UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Platform for finding optimal trip plan

Author:
Khrystyna KOKOLIUS

Supervisor:
Dmytro PRYIMAK

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences and Information Technologies
Faculty of Applied Sciences

APPLIED SCIENCES FACULTY

Lviv 2023

Declaration of Authorship

I, Khrystyna KOKOLIUS, declare that this thesis titled, “Platform for finding optimal trip plan” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

“Success is not final; failure is not fatal: It is the courage to continue that counts.”

Winston S. Churchill

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Platform for finding optimal trip plan

by Khrystyna KOKOLIUS

Abstract

This thesis focuses on developing an architectural solution for the trip planning platform for finding the optimal trip path with appropriate tickets using particular search filters. Existing solutions on the market were investigated to understand what new features can be developed and what can be improved. In addition, there were reviewed algorithms that can be used to solve optimization problems in searching for optimal paths. An important thing to mention is that the main aim was designing and developing an appropriate architectural solution. The usage of a particular algorithm for solving an optimization problem does not relate to the main goals of this thesis. There were investigated possible functional and non-functional requirements, designed microservices architecture together with data and math models and a simple user interface. Moreover, there was developed infrastructure on the Amazon Web Services cloud computing platform for hosting a website using the IaC approach. The platform is successfully hosted on AWS and tested on real users.

Link to the Github repository.

Link to the demo of the platform.

Link to the website.

Acknowledgements

I want to express my gratitude to my supervisor Dmytro Pryimak for his constant support, regular discussions, and help in decision-making. Many thanks for leading me in my thesis work and taking the time to help me.

In addition, I want to thank my family and friends for their faith and support for all four years of studies, accompanied by ups and downs. I am very grateful for the experience I got at the Faculty of Applied Sciences at Ukrainian Catholic University and very thankful for all gained knowledge, connections, and opportunities.

The last most important thanks to the Armed Forces of Ukraine for making it possible to write this thesis under a peaceful sky.

Contents

Declaration of Authorship i

Abstract iii

Acknowledgements iv

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Goal 1

2 Related Works 3
- 2.1 Trip planning applications 3
  - 2.1.1 Skyscanner 3
  - 2.1.2 Kiwi 4
  - 2.1.3 Omio 5
  - 2.1.4 Rome2rio 6
  - 2.1.5 Conclusions regarding existing applications 7
- 2.2 Algorithms for solving Travelling Salesman Problem 7
  - 2.2.1 Simulated Annealing 8
  - 2.2.2 Tabu Search 8
  - 2.2.3 Ant colony optimization 8

3 Data 10
- 3.1 Kiwi API 10
- 3.2 Flixbus API 10
- 3.3 Additional Data 11

4 Approach 12
- 4.1 Functional and non-functional requirements for the platform 12
- 4.2 Architecture overview 13
- 4.3 Microservices model 14
  - 4.3.1 Sign Up service 14
  - 4.3.2 Sign In service 15
  - 4.3.3 Search service 15
  - 4.3.4 Insert Update Tickets background process 16
  - 4.3.5 Main application service 17
  - 4.3.6 Motivation of built architecture and chosen framework 17
- 4.4 Data model 17
  - 4.4.1 Users database 17
  - 4.4.2 Tickets database 18
  - 4.4.3 Motivation of choosing Postgres 20
- 4.5 Math model 20
- 4.6 User Interface 22

4.7 AWS architecture of the platform 26
4.7.1 Motivation of choosing AWS and Terraform 26

5 Experiments and Results 28
5.1 Testing different search criteria on platform 28
5.2 Testing on real users 29

6 Conclusions 34
6.1 Result Summary 34
6.2 Future improvements and work 34

Bibliography 35

List of Figures

2.1 Multi-city search in Skyscanner . . . 3
2.2 Nomad Search in Kiwi . . . 5
2.3 Return Search in Omio . . . 6
2.4 Transportation Search in Rome2Rio . . . 6
2.5 Tickets Search in Rome2Rio . . . 7

4.1 Use case diagram of the platform . . . 13
4.2 Main components of the system . . . 14
4.3 Detailed architecture of the platform . . . 14
4.4 Data model of the platform . . . 18
4.5 Genetic algorithm steps . . . 21
4.6 Home page of the platform . . . 22
4.7 About section on the platform . . . 23
4.8 Sign up page on the platform . . . 23
4.9 Sign in page on the platform . . . 24
4.10 Search page of the platform with example, part 1 . . . 24
4.11 Search page of the platform with example, part 2 . . . 25
4.12 Sample result of the optimal path search . . . 25
4.13 Platform architecture on AWS . . . 27

5.1 Diagram of age of real users . . . 30
5.2 Evaluation of usefulness of this platform . . . 30
5.3 Evaluation of Sign Up part . . . 30
5.4 Evaluation of Sign In part . . . 31
5.5 Evaluation of Search part . . . 31
5.6 Evaluation of application’s speed . . . 31
5.7 Evaluation of UI . . . 32
5.8 Possible usage of application . . . 32

List of Listings

4.1 Sample input to Sign Up service 15
4.2 Sample input to Sign In service for sign in endpoint 15
4.3 Sample input to Sign In service for sign out endpoint 15
4.4 Sample input to Search service 16

List of Abbreviations

AWS	**A**mazon **W**eb **S**ervices
ECS	**E**lastic **C**ontainer **S**ervice
ECS	**E**lastic **C**ontainer **R**egistry
IaC	**I**nfrastructure **A**s **C**ode
TSP	**T**ravelling **S**alesman **P**roblem
UI	**U**ser **I**nterface

Chapter 1

Introduction

1.1 Motivation

In recent years the popularity of traveling is growing very fast, and people are exploring the world even more and more. But when it comes time for trip route planning, it can be a very complex task that can consume a lot of time. Many travel options are available, such as buses, trains, flights, etc., making decision-making even harder. Other essential factors are prices, travel dates, transportation times, and travel providers, which makes this task even harder.

