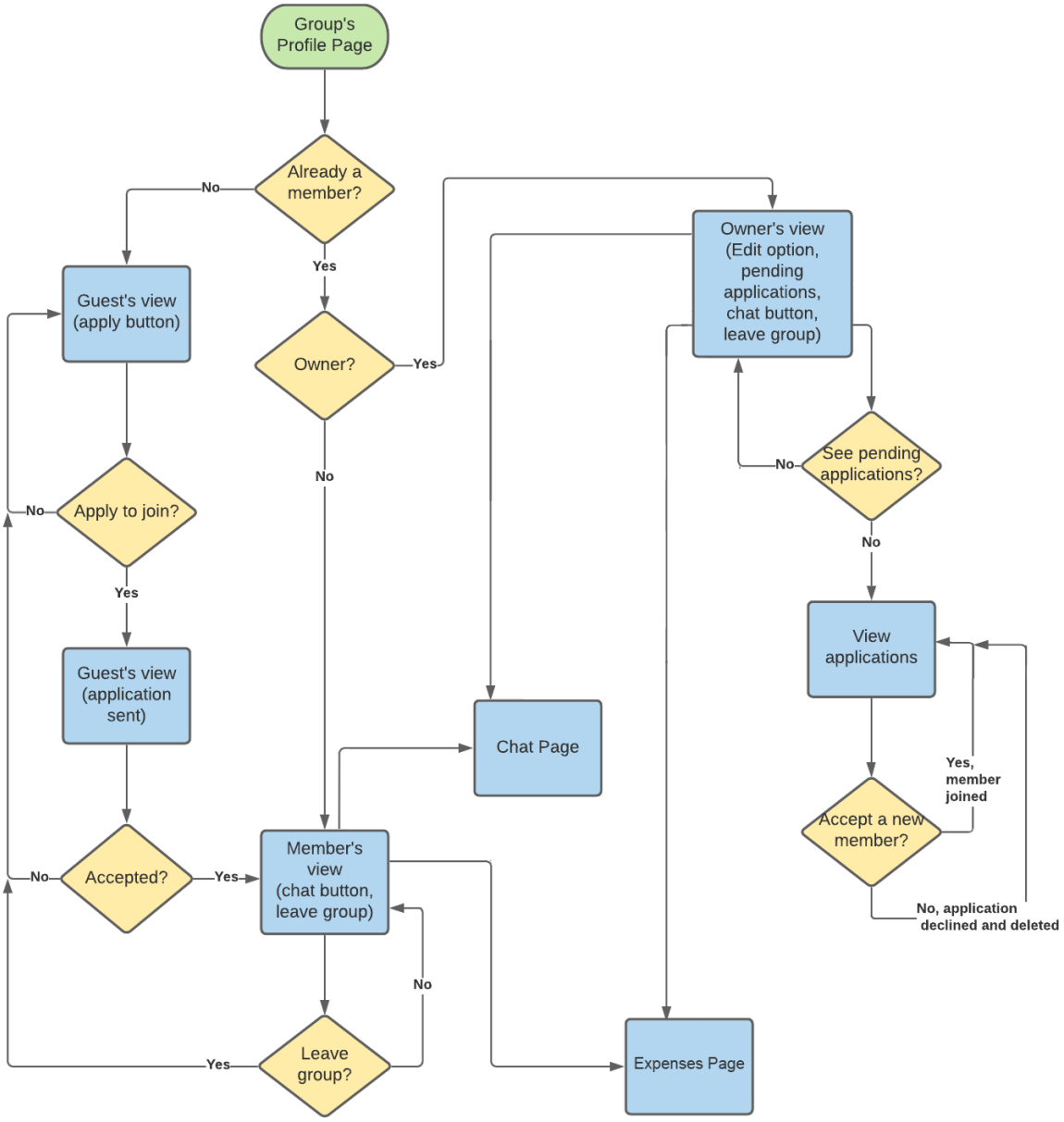

Figure 2.9: Flow diagram representing the flow and structure of the Group page

Pages with more functionalities and user interactions required more thorough planning. It is easy to quickly make the application appear complicated and burdensome to use.

The Figure 2.9 (flowchart) shows how we went about designing the structure of the group profile pages. Groups have a lot of to them and what a user can or cannot do also depends on their role in the group. Again we tried to go with a classic structure in such a way that users will know what to expect straightaway.

For example, when a user who is not a member of a group opens a group's page, a button asking the user to apply is shown. What happens after the button is clicked? In this case, the button's text changes and the button becomes inactive. Another option was to show a popup, or

after the button is clicked the page closes (which in reality, for a non-member is only a modal), meaning the search results are visible thus prompting the user to switch their attention now to other groups. Such seemingly simple and, at first glance, not important choices make up the UX. In our example of the button, whether the user is kept on the groups page or is showed back to the search results can have completely different outcomes. If the modal closes after a user decided to apply to a group, they will once again see the search results and possibly click on other groups. On the other hand, if the modal does not close, a user has to manually close the modal to go back to the search results and browse other groups. This one extra step might seem like an insignificant difference but it has been proved many times that people prefer the simpler, easier choices, and prefer not to put in extra effort. Meaning there is a higher chance that if the modal does not close, the user will get tired, bored, or simply lose interest in looking through any more results. This might seem counterproductive to our goal, however, the choice has been made with hopes that people will not look much further beyond the first group they applied to. The thought process was such that, we do not want people applying to many groups for the same trip. This might result in a user being accepted to multiple groups for the same trip, thus the same time period. This might then cause more trouble to not only the user but also to the groups. The user would have to then again go through the groups they joined and decide which one to stay in and which to leave. This will also cause a mess inside the groups. And what if the user will not bother leaving the groups? The group will have just accepted an inactive member.

In conclusion, every decision has its consequences, and thorough planning was necessary when designing the UX. Even seemingly small choices might add to Interaction Cost and while designing this should be kept in mind in order to try to manipulate or control user behaviour in a favorable way.

2.5.2. User Interface

When talking about User Interface we mean the visual side of things. Meaning the graphical layout, color choices, typography, etc. As developers we struggled with this part. In the end, we decided to focus on simplicity because making more things look pleasing to the eye is harder than just focusing on a few. Thus our UI is simple, without many icons, borders or transitions.

We began by sketching layouts of pages, an example of such a sketch is Figure 2.10. We favored an approach in which we focus on a particular aspect of the design at a time. In the example mentioned above, we were designing the layout, meaning we did not worry about the colours or even the exact components that will be included. Figure 2.11 shows the final product.

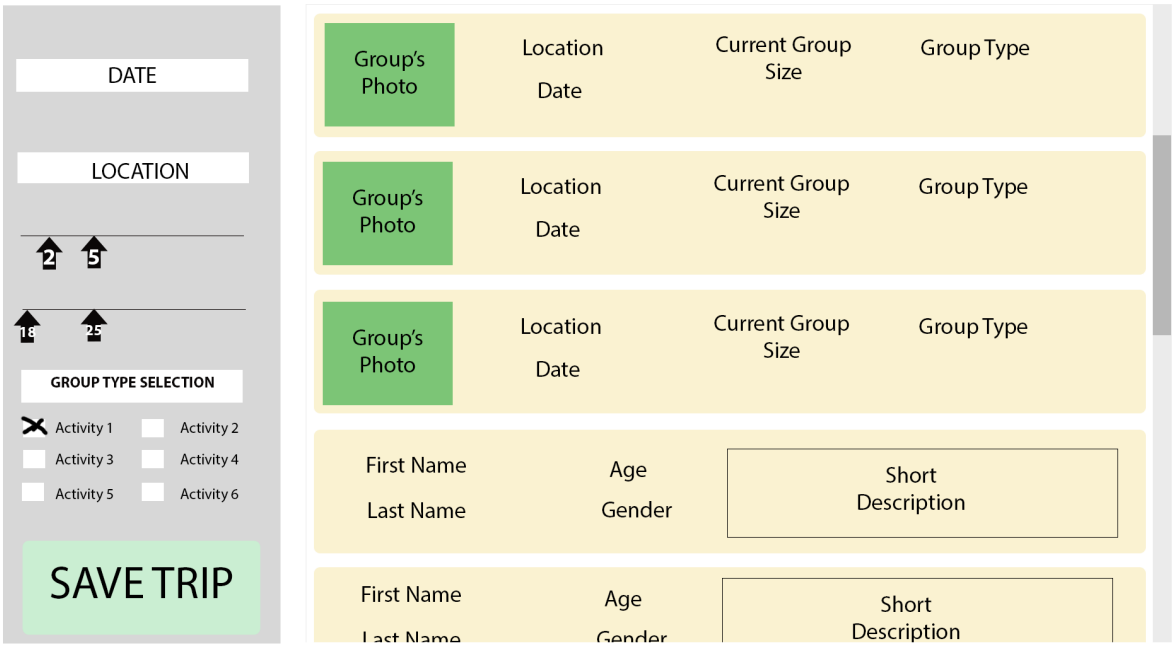

Figure 2.10: A final sketch version of the search page

It is clear that we went based off of the initial sketches but ultimately made adjustments as we saw fit.

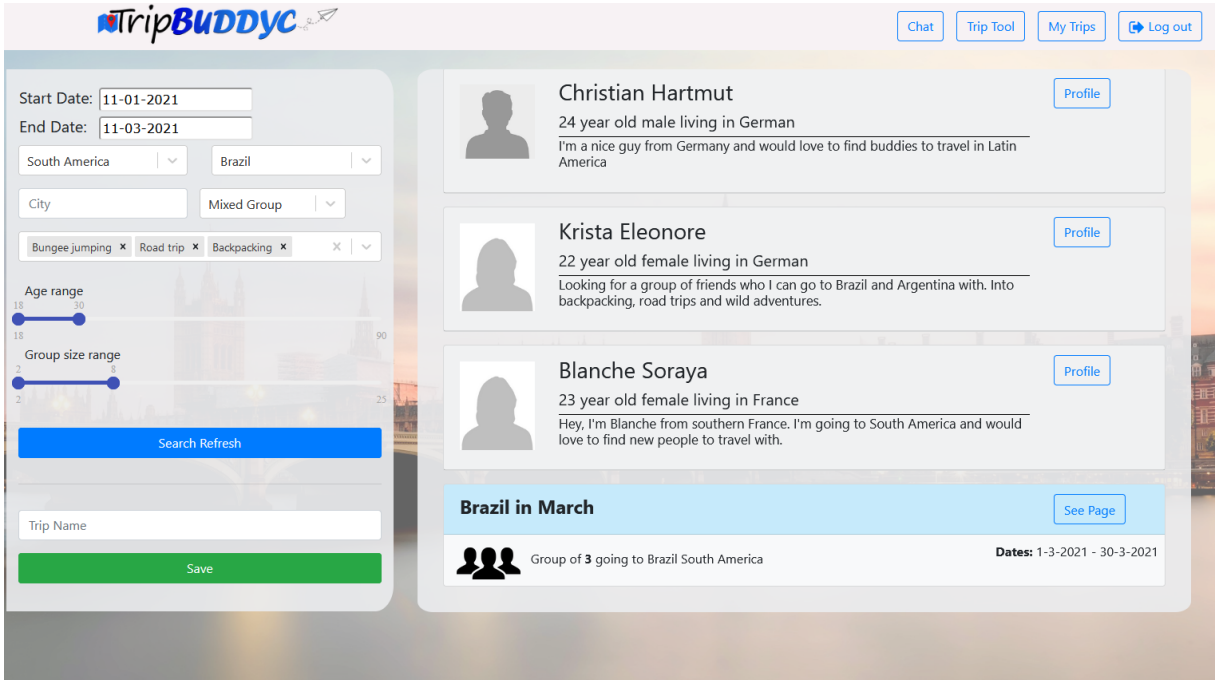

Figure 2.11: The final version of the search page

As mentioned previously, we went for simplicity in the graphical design. Alongside being easier to design for non artistic people, we also believe it gives the application an "open" look.

If we inserted more items into the design, it would give the user many stimuli. We agreed on a "open" and "refreshing" style. There are not many things to look at beside the key components thus there are no distractions and it is hard to get tired of looking at the page. We also decided to made the components semi-transparent and insert a strong background. As backgrounds we used photos of popular tourists destinations, in hopes of inspiring users to travel more.

Before we went about designing the UI we designed the UX, in turn giving it priority. At times we forfeited "good looks" for a better user experience. One such case is seen in the Figure 2.11. We opted out of using user's profile images in the search results. Which does not look as well as with the images. However, it has an important purpose - it serves as a deterrent. Once again, trying to manipulate user behaviour by adding an additional step. The profile pictures can be seen by simply clicking the user's card. Our goal was to make it harder to judge people by their looks or skin colour, and hopefully create travel groups without discrimination. Additionally, we wanted to deter any users that wish to use our service as a dating site.

2.6. User authentication

To authenticate users, we decided to use JSON Web Token authentication. It is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. JSON - JavaScript Object Notation - seems to be the right choice when working with a front-end written in JavaScript. And of course, the information contained in the token can be trusted because it is digitally signed. Our application signs the token using a secret with the HS512 algorithm. The token itself is a string of characters composed as follows:

Header.Payload.Signature

- Header – contains information about the algorithm which has been used to encode the token and how the token should be interpreted.
- Payload – contains user's data and additional information (in our case these are user's e-mail address, the time the token was issued at and the expiration time).
- Signature – contains encoded header, encoded payload, information about the coding algorithm and the secret.

The Figure 2.12 shows an example of a JSON Web Token. The left side shows the encoded version, and the right side shows the decoded equivalent. One can clearly see the information that is contained in the token.

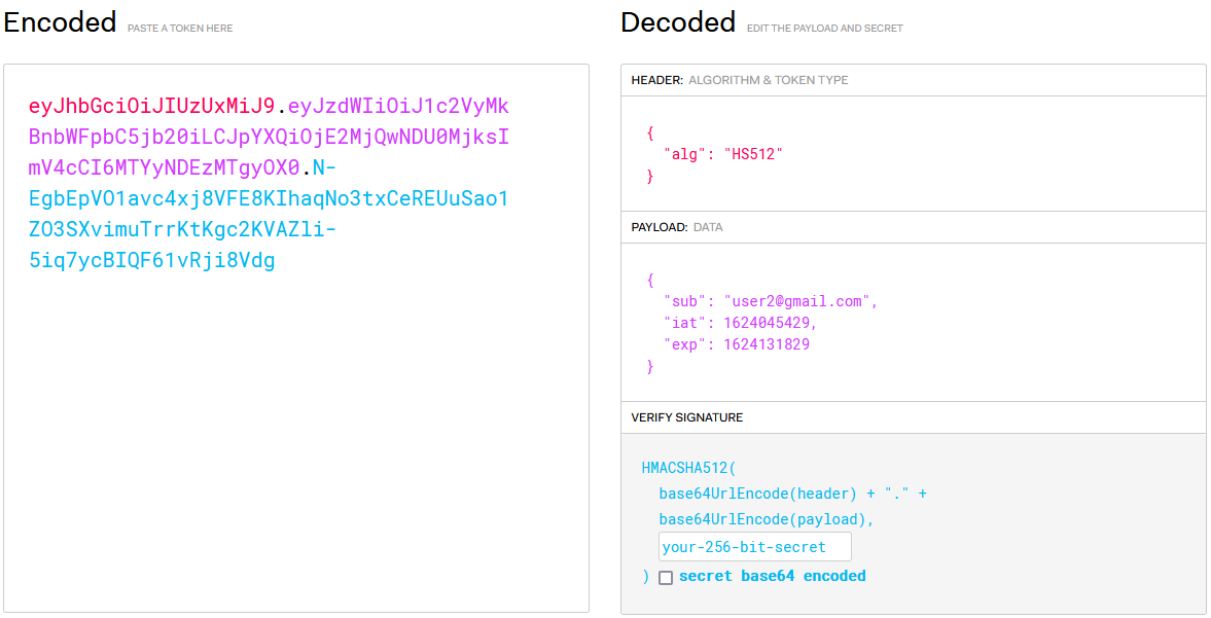

Figure 2.12: A screenshot from a website https://jwt.io/

The Figure 2.13 (activity diagram) shows actions taken by the client and the server when a user registers, logs in or wants to access a protected resource.

The Figure 2.14 (activity diagram) shows the procedure of accessing a protected resource. Firstly, the client sends an HTTP request to the web server with a JSON Web Token included in the header. The Token is received while logging in and is stored on the Client's side. Then it is retrieved in order to attach it to the header of HTTP requests. Then a class JwtAuthenticationTokenFilter checks whether the request has a token and then extracts it. In the next step a class JwtUtils validates the token. If nothing goes wrong, then a class UserService loads details of the user based on the e-mail address taken from the token. Then, the JwtAuthenticationTokenFilter class is responsible for setting up Spring Security feature for the user. After it is completed, the web server can send the requested resource to the client.

2.7. Instant messaging system

Our application provides two types of chats for the users. A user can have a private conversation with another person or a group conversation with other members of a group he belongs to. The application allows to have a real-time communication. When a user selects a chat from the list of chats displayed to them, they can instantly exchange messages with other members of that chat.

To make it possible we used WebSocket communication protocol. It provides full-duplex

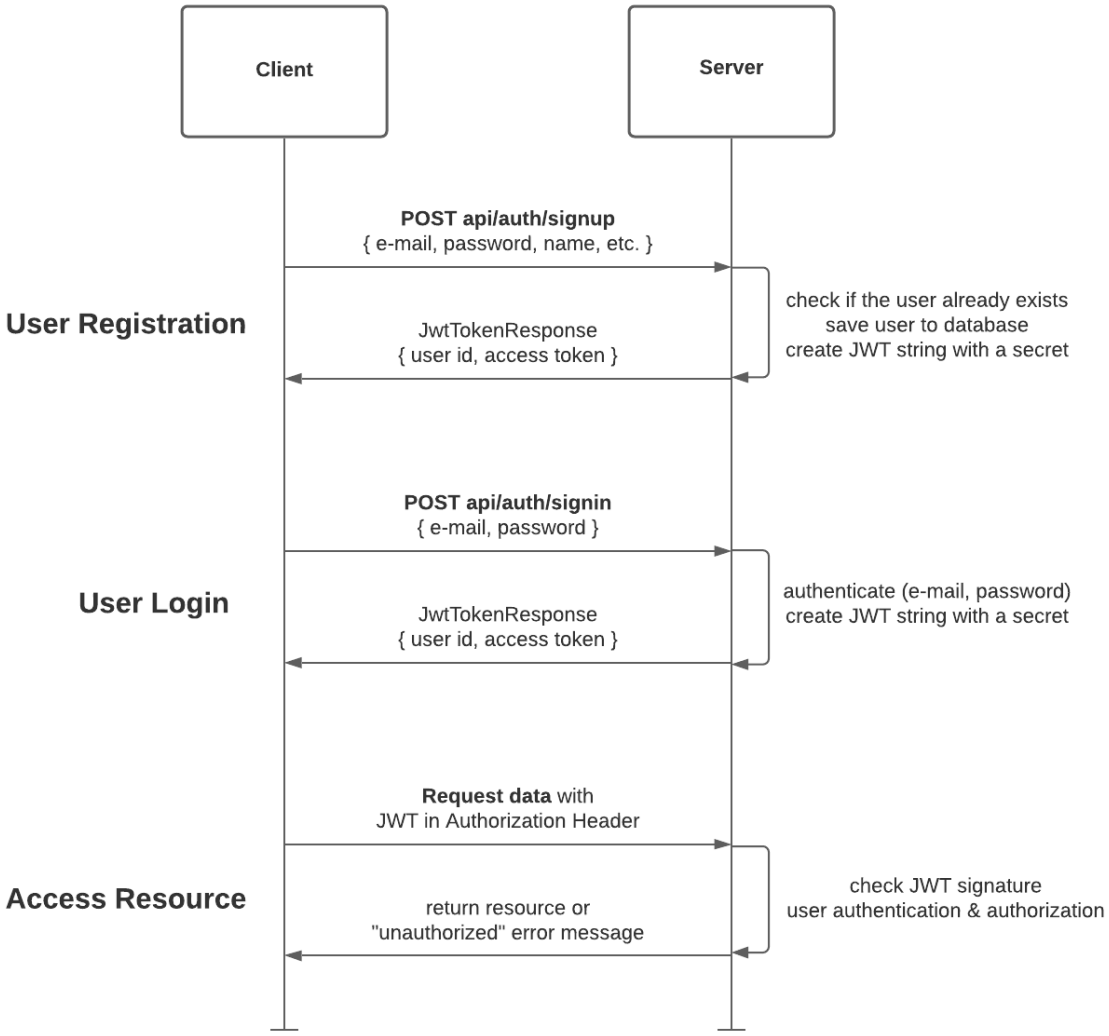

Figure 2.13: Actions triggered in case of user registration, login or a resource access request

communication channels over a single TCP connection between a user's browser and the server. This technology allows to exchange information between both sides of the connection without having to poll the server for a reply (as it is needed with HTTP protocol).