As the business market grows very fast, many platforms for building optimal trip plans were created using different means of transport with diverse filtering options. They use various algorithms and advanced techniques to give as many travel choices as possible.

However, there are some challenges left that are not making these platforms perfect for planning. It still takes a lot of time to find tickets to visit more than two or three places in one trip because travelers should make different combinations to find a suitable one. There can be many variants, especially when the number of places to visit grows.

This thesis aims to develop a platform that will not just find travel options but will build optimal paths through different places using specific search criteria. This optimal path will not be based on locations' order but will be focused on the best option to travel, minimizing time on transportation and price. An important thing to mention is that the main focus is on designing and developing the architecture but not on investigating and testing algorithms for solving optimal path searches.

1.2 Goal

There will be created a platform for finding optimal trip paths using destinations that users will provide. It will find a way that matches the search criteria, and the order of visiting places will be the most suitable. Users will get information about possible flight and bus tickets to visit all the destinations as best as possible.

The main search filters will be:

- Cities that traveler wants to visit
- Amount of days that should be spent in every city
- Travel dates
- Maximum price of the trip

- The maximum amount of stops during the transportation from one city to another
- Adults amount
- Transport options
- Airlines to exclude

There will be a focus on creating such optimal paths where price and transportation time will be minimized. Every traveler wants to spend a minimum amount of time on transportation and the same on the price. Furthermore, one more objective covers saving time on the trip planning process and ticket comparison.

Moreover, the next goals are going to be reached:

- Review of related works and algorithms for solving optimization problems
- Implementation of all needed services for making this platform work with appropriate architecture, data, and math model
- Implementation of simple UI design for possible usage
- Conduction of experiments on finding optimal routes on different numbers of cities and using various filters
- Host site and test it on real users

Chapter 2

Related Works

2.1 Trip planning applications

This section provides information about existing platforms that help travelers to plan trips. Considering existing solutions on the market is vital because it can help create a competitive product with more specific business needs and stand out among existing ones.

There are a lot of different applications that help plan trip routes with their own features. Here will be considered the most popular ones, such as Skyscanner, Kiwi, Omio, and Rome2rio.

2.1.1 Skyscanner

Skyscanner (*Skyscanner website*) is a flight aggregator that allows searching for available flights with such filters as price, amount of stops on the way, time of departure, time of transportation, airline, and airport.

FIGURE 2.1: Multi-city search in Skyscanner

It can help to find convenient tickets to one or multiple points, but using general criteria it is possible to get all relevant flights to the place or places chosen. There are search types such as Roundtrip, One way, and Multi city and categories of found

flights as Best, Cheapest, and Fastest. But there is no officially published algorithm that uses Skyscanner for flight search.

Skyscanner searches multiple options and as output generates a lot of available flights. Apart from that, a search can be narrowed using various criteria by choosing airlines to travel with or only suitable departure times. In addition to that, there is a great possibility to choose among the best, the cheapest and the fastest flights.

However, the multi-city option allows searching flights to different places only by a specific order and dates that are given to the application. This can make the search for appropriate flights a complex task. Moreover, Skyscanner allows searching only for one transport type. In such a way, the cheapest trip to a particular place cannot be found because flights can be rather expensive. Another thing is that there is no potential to buy tickets through Skyscanner, and there is a need to redirect to airlines' websites directly.

2.1.2 Kiwi

Kiwi (*Kiwi website*) is a platform for travelers that allows searching for flights, buses, and trains using different options and search types. There are such search types as Return, One-way, Multi-city, and Nomad, and there are such categories of tickets as Best, Cheapest, and Fastest.

Return allows finding flights in two ways at the same time. There are considered such filters as dates of departure and return, amount of passengers, cabin type, baggage preferences, amount of stops, transport type, connections, carriers, excluded countries on the way, times of departure and arrival, the maximum value of trip duration time, maximum time for stopovers, price, days of departure and return. One-way offers only flights to a specific destination on a range of dates, and the filters are used the same as in Return search.

The multi-city feature is for finding flights between a few cities using a specific order of visits and dates. In this case, there are fewer filters, namely stops, baggage preferences, times of departure and arrival, transport types, and days of departure. In addition to that, there is an interactive map that shows the plan of the trip. The last one is Nomad, which allows selecting cities to visit, and Kiwi generates optimal path solving Travelling Salesman Problem. Moreover, there are used such filters as departure dates, allowed trip length in days, baggage preferences, amount of stops on the way. After investigating this feature, there was noticed that the best route is mainly equal to the cheapest one or the fastest one, but it looks like there is not presented such a solution that is optimal considering cost and time together. Furthermore, in Nomad only flight options are used, but Kiwi also provides information about buses and trains in general. Apart from that, no official details were found on algorithms used for Return, One-way, Multi-city, and Nomad search types.

On the one hand, Kiwi provides many options to search for optimal travel plans, as many airlines and travel agencies are used. There can be found not only flights but also possible buses and trains. Apart from that, the availability of different search types is no less critical. Another great thing is that tickets can be bought directly from the Kiwi website. On the other hand, Kiwi's solutions are mostly considered separately on transportation time and price, so it concentrates only on one thing at the same time giving the best, cheapest, and fastest solutions.