When a user opens a chat in the web browser then a persistent connection between the browser and web server is established. After the user sends a message, it is received on the server side. The server sends the message further to all the members of the chat who currently have the chat opened in their browsers and are connected to the server using web socket.

Obviously, members of the chat who do not have the chat tab opened in their browsers at the time when messages are sent, can see these messages later. After they open the chat tab, all the messages will be loaded (using a HTTP request).

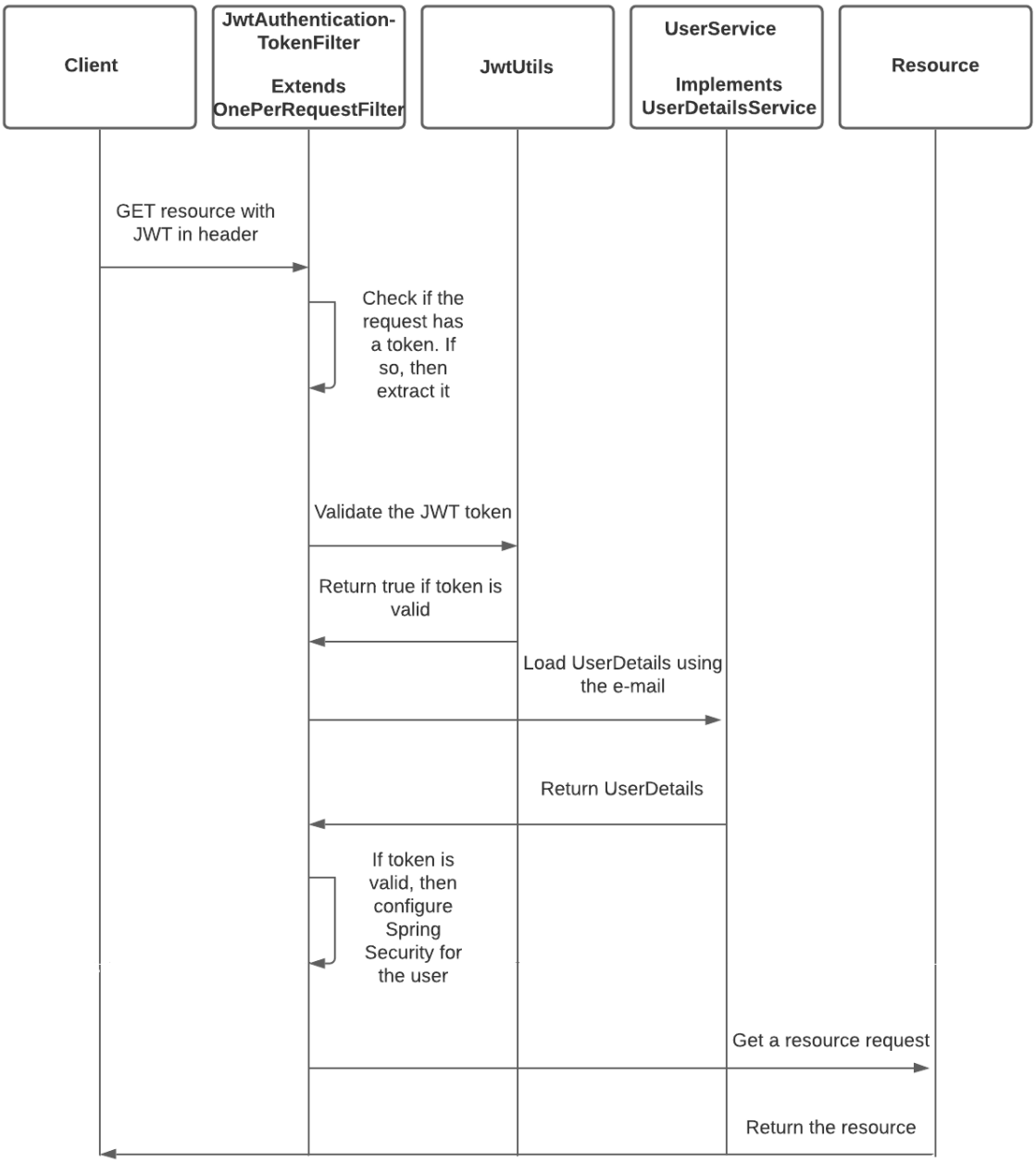

Figure 2.14: Action diagram showing the procedure of accessing a protected resource

Before we started implementing the messaging functionality, we had to choose a protocol for the exchange of messages. There is a popular messaging protocol called STOMP (Simple Text Oriented Messaging Protocol) which specifies ways on how the messages are exchanged between the client and the server using WebSockets. We decided not to use it because of its complexity. We concluded that we need a simpler solution and made a decision to make our own protocol. The team made an agreement of how the information will be structured and how each side (a client and the server) will send and receive it.

The information structure of a single chat message is based on JSON format and contains the following information:

- Type of the message – there are two types. The first one is the initial message which is needed because of technical reasons. After a server receives this type of message it saves information about a new WebSocket session to the memory. The second type is normal message, which stores a text sent by the user.
- Token – a JSON Web Token must be sent in each message because of the security reasons. The chat must be secured in a way that an unauthenticated person cannot send a message.
- Chat ID – the identifier of the chat must be present.
- Message – a string representing a text message sent by a user.

There are three controller classes handling requests coming from clients. These are WebSocketHandler which is responsible for handling events related to web sockets, ChatController which takes care of requests related to chats and ChatMessageController which has a method to retrieve all messages stored in a given chat.

In order to make the system work, WebSocket package was added to our front-end. It is an almost pure JavaScript implementation of the WebSocket protocol and gives us access to all the necessary classes and methods needed to communicate with the sever. It provides us with important functions: *send*, *onmessage*, *onopen*, and *onclose*. The *onopen* event listener is called when the connection is first established. Then we can use *send* to send messages. The *onmessage* event is triggered when a message is received from the server, we can then display it to the user. Lastly, *onclose* is called when the WebSocket connection is closed.

3. Analysis of the solution

3.1. Deployment into production server

For the production server we decided to use an Ubuntu virtual machine running on Microsoft Azure platform. We chose this platform because it offers free credits for educational purposes. To make sure the non-functional requirements such as high responsiveness and simultaneous usage by hundreds of users are met, we use a virtual machine of a size "B2s". It has access to 2 virtual CPUs, 4 GB of RAM memory, and 8 GB of data storage on a premium SSD disk. In case our web application needs more resources to run smoothly in the future, it will be easy to switch to another virtual machine offering more capacity.

3.1.1. Docker

Instead of installing the environment for React, Spring Boot and PostgreSQL database directly on the virtual machine, we decided to host our web application using containerization. For this, we agreed to use the most popular Docker platform, which is a tool designed to create, deploy, and run applications by using containers. Using this modern way of deployment has many advantages over the old-fashioned one, such as:

- Portability – if a containerized application works correctly on one machine, we can be sure that it will work correctly on all other machines with Docker installed.
- Isolation – running containers work in complete isolation from each other. We can easily have many containers running different versions of our application in the same time.
- Scalability – by using a tool called Docker-Compose it is easy to manage multiple containers. We can specify the order in which the containers run. This feature was important when we wanted to run a container with the database first, and then run a container with the web server, after the first one has instantiated correctly.
- Performance – in general Docker containers are faster to create and quicker to start, comparing to their alternative, which is a virtual machine. Their performance is better since

they do not contain an operating system but just the environment needed to run an application.

To run our diploma project, we need to use 3 containers. One for the React application, one for the Spring Boot application and the last one for the PostgreSQL database. The database container runs constantly, and we do not shut it down unless some changes in the database structure are required. To run the other two containers, we take advantage of the Docker-Compose tool. A part of the file needed to run these containers is shown below:

```
services:
    backend:
        image: 'rathumbel/tripbuddyc-back:latest'
        container_name: backend
        network_mode: bridge
        ports:
             - 8080:8080
        environment:
            - SPRING_DATASOURCE_URL=jdbc:postgresql://172.17.0.2:5432/readydb
            - SPRING_DATASOURCE_USERNAME=postgres
            - SPRING_DATASOURCE_PASSWORD=password
            - SPRING_JPA_HIBERNATE_DDL_AUTO=update
            - DEMO_APP_JWTSECRET=secretKey


    frontend:
        image: 'rathumbel/tripbuddyc-front:latest'
        container_name: frontend
        network_mode: bridge
        ports:
            - 80:3000
        stdin_open: true
```

In this file we specify:

- images which should be fetched from our DockerHub repository
- names for the containers
- necessary ports to be exposed - to be publicly available from the global internet

- type of the network
- some necessary environmental variables (e.g., credentials to log in to the database)

A screenshot from the production server console is presented in Figure 3.1. We can see that 3 containers are running.

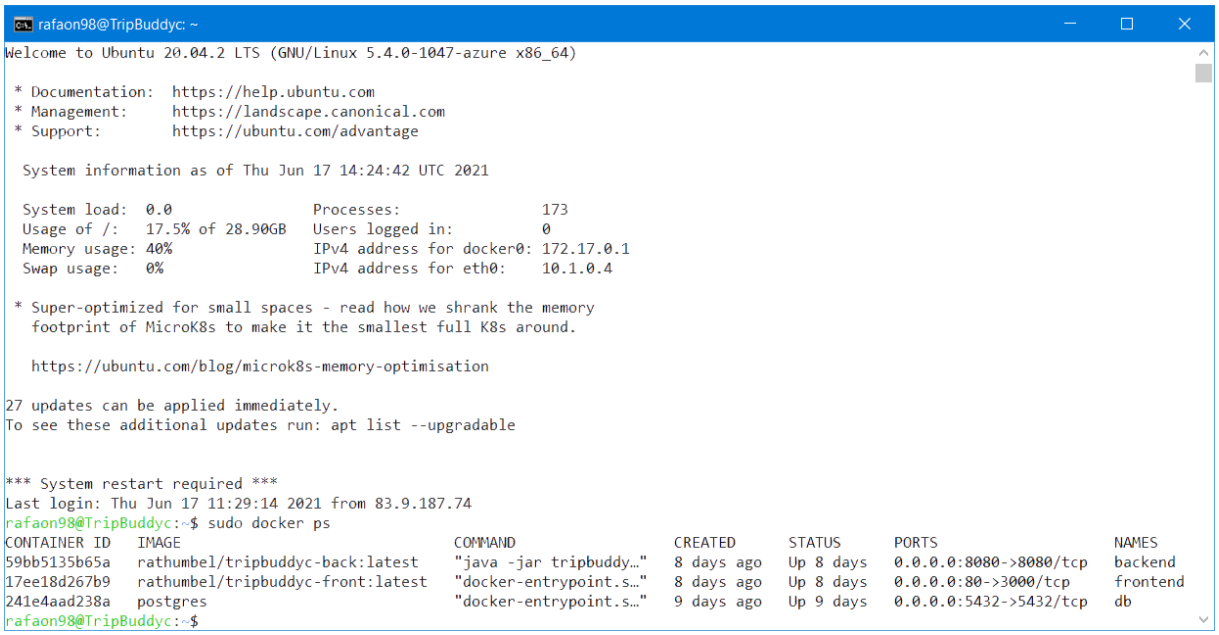

Figure 3.1: Three containers running in the production server

3.1.2. GitHub Actions

After we make some changes in the source code of the application, we commit and push them to our GitHub repository. Then we take advantage of a process called Continuous Integration and Continuous Deployment. For that, GitHub offers a feature called GitHub Actions. We use it to build Docker images for both React and Spring Boot applications and then to push and run containers from these images on the production server. The images are build based on the instructions stored in Dockerfiles. Figure 3.2 shows the steps of the GitHub Actions workflow. In order to take advantage of GitHub Actions we needed to create a .yml file in a specified directory in the repository.

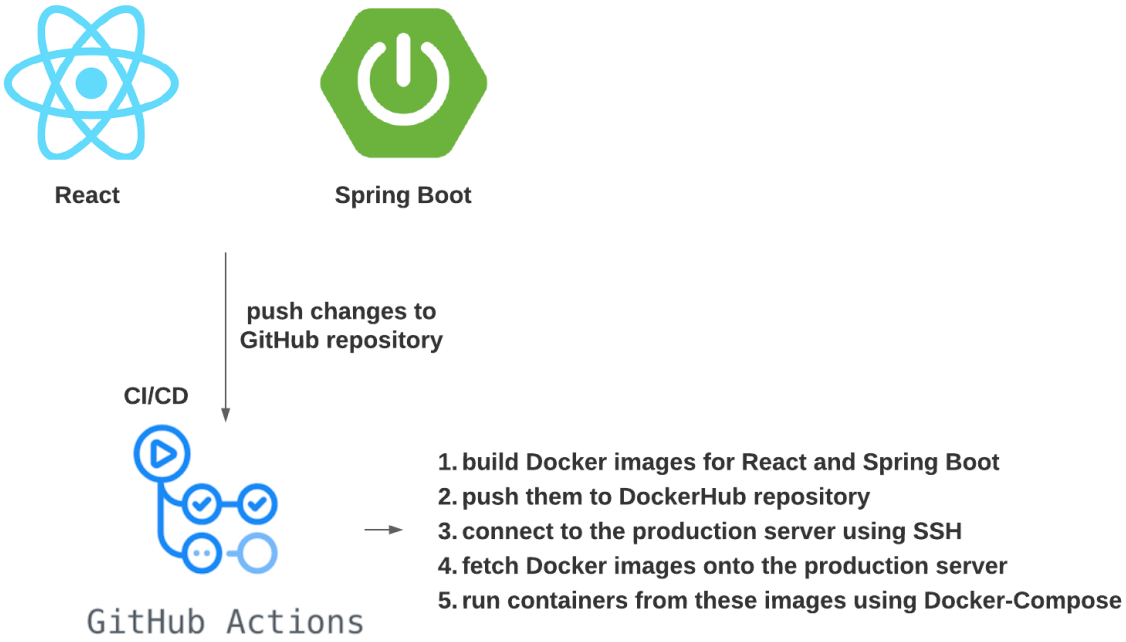

Figure 3.2: Steps of the GitHub Actions workflow

3.2. Development tests

During the development time of the back-end of our application, we have written numerous tests which were helping us to control the behavior of certain methods and modules. The advantage of spending time on writing them is that the testing process could get fully automated. Each time after making changes to the source code, we could quickly check if the web server still works fine before we committed the changes to our repository.

We have focused on unit and integration tests. Unit tests were responsible for testing individual modules of the application in isolation (without any interactions with other modules and dependencies). To check whether all the modules could cooperate correctly with each other, we had to write integration tests. The integration tests cover most of the functionalities of the web server.

The integration tests cover all the controller classes and their main functionalities. In total there are 77 unit and integration tests checking if the web server works properly after every change in the source code. For the testing purposes we have used JUnit 5 testing framework.

3.3. Fulfilling the functional and non-functional requirements

During the development phase, we managed to implement all the functionalities from the functional requirements list. The application also meets the non-functional requirements:

1. The application is safe. For the user authentication, the Spring Boot Security framework together with JSON Web Token has been used. It is a standard in securing Spring based applications.

2. The application is responsive and potential shutdowns are minimized. Since the application is deployed to the Microsoft Azure platform, we rely on its stability. Microsoft prides itself on the Azure platform, so we can be sure that it is a high-quality service. Potential shutdowns should happen only during the maintenance time, when new versions of the front-end or the back-end applications will be deployed into the production server.

3. The application is able to store data of thousands of users. Currently the storage capacity is limited to 8 GB, but it can be easily extended if needed.

4. Hundreds of users can use the application simultaneously. The production server has access to 2 virtual CPUs and 4 GB of RAM memory, so it should handle that many users using the service at the same time. If we notice that there is any delay or the server is lagging, we can easily upgrade the virtual machine on Azure and provide more resources.

5. The application is easy to maintain and extend. We can easily add new modules and functionalities to both React and Spring Boot applications and easily deploy new versions into the production server.

6. The application is available in English language.

7. The application has user-friendly interface. It is a subjective opinion of the team, but we find the user experience friendly. We have also asked other people to give feedback about the user interface and the feedback was positive.

4. Examples of use case scenarios

4.1. Welcome

Figure 4.1 shows the home page for all unauthenticated users. From here it is possible to log in or to register. To log in users use their email addresses and passwords which were entered during registration.

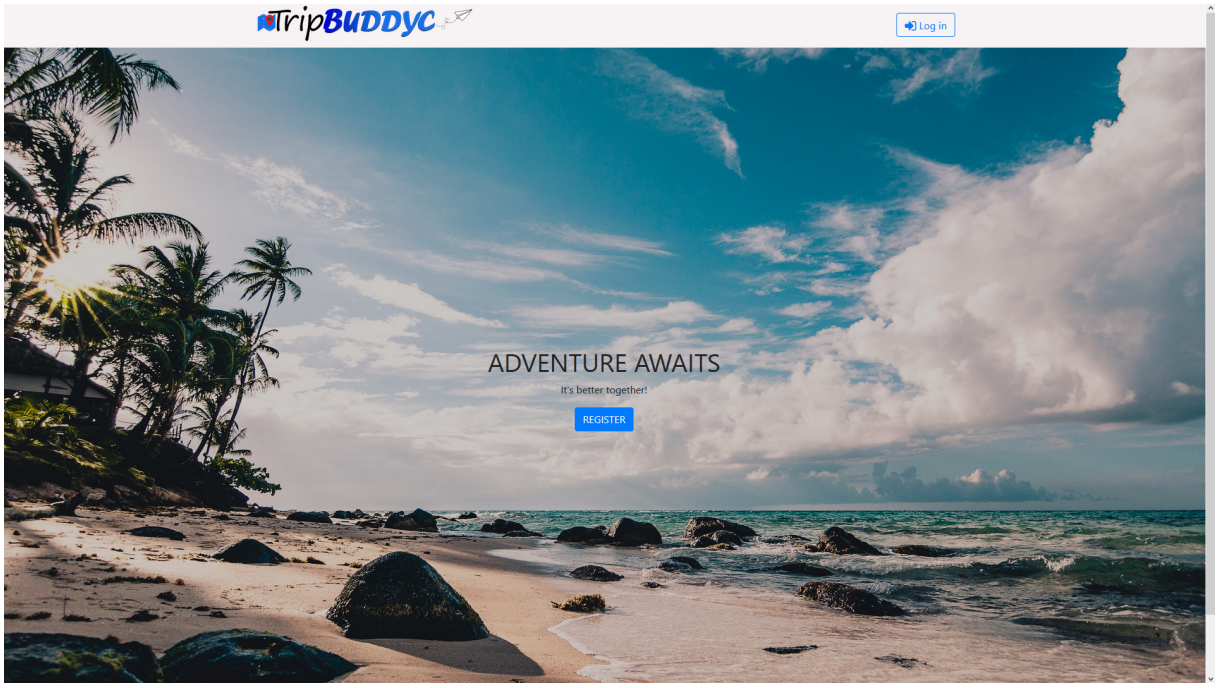

Figure 4.1: Welcome page of our web application

4.2. Profile

After logging in or creating a new account, the user is authenticated and their profile page becomes their new home page. The profile page, as shown in Figure 4.2, is simple but contains all the information about the user. Most of the information displayed on the page was required during registration, without most of this data it would not be possible to accurately match users.