FIGURE 2.2: Nomad Search in Kiwi

2.1.3 Omio

Omio (*Omio website*) is a travel platform that provides such services as comparison and booking options. It has more than 1000 travel partners across trains, flights, ferries, and airports. There are two search alternatives as Round trip and One-way. Round trip offers four different means of transport trains, buses, flights, and ferries, and gives the best trips using different categories such as cheap and fast, cheapest price, fastest and departure times. In addition to that, there are such filters as the number of passengers, departure and return dates, amount of stops on the way, carries, duration of the trip, departure time, arrival time, price, arrival and departure stations or ports or stations or airports depending on the transport. For One-way search type filters are the same.

Using Omio, buying tickets directly from this website is possible, but it depends on the provider, and for some, it redirects to the provider page. Furthermore, Omio produces an excellent interface, making navigating and choosing appropriate filters easy. One more important thing is that it provides a great variety of transport.

However, Omio is more suitable when a traveler wants to travel only from one place to another one and return to the same place, as there are no search options for multi-city trips. Users can only make comparisons, but it can take time and be complex.

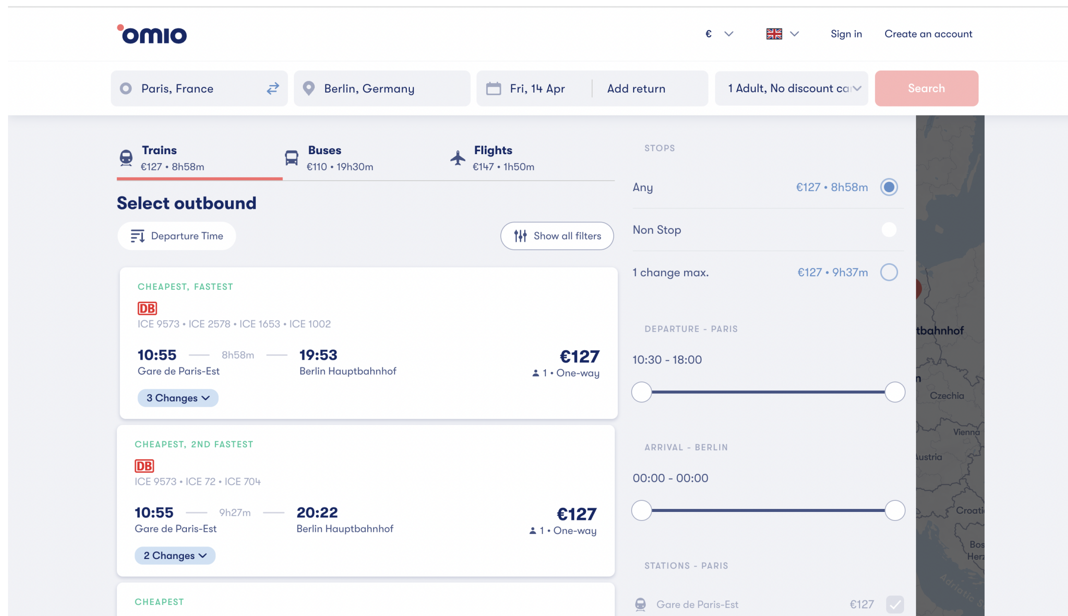

FIGURE 2.3: Return Search in Omio

2.1.4 Rome2rio

Rome2rio (*Rome2Rio website*) is a service for trip planning that provides two search options: Transport and Tickets. As for Transport, it offers the possibility to find all Transport offers to travel from one place to another one. Using this option, travelers can get information about approximate transportation prices using different means of transport. Apart from that, regular bus or train timetables are also provided with carriers' names and links to their website.

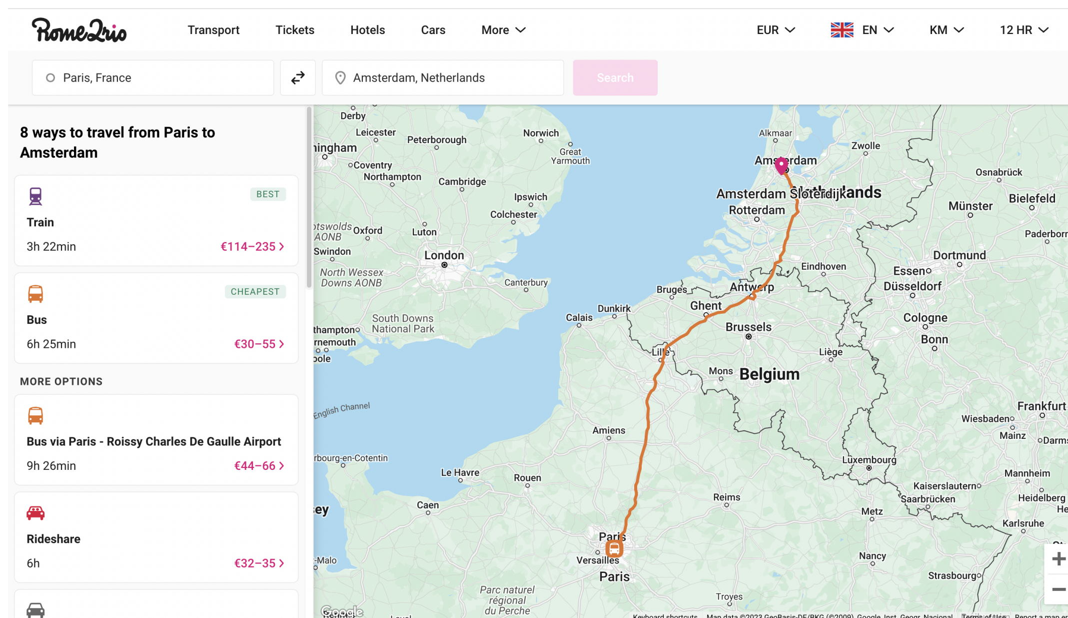

FIGURE 2.4: Transportation Search in Rome2Rio

Regarding Tickets search there are One-way and Return possibilities. There is a search for such types of transport as trains, buses, and flights. Apart from that, there are such filters as departure and return dates, amount of passengers, amount

of changes on the way, carries, and is the potential to sort flight tickets by departure time, cheapest, and quickest or bus tickets by departure times.