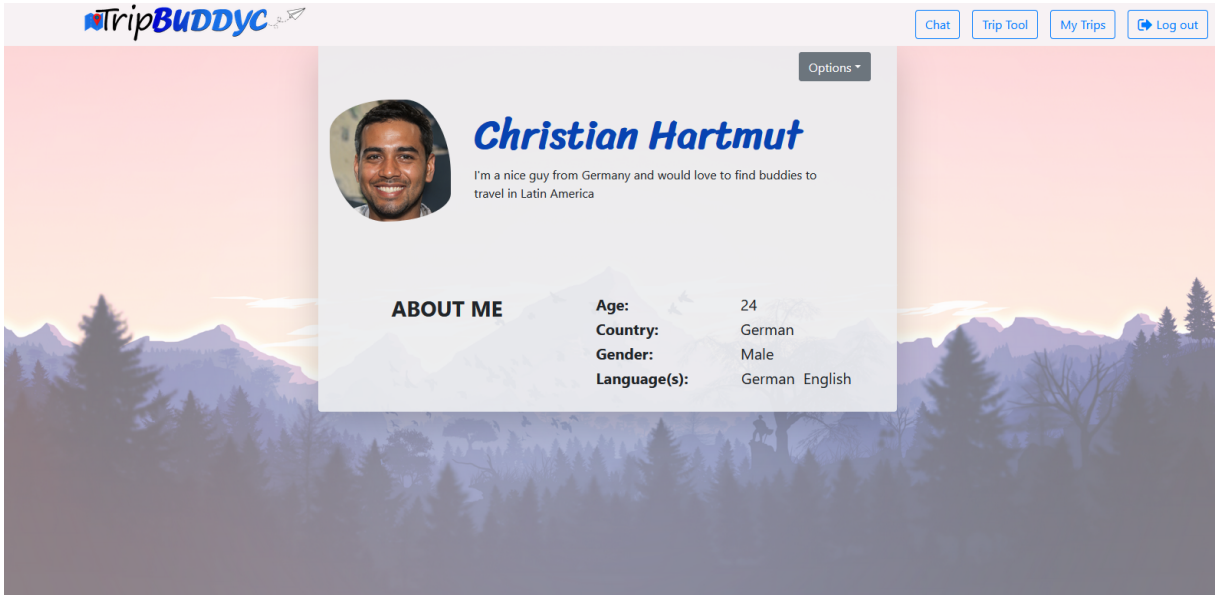

Figure 4.2: An example of a profile

Each user while on their own profile has a dropdown menu with all their profile and account options, visible on Figure 4.3.

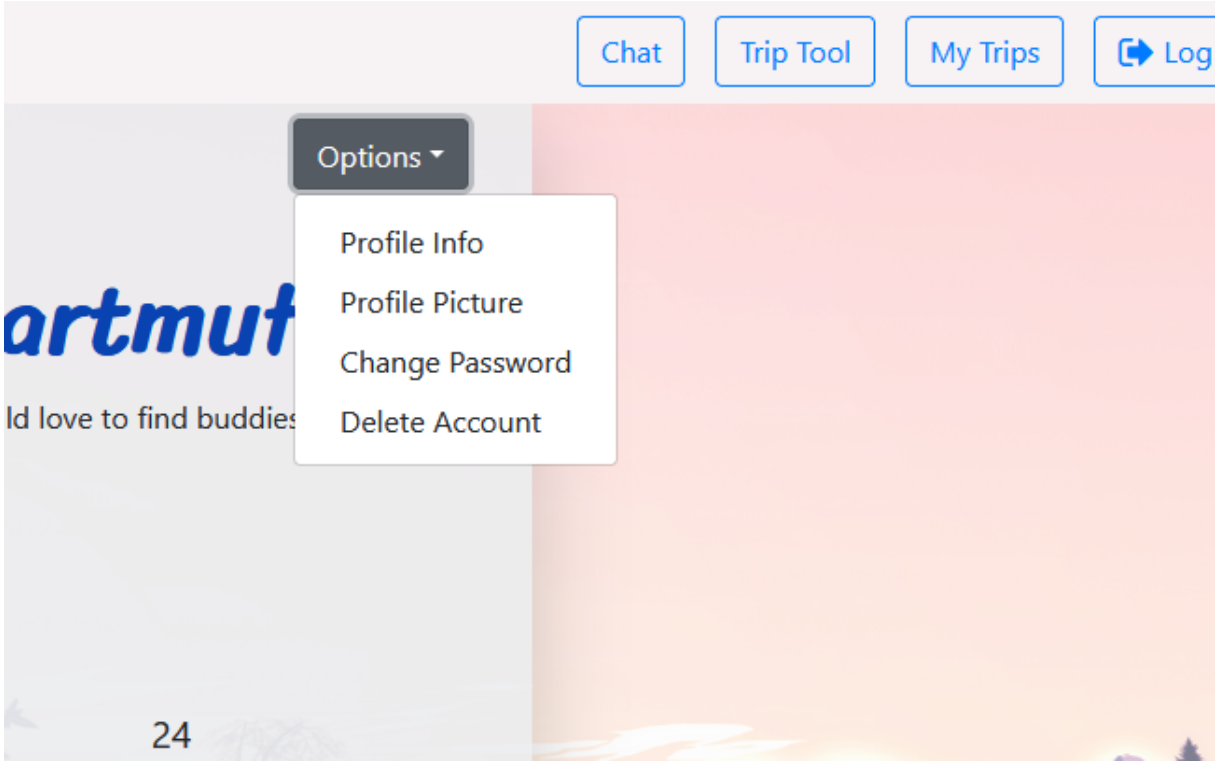

Figure 4.3: List of options on a profile

4.3. Trips

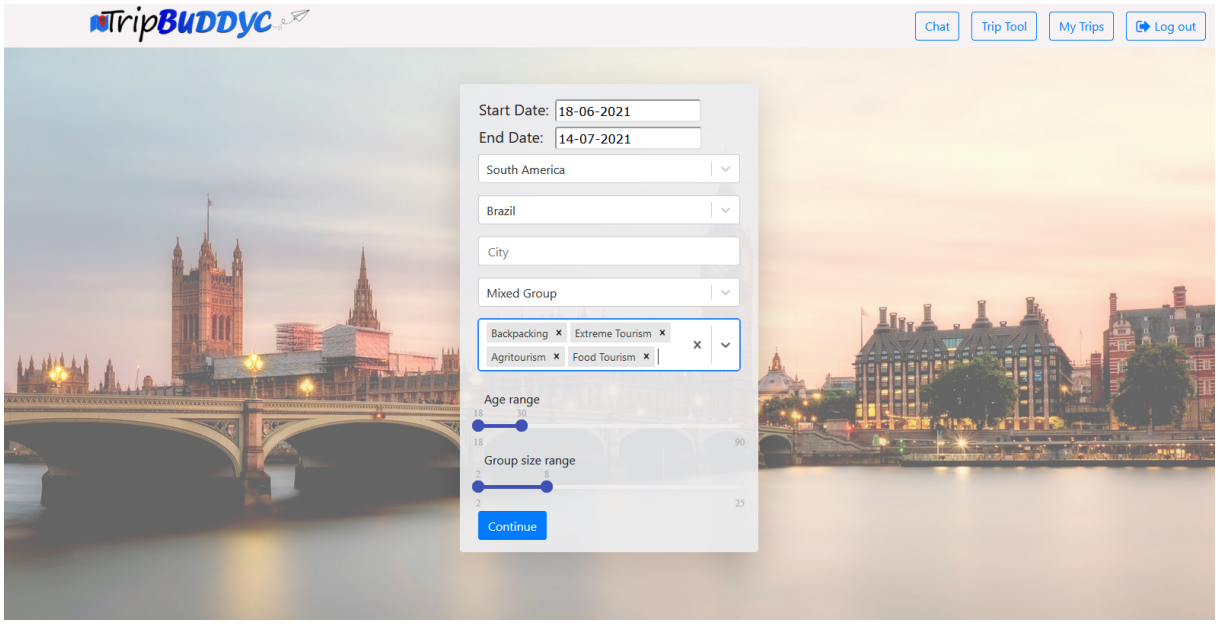

Figure 4.4: Trip creation tool form

In Figure 4.4 we can see the form for creating a new trip. This is accesses easily through the "Trip Tool" button on the navigation bar. Other than the country and city, all other data is required in the form. A user will fill out the form with the information matching their travel plans and then simply click continue.

On the next page the results are shown, if there are any matches. In Figure 4.5 we can see there are multiple users and a group visible, meaning our trip matched with these people and this group. What is not visible on the provided image is that, the container with the users and groups is scrollable and in reality this trip matched with more groups. At this point the user will look through the matches. It is possible to peek the profiles of both the groups and of the users as shown in Figures 4.7 and 4.6. If the user isn't satisfied with the results, it is possible to save the trip information and later try again or the user can then create their own group. Another option is to alter the trip's information and refresh the search results.

If, after opening someones profile, the user is interested in traveling with that person, there is an option to instantly start a chat conversation with that person.

In case the user is interested in a certain group, he can click the "See Page" button as visible in the Figure 4.5 and there, on the group's profile page is a button "Apply", as shown in Figure 4.7. The button changes after it is clicked, letting the user know their application is pending. The group may then choose to accept or deny the user from joining their group.

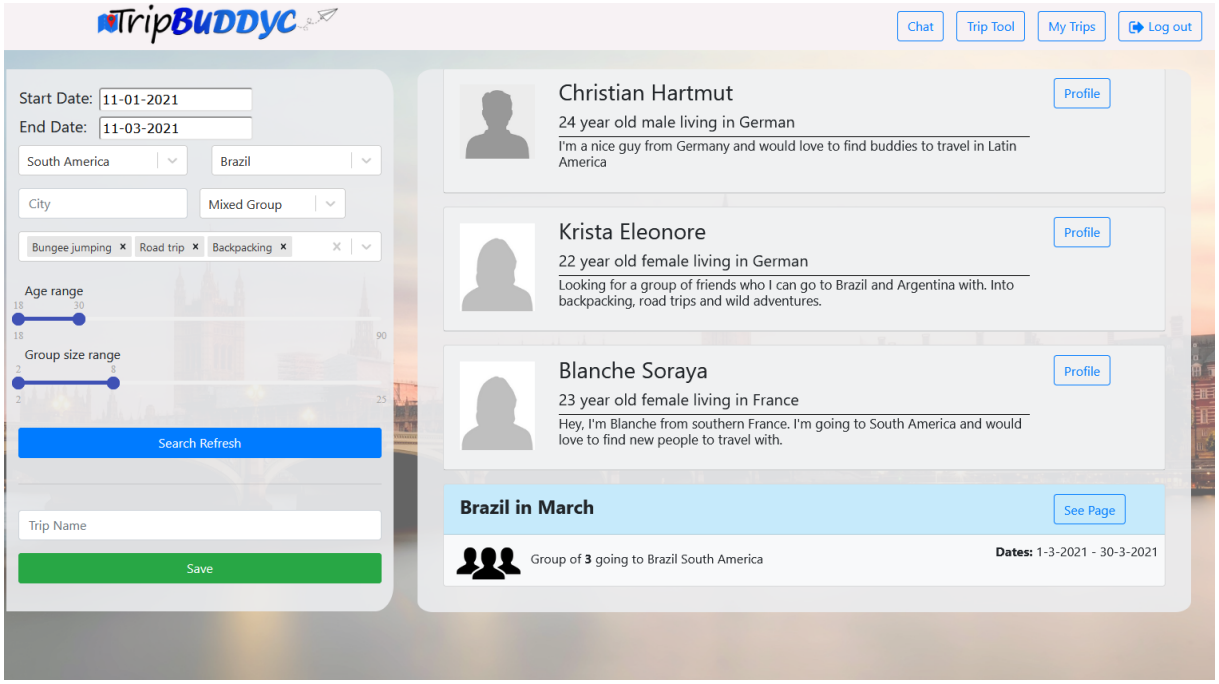

Figure 4.5: Example results of a search

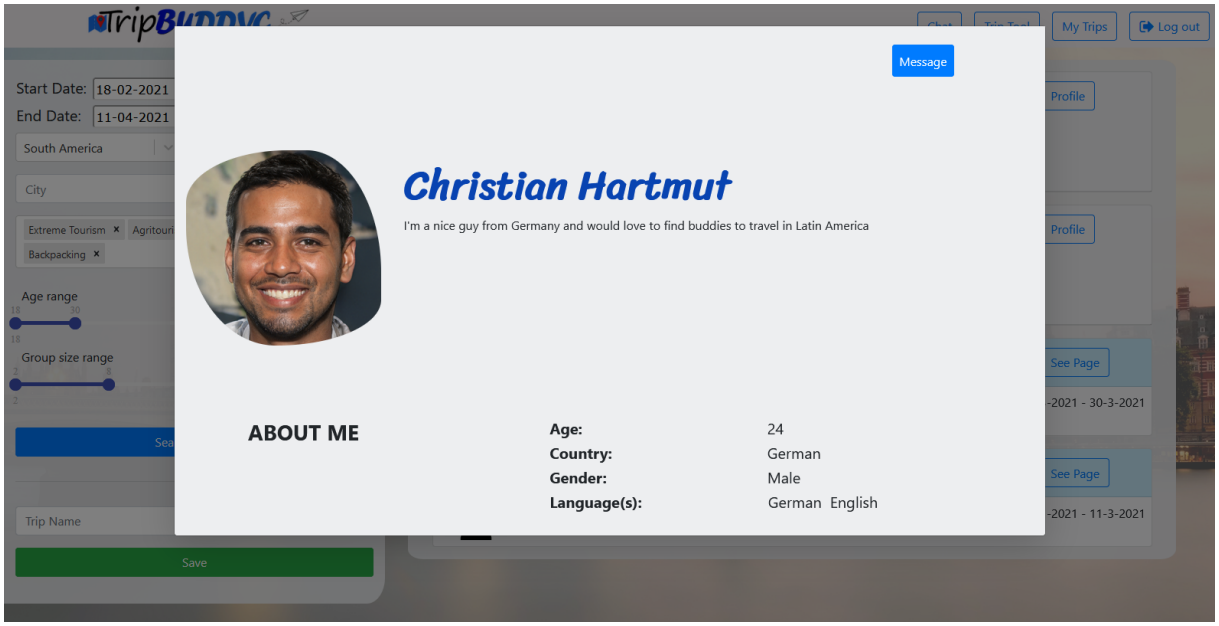

Figure 4.6: Profile modal

Another button on the navigation bar is the "My Trips" button. The My Trips page is almost like the center of the application. Here all the trips that the user has saved, or has joined a group to, are visible. An example of this is Figure 4.8, and it can be seen that we have two types of trips here. To one of them, we have a group attached, and the other trip is not connected to a group. It can be seen by the difference of the text in each of the buttons. This page has been created with functionality in mind. Each trip card is clickable, however, what the click does

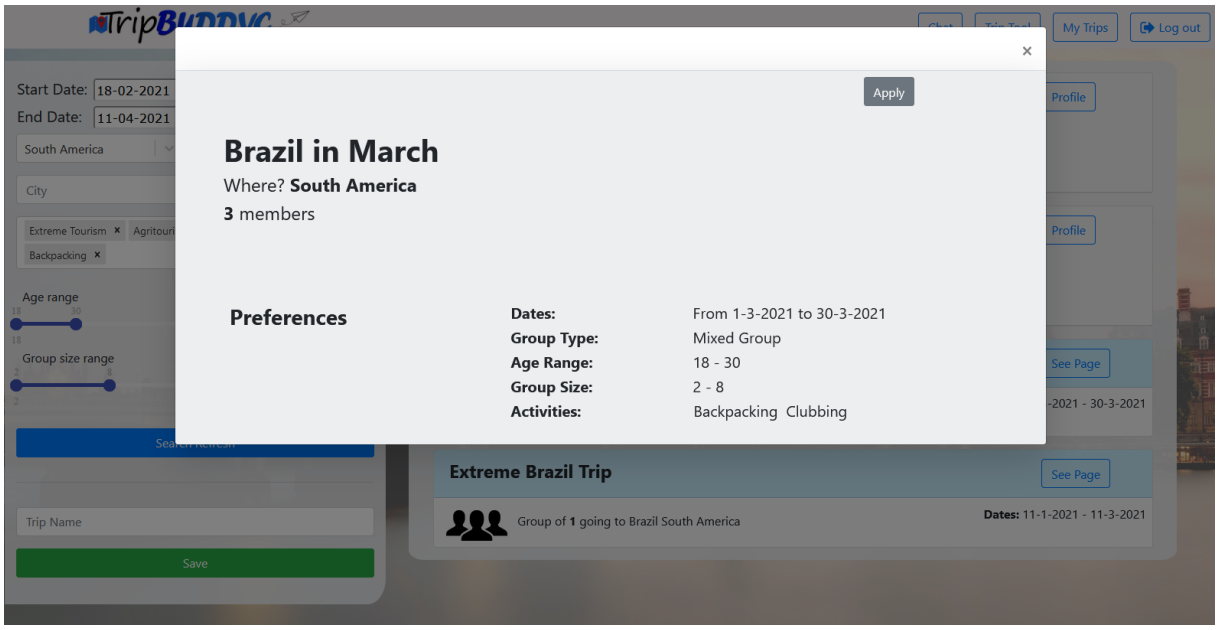

Figure 4.7: Group modal with "Apply" button

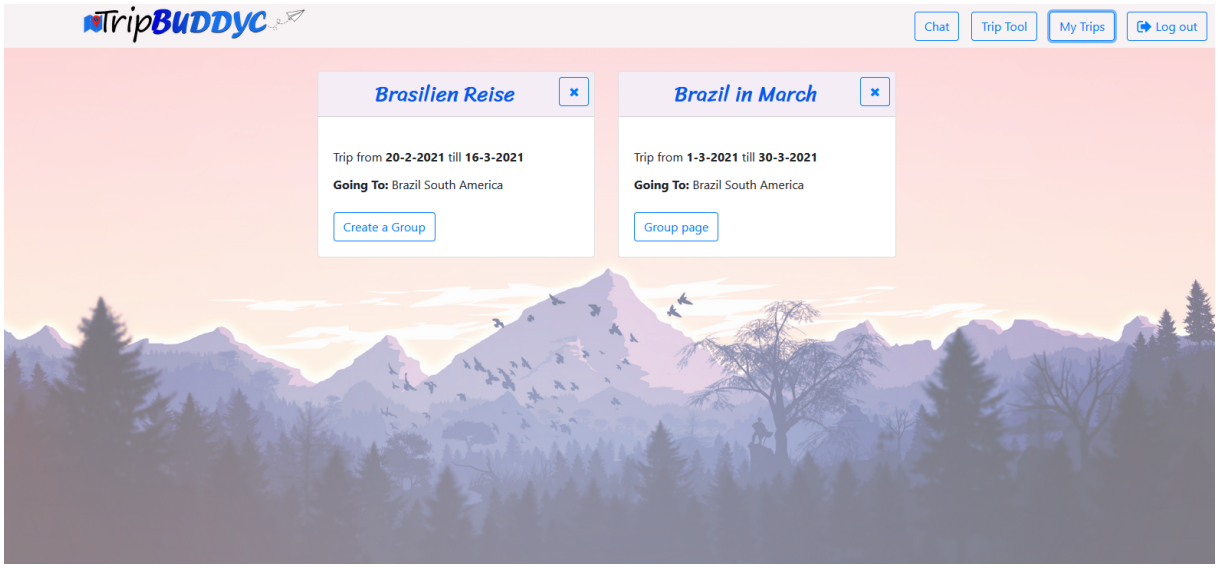

Figure 4.8: An example MyTrips page

depends on the trip type. If the trip has a group then clicking on the card accomplishes the same as clicking the button - redirects to the group's page. On the other hand, if the trip still is without a group, meaning the user has not found any trip partners yet, clicking on the trip card will redirect the user to the search results. Meaning the user can easily come back and check if any new matches were found. This type of trip card has a button that will let the user create his own group if he chooses to do so. He will then be the group owner and can then decide who will join the group.