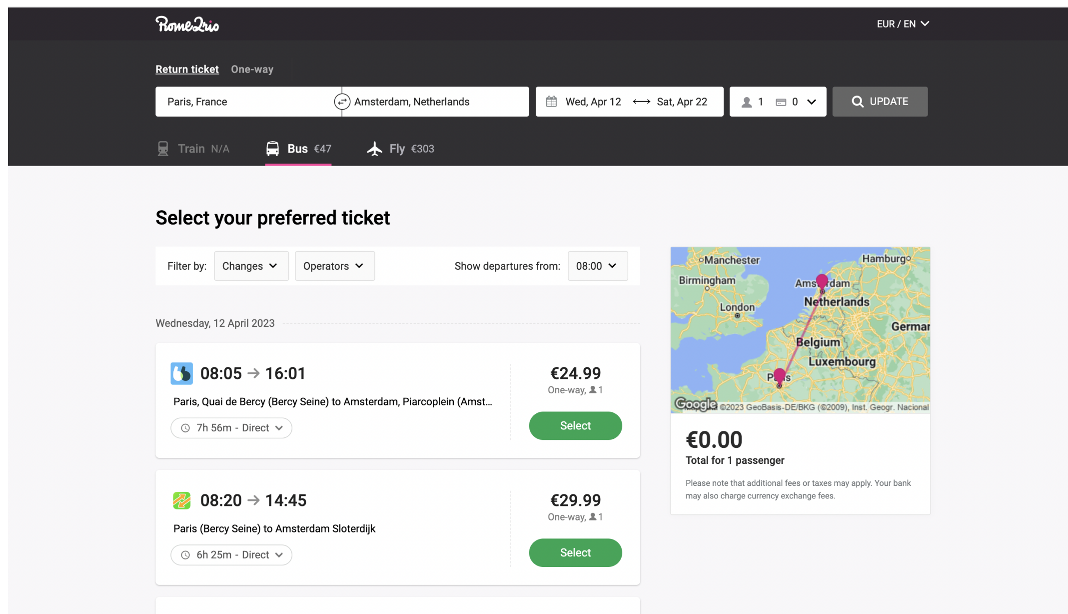

FIGURE 2.5: Tickets Search in Rome2Rio

Using Rome2rio, there is a possibility to get information using what kind of transport can be done transportation to a particular place. However, there are only a few filters when searching for travel tickets. In addition to that, only one-way and return travel options can be found that are more suitable for trips to one city. Furthermore, there is no possibility of booking a trip directly from Rome2rio.

2.1.5 Conclusions regarding existing applications

After an investigation of popular travel planning platforms, there was found that there can be provided great solutions when a traveler is searching for one-way or return ticket types considering different means of transport. All of the services have their benefits and drawbacks. Skyscanner is better for finding One-way or Return tickets when a traveler is searching for flights, Omio gives information on a bigger amount of transport types, Rome2rio is excellent for finding transportation options to different cities, and Kiwi is more effective for discovering travel offers when planning a trip to more than two or three cities. Unfortunately, these services do not provide official information on algorithms used for finding suitable transport tickets.

Furthermore, during the research, it was found that services are mainly concentrated on price and transportation times separately, among them only Omio has a filter for cheap and fast trips. It means there is what to improve, and in this thesis will be applied more filters to find tickets with minimum prices and transportation times.

2.2 Algorithms for solving Travelling Salesman Problem

In this section are reviewed metaheuristic algorithms for solving Traveling Salesman Problem. In this thesis definition of the Travelling Salesman Problem is used to

find optimal tickets between a few destinations. Such metaheuristic algorithms are considered as Simulated Annealing, Tabu Search, and Ant Colony Optimization.

2.2.1 Simulated Annealing

Simulated Annealing is a metaheuristic algorithm that starts from a single random state and is based on an annealing process in metallurgy. After that, a new solution is generated, and if this solution is better than the previous one, it is selected instead. If it is not better there is used a probabilistic function defined at the beginning of the algorithm, and if the calculations of this function are acceptable, then this new state is accepted. The generation of states is continued until equilibrium is met and temperature is continuously decreased. While evaluation criteria is not met previous steps are continuing. In general, in each iteration probability of choosing a worse solution is decreased, and the probability of finding the optimal solution increases. (Talbi, *Metaheuristics*)

An article prepared by Marek Antosiewicz, "Choice of best possible metaheuristic algorithm for the travelling salesman problem with limited computational time: quality, uncertainty and speed" investigated the computational possibilities of a few algorithms, including simulated annealing on Travelling Salesman Problem. According to the results of the experiments on different amounts of vertices in the graph, simulated annealing showed outstanding performance and solution quality among other algorithms.