4.4. Groups

Groups are the next stage in finding travel buddies. After creating a trip, forming a group is the next goal. As it was mentioned before, groups have their own profile pages. What options are there depends on the user's role in the group. The Figure 4.9 shows the view as the owner of the group. The owner, on top of the features that each member sees, has a an option to see all the pending applications. In the example image we can see that the dropdown menu "Options" has a "(1)". This is a notification to grab the owners attention and direct him to go to the pending applications view. In the pending applications, as seen in Figure 4.10 , there is one user waiting to be accepted. The owner has the choice to either accept or reject the user. To aid the owner in his decision, the waiting user's information and a button to their page is visible. The owner may choose to contact the applicant by starting a chat conversation with them. The owner may also choose to talk to the rest of the group and come to a democratic decision. Although in the end, the decision is all up to the owner.

Each group has their own chat conversation which can be accessed by the "Chat" button in the

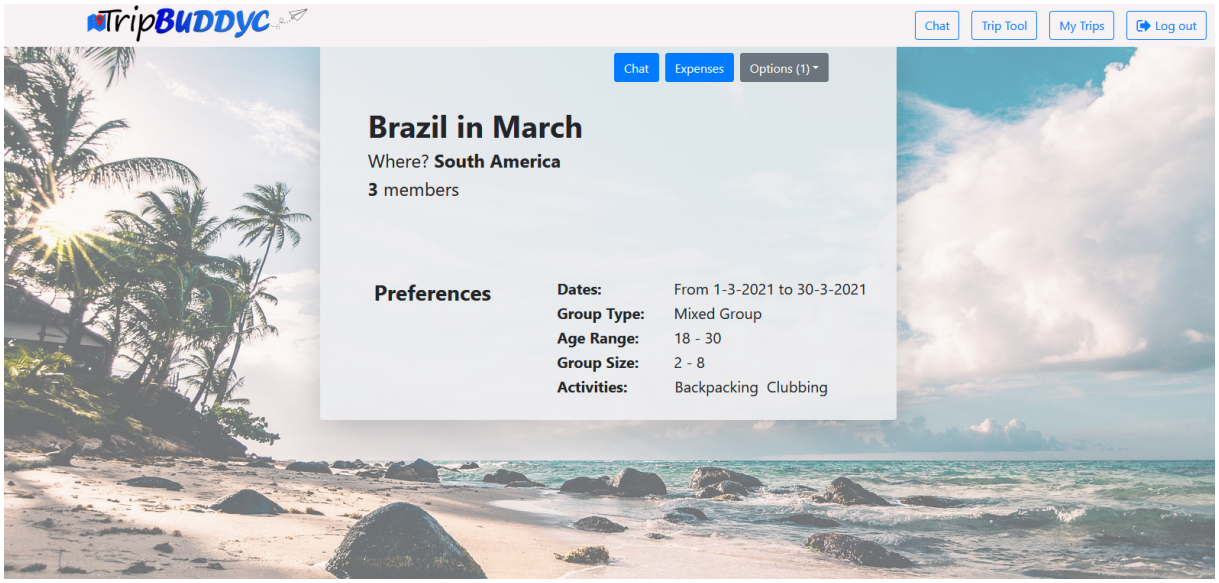

Figure 4.9: An example owner view of a group's profile page

navigation bar and then selecting the group chat or by clicking the "Chat" button on the group's page, and this will redirect the user to the chat page with the group chat already selected.

Another feature of groups is the expenses page, Figure 4.11. After the owner initializes the page, by selecting a currency, he may then add expenses to the list. Each member has access to view this page. The total is automatically calculated and an estimate of how much each member will pay is also displayed. The idea is that members of the group will discuss expenses through

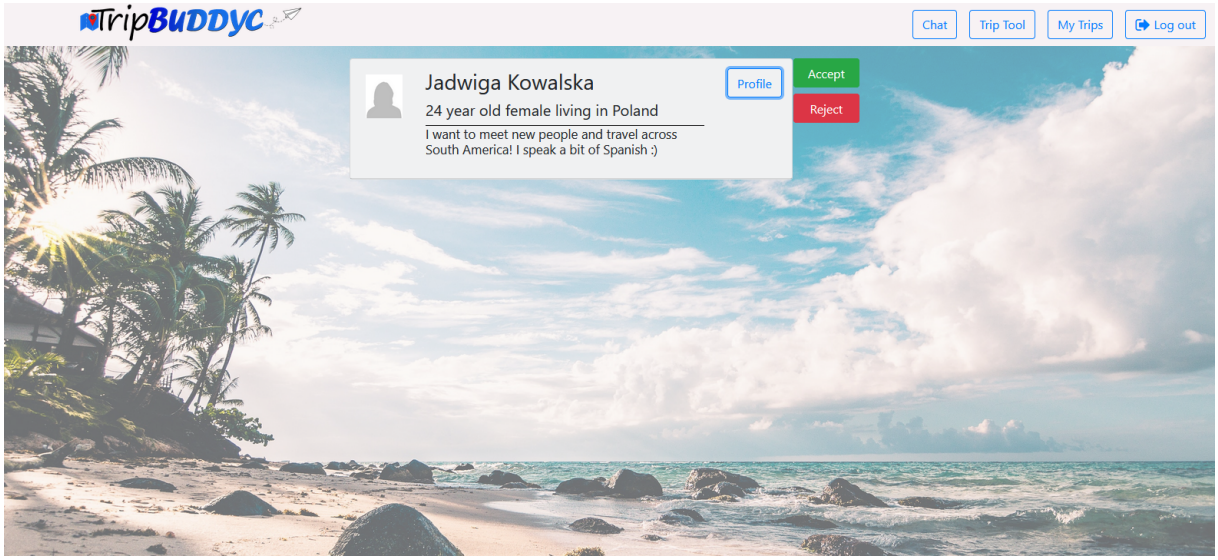

Figure 4.10: List of pending applications

the chat, and then add them to the expenses list. Thus helping the group organize themselves and make everything clear.

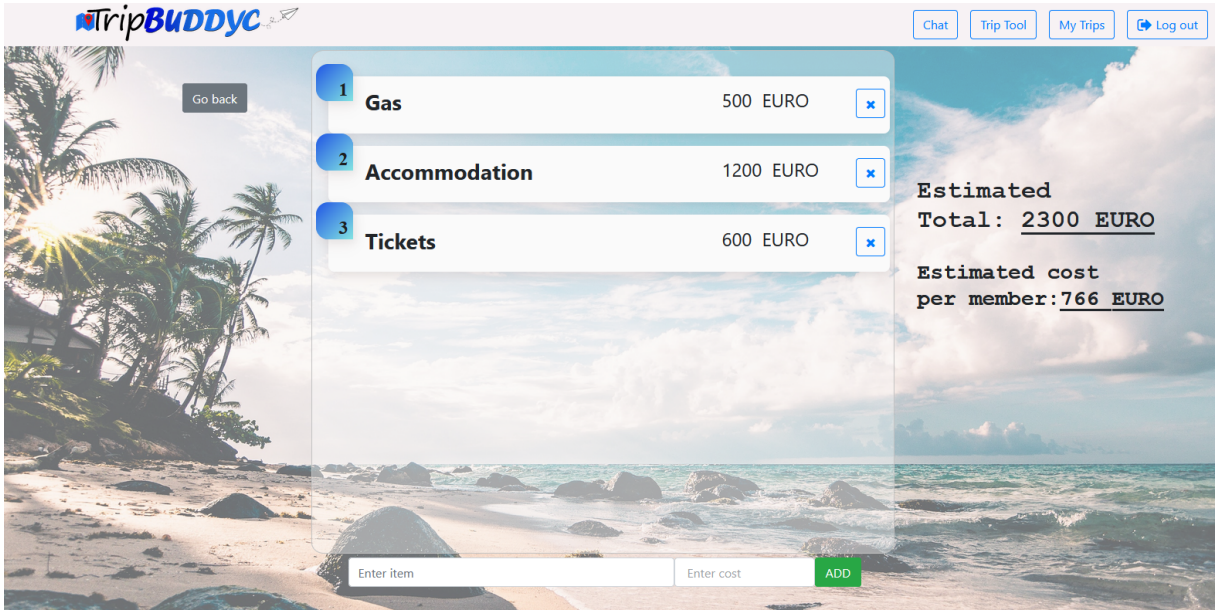

Figure 4.11: An example of an expenses list

4.5. Chat

The use of the chat service is very straightforward. The chat can be accessed via the navigation bar or buttons on user's and group's profile pages. User needs to select a chat from his list of recent chats, it can be seen on the left side of Figure 4.12. There are two types of chats, private

and group chats. In Figure 4.12 we see an example of a group chat, and in Figure 4.13 the user started a new private conversation. Private conversations are named after the user we are conversing with, and group chats are named after the group.

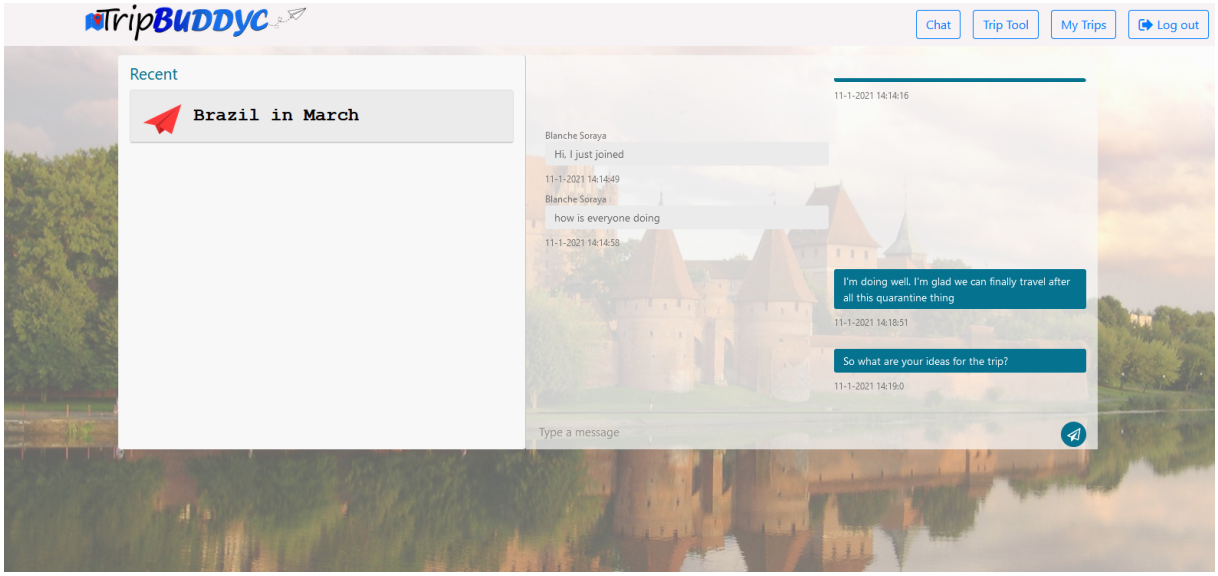

Figure 4.12: A sample group conversation

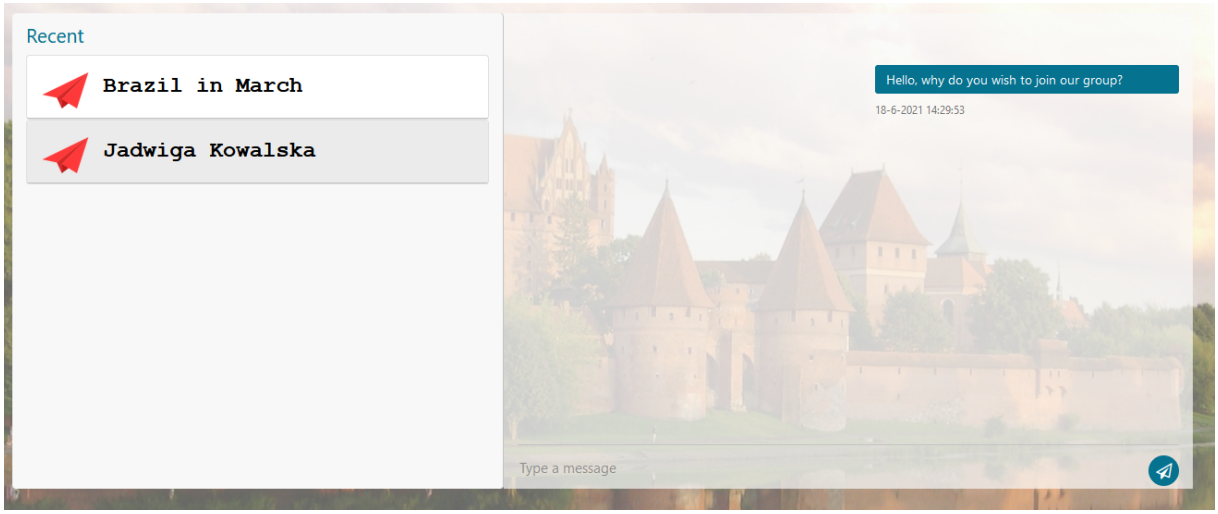

Figure 4.13: A sample private conversation

5. Conclusions

We have developed a fully functioning web application. It is not a perfect tool. It has its weaknesses, but it solves a problem that was highlighted at the beginning of this thesis. We believe that putting a bit of extra work to add some minor features, could make this application well recognizable on the market of web applications designed for travelers.

5.1. Possible improvements

We have listed some features and functionalities which could be added or improved in the future:

- Dedicated mobile application – if we wanted to publish the application for commercial use, a dedicated mobile application would be necessary (for both Android and iOS systems). It takes time to develop such an application, so we would need a third person in the team to make it for purposes of the diploma thesis.

- PUSH notifications for the messaging system – currently a user needs to open a chat tab to see if he received any new messages. It would be a useful feature to inform the user if there are any unread messages.

- OAuth 2.0 authentication – it would allow to register a new user and log in by using already existing social media accounts. Currently users can register and log in by using their private e-mail addresses. With OAuth authentication a user could log in with his Facebook or Twitter account which would be a convenient option. This feature is present on many popular web applications.

- Linking social media profiles to the profiles in the application – currently a user has to post a link to his social media accounts in the "description" part of the profile. It would be better if there was an option to link these accounts to the profile in the application.

5.2. Difficulties in developing the application

During the development time we have encountered many difficulties. Quite soon we have realized that our initial specification was not detailed enough, and we needed to think about it much more in detail. Although, it was impossible for us to make a proper specification since we had no experience in developing web applications, thus we had no idea what we really needed. We had to learn everything step by step during the development phase.

To create the front-end, it was necessary to not only learn how to use the React library but also to learn JavaScript itself. The development process ended up being very messy, and the following are a couple issues that were encountered.

- **There are countless ways of doing the same thing.** However, the methods by which we do them must be persistent. Which meant that when researching how to do something, the most popular results were basically the same, all using the same packages and methods, because that is the easiest or simplest way. But then when researching how to accomplish another thing, the most popular solutions, the ones with the most documentation and examples, might use completely different methods or packages making them useless in our case. This resulted in a lot of digging for the right answers or sometimes solving the piece in creative ways, often sacrificing efficiency (because of lack of experience in JavaScript and React).

- **Updates.** React, and its vast library of packages, are very much "alive" meaning they are updated often. Although we worked on this project during a span of only 3 months, we encountered multiple updates. Some resulted in complete code changes because the recommended way of doing something (using some package) changed completely due to a new package rolling out and the old one becoming obsolete.

- **Structure** Again, because of lack of any experience in web development and time being limited, there was no time to learn and then do, it had to be learning by doing, and by doing many mistakes. React is not based on classes but on functions, it was hard trying to plan ahead how they will all come together. Thus the code is not very well structured. Some functions are hundreds of lines long and parts of code are repeated multiple times. The folders and files are also not organized properly. With more experience now, it is clearer how each component could have been broken down and then connected up. In turn allowing for more re-usability and scalability.

We have also encountered problems with the Spring Boot framework. Since it is a highly developed framework, it is hard to understand what is going on behind the code, when one has little experience. It has many annotations and hidden functionalities which had to be learned slowly one by one, before we could fully understand how the application built with this framework works. Initially we were also dealing with problems with Intellij Idea IDE, which did not work as it was supposed to.

On top of battling individually with our front-end and back-end struggles, we both had to expand our understanding on how full-stack websites are made and how to make everything work. Everything from HTTP requests and establishing a persistent, two-way communication using web sockets, to understanding how to establish user roles by authentication. Some things took a significant amount of time to simply prepare to start coding. For example, the chat service. We not only did not know how to code it, we had no idea what we needed. Hence, we first had to research what is necessary to create a chat service. After understanding we need a way to send and receive messages by first establishing a two-way connection and so on, we had to research possible solutions and decide and agree on which one we will use. The more choices there were, the longer this process took. And only after that, it came the time to research on how to implement it in our environments.

5.3. Final word

We have managed to create a fully working web application. It was a new experience for the team members, since we have never worked with web applications before nor with the technologies which have been used in this project. Despite the obstacles and many difficulties, we both agree that it was a positive experience for us. We have learned a lot; each of us could expand horizons in the field that interested us. Rozalia has learned how to use the React library and Rafał gained his first experience in Spring Boot framework. We both have learnt how to connect the front-end application with the web server using REST API and in general how the network protocols work.

We are sure that we will be able to use this experience in our future careers as software engineers. Moreover, we delivered a working tool for travelers, which, after putting some extra work in improving the details, could become a popular social application on the market of services for travelers.

6. Attachments

6.1. Division of work

Table 6.1: Division of work

Author	Work
Rafał Bogdanowicz	Back-end
Rozalia Korycka	Front-end
Rafał Bogdanowicz	Database
Rafał Bogdanowicz, Rozalia Korycka	Creating Docker images
Rafał Bogdanowicz, Rozalia Korycka	Deployment to Azure platform
Rozalia Korycka	Abstract
Rozalia Korycka	General Description
Rafał Bogdanowicz, Rozalia Korycka	Requirements Specification
Rafał Bogdanowicz	Existing Solutions
Rafał Bogdanowicz, Rozalia Korycka	Description of the Solution
Rafał Bogdanowicz	Analyis of the Solution
Rozalia Korycka	Examples of Use Case Scenarios
Rafał Bogdanowicz, Rozalia Korycka	Conclusions

Bibliography

[1] Solo Travel Statistics and Data: 2020, *https://solotravelerworld.com/about/solo-travel-statistics-data/*, 10th November, 2020.

[2] Spring Boot features, *https://dzone.com/articles/top-5-spring-boot-features-java-developers-should*

[3] Spring Boot dependency management, *https://www.javatpoint.com/spring-boot-dm*

[4] Securing Spring Boot application, *https://medium.com/wolox/securing-applications-with-jwt-spring-boot-da24d3d98f83*

[5] Spring Boot JPA, *https://www.javatpoint.com/spring-boot-starter-data-jpa*

[6] JSON Web Token, *https://jwt.io/introduction*

[7] Development testing, *https://www.softwaretestinghelp.com/the-difference-between-unit-integration-and-functional-testing/*

[8] Interaction Cost, *https://uxdesign.cc/the-simplest-but-most-abused-ui-components-part-1-3b30f32dd65c* Merve Orhan Nov 4, 2018

[9] Proximity of snacks to beverages increases food consumption in the workplace: A field study, *https://pubmed.ncbi.nlm.nih.gov/27112315/*

[10] WebSocket package for React, *https://www.npmjs.com/package/websocket*

[11] React, *https://reactjs.org/*

[12] JWT *https://jwt.io/*

[13] User Experience (UX) Design *https://www.interaction-design.org/literature/topics/ux-design*

[14] Using WebSocket to build an interactive web application *https://spring.io/guides/gs/messaging-stomp-websocket/*

List of symbols and abbreviations

API	Application Programming Interface
ID	Identification
SQL	Structured Query Language
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
DOM	Document Object Model
HTTP	Hypertext Transfer Protocol
JPA	Java Persistence API
JDK	Java Development Kit
MVC	Model-View-Controller
REST	Representational State Transfer
JWT	JSON Web Token
JSON	JavaScript Object Notation
GSON	Open-source Java library to serialize and deserialize Java objects to (and from) JSON
JSX	JavaScript XML
SPA	Single Page Application
RTL	React Testing Library
UI	User Interface
UX	User Experience
SPA	Single Page Application
SQL	Structured Query Language
TCP	Transmission Control Protocol
STOMP	Simple Text Oriented Messaging Protocol

List of Figures

List of Tables