2.2.2 Tabu Search

Tabu Search is a metaheuristic algorithm that generates an initial random solution and considers a tabu list of solutions that can not be applied. From the start, the first solution is assumed to be the best, and then a neighborhood with possible solutions close to this one is generated. The next best solution from the neighbors is selected despite the fact it can be even worse than the current ones, but it should not be in the tabu list. The previous best solution is then moved to the tabu list. On the next iterations, the best solution is replaced when there is found better one, and the previous one is added to the tabu list. The process of generating a new neighborhood for the last best solution continues while evaluation criteria is unmet. (Talbi, *Metaheuristics*)

Tabu Search was also investigated in this article (Marek Antosiewicz, "Choice of best possible metaheuristic algorithm for the travelling salesman problem with limited computational time: quality, uncertainty and speed") on solving the Travelling Salesman Problem. It showed a great performance on different amounts of nodes and together with simulated annealing is a leader among other tested algorithms.

2.2.3 Ant colony optimization

Ant colony optimization is a metaheuristic algorithm that considers ants' behavior while solving the problem. First, there are generated random solutions that contain the so-called ants. They are considering pheromone concentration on the way to find an optimal solution. After the first pheromone generation, newly generated ants are based on the previous solution, not to repeat mistakes and to choose the path that contains more pheromones and is shorter. There are created new ant solutions until evaluation criteria is not met. In general, this algorithm performs well on solving problems like Travelling Salesman Problem. (Talbi, *Metaheuristics*)

In addition to that, an article prepared by Míča1, "Comparison of metaheuristic methods by solving travelling salesman problem" investigated ant colony optimization on Travelling Salesman Problem together with simulated annealing and tabu search and showed the best results among them. But in general, all the algorithms performed well on a smaller amount of nodes.

Chapter 3

Data

This chapter describes the data sources used in this thesis and the retrieved data explanation. Kiwi and Flixbus APIs are used to get information about travel options to different places.

3.1 Kiwi API

Kiwi API provided by Tequila API (*Kiwi API provided by Tequila Reference*) offers information from the Kiwi website about available flights to different places. There is a limit in retrieving data from this API and it is 30 requests per minute.

Before starting the ticket search, there is made a request to **/locations/query** endpoint for getting the ids of cities used in the ticket search, but only when code for a particular city is not already saved in the table in the database. Using these ids is easier to find flights considering all airports located in specific city.

/search endpoint allows retrieving all needed ticket data from one city to another. There are such required fields in a query as **fly_from** - departure destination, **fly_to** - arrival destination, **date_from** - date from which search for flights, **date_to** - search flights up to this date. In addition to that, in this thesis will be used **select_airlines** - a list of airlines that will be excluded in the search, **select_airlines_exclude** - this parameter should be set to True to exclude in flight search certain airlines, **adults** - amount of passengers, **max_stopovers** - maximum value of the stops on the way.

As a result, a call to API provides such data that will be used for further investigation as **id** - unique id of the flight, used then to update information about the flight because price and availability are changing, **cityFrom** - departure destination, **cityTo** - arrival destination, **duration** - duration of the transportation, **fare** - the cost of the flight in euros, **availability** - the number of available places on the flight, **airlines** - airlines that are used in transportation, **local_departure** - local departure time in ISO format, **local_arrival** - local departure time in ISO format.

3.2 Flixbus API

Flixbus is a company that provides bus trips to different cities in Europe, North America, and Brazil. There are provided 3,000 travel destinations in 39 countries. It aims to give travelers the best service at a low cost.

Flixbus provides an API (*Flixbus API Reference*) with a few endpoints that give data about all available bus stations, schedules on incoming and outgoing buses, available tickets, and details of each trip. This thesis work uses endpoints such as /stations and /search-trips. There is a limit in retrieving data from this API and it is 10,000 requests per month.

/stations endpoint outputs information about all possible bus stations in different countries. It outputs such vital data as station name, station address, city name, and city id. In this thesis city id field is used for bus trip searching. The data from this endpoint is saved in a file in CSV format for further processing.

/search-trips endpoint provides information about all possible trips from one city to another one on a specific date. There are such required fields in a query as **to_id** - id of the city to where should be a trip, **from_id** - id of the city from where should be a trip, **currency** - currency in which is the cost of the ticket, **departure_date** - departure date of bus, **number_adult** - number of adults. The search is made by cities and is not dependent on certain stations. The data that is used after retrieving from API is **uid** - unique id of the trip, used then to update information about the journey because price and availability are changing, **departure timestamp** - departure time from the departure point that is represented as unix timestamp, **arrival timestamp** - arrival time to the arrival point that is defined as unix timestamp, **status** - means whether this bus trip has available places or not, **available** - gives information about the number of available seats, **price_average** - the price of the trip, **transfer_type** - provides information whether this bus trip is direct or has stops on the way, **interconnection_transfers** - is used to determine whether there are additional transfers in the journey.

3.3 Additional Data

Besides API data, this thesis work also uses airline and airport datasets from Kaggle. (*Airlines, Airport, and Flight Routes datasets*).

Airports dataset is used for background data load from Kiwi API using the airport name. From this dataset are used three fields, namely **name** - the name of the airport, **city** - the name of the city where an airport is located, and **iata** - unique identifier of the airport.

Airlines dataset is used for getting full airline names when retrieving data from Kiwi API. When calling Kiwi API in output are represented only airline codes that are not suitable as output for users. From **airlines** dataset are used such fields as **name** - the name of the airline and **iata** - unique identifier of the airline.

Chapter 4

Approach

This chapter will describe functional and non-functional requirements, the designed architecture of the travel platform, the data model, the math model used to solve the optimization problem for finding optimal trip plans, and the developed UI and IaC. In addition, the motivation for choosing system components will be explained.

4.1 Functional and non-functional requirements for the platform

Before considering the design of the architecture, it is necessary to determine the functional and non-functional requirements of the application. Functional requirements define the main use cases of the platform (figure 4.1), while non-functional describe operational capabilities.

In the case of a travel platform for optimal trip planning, there are such main functional requirements:

1. New users should be able to register on the platform.
2. Users can log in to the platform.
3. Users can search for optimal trip plans with such filters as departure city, cities that the traveler wants to visit, amount of days that should be spent in every city, travel dates, the maximum price of the total trip, the maximum amount of stops in transportation between cities, adult number, transport type and the possibility to exclude certain airlines.
4. Users can get a few variants of the best trip plans that consist of flight or bus ticket information with total price and transportation time with the order of cities to visit.

In addition to that, there are defined such non-functional requirements for the travel platform:

1. Highly consistent.
2. Great availability.
3. Great reliability.
4. User data should be durable.
5. The maximum latency of search of the optimal path and possible variants should be maximum of 60 sec.

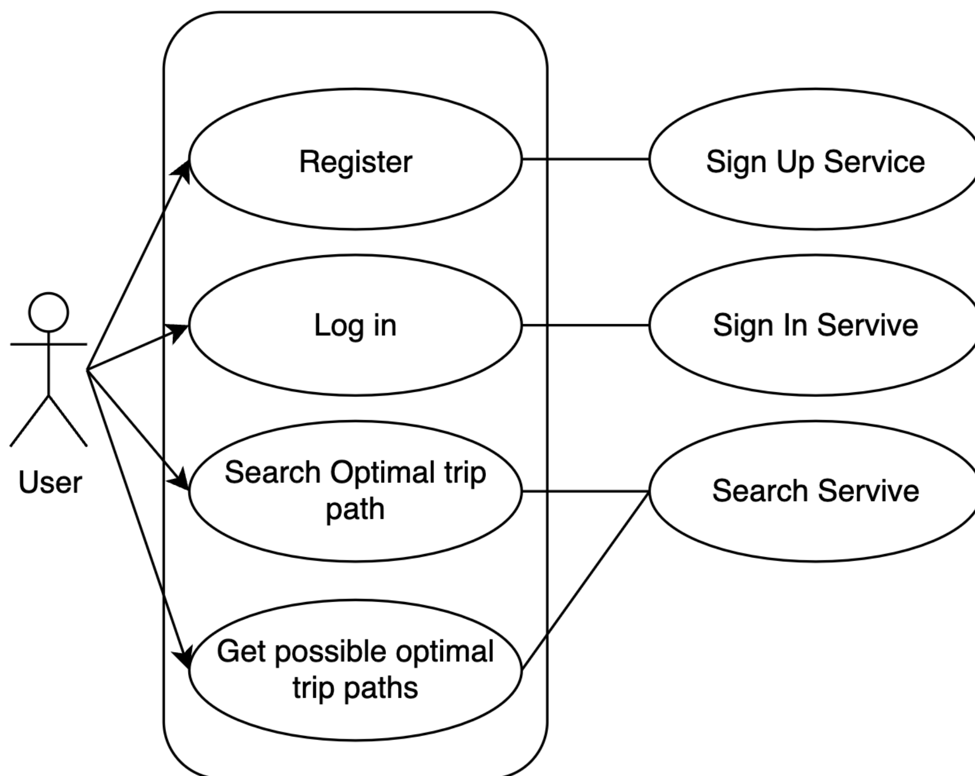

FIGURE 4.1: Use case diagram of the platform

4.2 Architecture overview

Using functional requirements, there can be defined main components of the system(figure 4.2):

- Microservices that are responsible for performing main operations on the platform. There can be defined such services as:
    1. **sign_up** service - for performing users registration.
    2. **sign_in** service - for performing users log in and log out to the travel platform.
    3. **search** service - searching for optimal trip plans by solving a mathematical problem.
- **insert_update_tickets** background process daily deletes outdated tickets from the database, inserts new ones, and updates existing ones.
- Main server that uses all microservices for building a platform.
- Storage for users and tickets data.

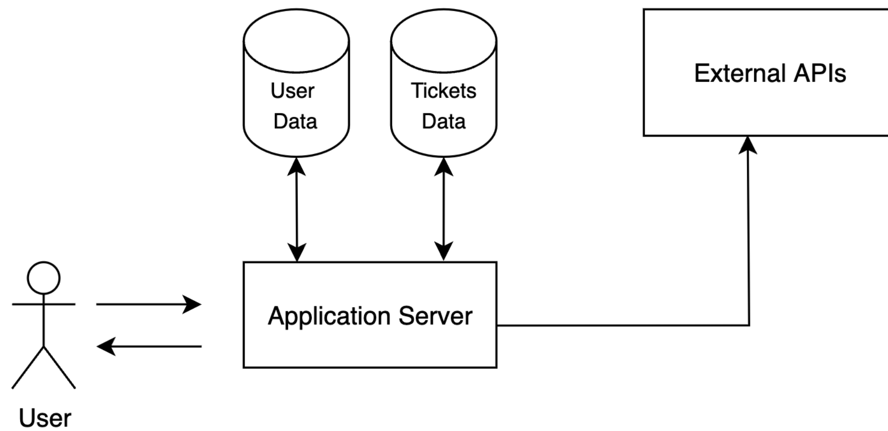

FIGURE 4.2: Main components of the system

4.3 Microservices model

In this section will be considered main services that are created for building the platform. Figure 4.3 shows the detailed architecture of the travel application with main databases and services.

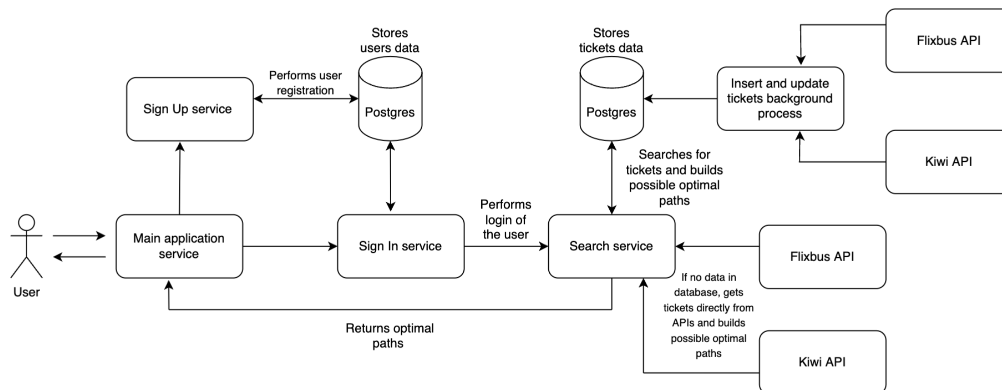

FIGURE 4.3: Detailed architecture of the platform

4.3.1 Sign Up service

Sign Up service is responsible for registering users and saving their data to the database. It is written using *Flask* framework in Python and has one main endpoint **/users_api/sign_up** to where can be sent only *POST* requests. Main service of the program sends *POST* requests to **Sign Up** service to register users in the system. The format of input data should be JSON. The main fields are username, email, and password. On listing 4.1 is showed sample content sent to **Sign Up** service.

LISTING 4.1: Sample input to Sign Up service

```
{
  "username": "test_user",
  "email": "test_user@gmail.com",
  "password": "Password12@"
}
```

After data is sent to the main endpoint of the service there is checked whether the user with such email has an account on the platform. If the user already exists, an account with such an email can not be created again. User should create a new account with a new email or sign in using the sign-in form. In addition to that, to register, a valid password should be entered that should contain upper and lower case letters, numbers, a minimum of one special character, and should consist minimum of 6 symbols. If all entered fields are appropriate, information about the user is saved in users table in users database, and **200** Response is sent back. One more thing, the password is encrypted, and only this version is saved in the table. For encryption is used *flask_bcrypt* extension in the *Flask* library that provides a hash function for passwords based on Blowfish cipher.

4.3.2 Sign In service

Sign In service logs users to the platform. It is written in *Flask* and has two main endpoints **/users_api/sign_in** and **/users_api/sign_out** to where can be sent only *POST* requests. Main service of the program sends *POST* requests to **Sign In** service to sign in or sign out users of the platform. The format of input data should be JSON. The main fields are email and password for sign in and email for sign out. On listing 4.2 is showed sample content for Sign in endpoint, and on listing 4.3 is showed sample content for Sign up the endpoint.

LISTING 4.2: Sample input to Sign In service for sign in endpoint

```
{
  "email": "test_user@gmail.com",
  "password": "Password12@"
}
```

LISTING 4.3: Sample input to Sign In service for sign out endpoint

```
{
  "email": "test_user@gmail.com",
}
```

When data is sent to Sign In service to **/users_api/sign_in** endpoint, there is checked whether a such user exists in general. If such a user does not exist, he can not log in to the system. In addition, there is made a check whether a user is already logged into the system. If the user is logged in, the users' table field **log_in** equals True, and when not False. If the user is registered and was not logged in before, **Sign In** service returns **200** Response and changes **log_in** to True.

4.3.3 Search service

Search service is designed to search for an optimal trip plan. The main responsibilities of this service are to search for appropriate tickets and solve optimization problems considering all constraints for finding optimal trip plans. It is written using

Flask framework in Python with such main endpoint **/search_api/search** to where can be sent only *POST* requests. The main service of the platform sends *POST* requests to **Search** service to find optimal paths between certain cities. The format of input data to this service should be JSON. The main fields are departure_destination, other_destinations, days_in_cities, date_from, date_to, price_max, adult_number, stopover_during_transportation, search_buses_flights and airlines_to_exclude.On listing 4.4 is shown sample input for search.

LISTING 4.4: Sample input to Search service

```
{
  "departure_destination": "Warsaw",
  "other_destinations": "Berlin,Paris,Barcelona",
  "days_in_cities": "4,5,4",
  "date_from": "01.07.2023",
  "date_to": "14.07.2023",
  "price_max": "500",
  "stopover_during_transportation": "1",
  "adult_number": "1",
  "search_buses_flights": "both",
  "airlines_to_exclude": "Ryanair"
}
```

Here is presented a more deep explanation of input values:

departure_destination - name of the city from where user will depart and to where will come back; *other_destinations* - names of the cities user wants to visit in the trip separated by coma; *days_in_cities* - amount of days user want to spend in each city separated by coma; *date_from* - date from which user want to start trip, *date_to* - date on which user want to come back to departure city; *price_max* - maximum price of total journey; *stopover_during_transportation* - amount of stops allowed during one transportation between cities, for example, 0 means that no transfer change during all transportations and 1 means that there are allowed maximum 1 transfer change on transportation between every two cities; *adult_number* - number of adults that want to perform such trip; *search_buses_flights* - identifies which transport to search, there are possible 3 options: both, flight and bus; *airlines_to_exclude* - airlines to exclude in tickets search.

When **Search** service gets input values, it first generates all possible transportation days depending on *date_from*, *date_to*, and *days_in_cities*. Then checks for tickets in the tickets database. If there is no ticket for the needed date and cities, the service starts to search tickets directly from *Kiwi* and *Flixbus* APIs, depending on the *search_buses_flights* it looks for flights, buses or both. An important thing to mention here is that now only the best ticket for one date for two cities combination is considered, and not all possible ones found to reduce the calculation time. This new data is saved then in the tickets database for future usage. When tickets for all possible dates for specific input parameters are gathered, their combinations are created, and optimization problems are formed. After problems solving, all results are gathered and returned.

4.3.4 Insert Update Tickets background process

Insert Update Tickets background process is responsible for deleting outdated ticket data from the tickets database, inserting new ones, and updating existing ones. This process runs once a day, loads and updates data for a few months. One important

thing to mention is that daily run is currently not working due to high cost usage. The purpose of this process is to make the work of the main platform service faster as there are for now some limitations of load from API and it makes search of tickets a rather long process, especially when there are many points and many generated possible dates. There are no inputs to this background process, and it first deletes tickets that are outdated and consume extra space in the table and starts the tickets search from tomorrow's date.

4.3.5 Main application service

Main application service is a main entry point to the trip planning platform. It is written in *Flask* and there such main endpoints **/**, /**sign_up**, /**sign_in**, /**sign_out** and /**search**. From **/** application starts and redirects to the home page from where the user can register and log in to the system. When the user is redirected to **/sign_up** and fills out the form, the main service sends *POST* request to **Sign Up** service, respectively when the user is redirected to **/sign_in** endpoint and fills out the form, there is send *POST* request to **Sign In** service. When the user is logged in, he is automatically redirected to **/search**, where he can start searching for optimal trip plans by submitting his request through the form. All submitted values are sent to **Search** service for further processing, and when it is done, optimal paths are sent back to the main service, which outputs these results to the user.

4.3.6 Motivation of built architecture and chosen framework

In building the platform architecture was chosen microservices approach. One main service communicates with three others through *HTTP POST* requests. One of the main reasons for choosing such an approach is that all microservices are independent and do not affect the work of others. In addition to that, it is easier to scale services separately, and fault tolerance together with data security are improved. If one of the services fails, other application parts will continue working.

For building the main service and other microservices was chosen *Python Flask* framework for a few reasons. First of all, it is easy to use and flexible. It is suitable for making the first version of the system and using Flask it is not hard to debug failures using its built-in debugger. In addition to that, this framework has a great scaling possibility and can process many requests at the same time.

4.4 Data model

Two databases are created to save users' and tickets' data in the travel platform (figure 4.4). Each of them has its tables with specific fields. In both cases, is used Postgres database management system to save data.

4.4.1 Users database

Users database consists of users table that contains information about registered users on the platform. Description of users table fields:

- **id** - unique id of the user.
- **username** - name of the user.

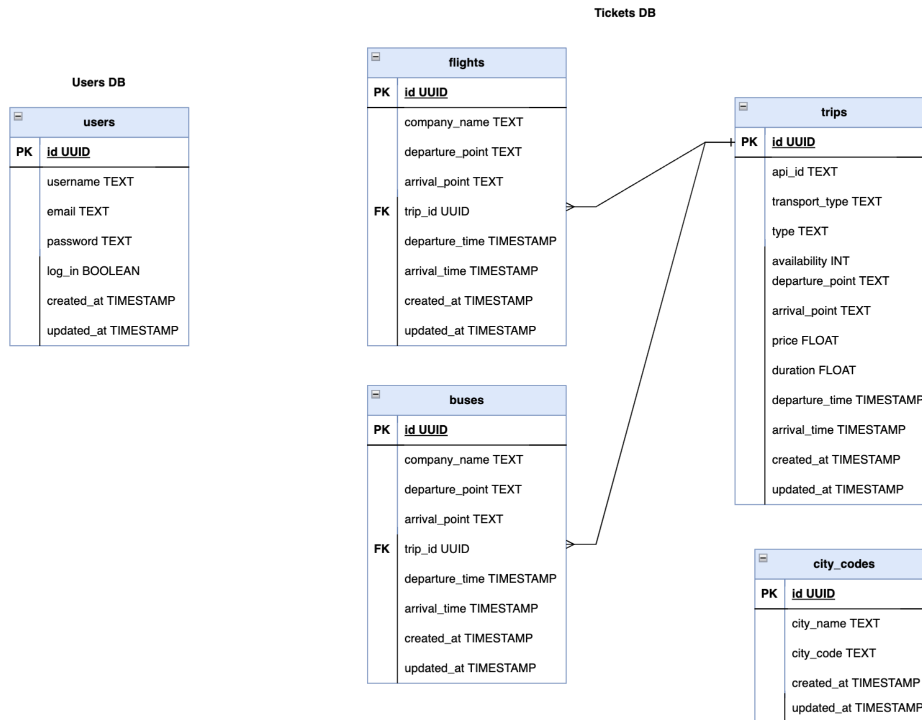

FIGURE 4.4: Data model of the platform

- **email** - email of the user which must be unique.
- **password** - encrypted password of the user.
- **log_in** - boolean value which indicates whether user is logged in to the platform.
- **created_at** - saves timestamp that identifies user registration time.
- **updated_at** - saves timestamp which indicates when record was updated last time.

4.4.2 Tickets database

The tickets database consists of four tables: trips, flights, buses, and city_codes. Data that is saved in these tables is gotten using Kiwi and Flixbus APIs. The trips table holds information about possible tickets from one place to another. Description of trips table fields:

- **id** - unique id of the trip.
- **api_id** - unique id of the returned possible trip from the API. Using this id then can be updated information about the ticket from one place to another.
- **transport_type** - type of the transport, possible values are flight and bus.
- **type** - this field describes whether ticket from one place to another is direct or with stops.
- **availability** - saves how many tickets are still available on such flight or bus trip.

- **departure_point** - place from where is ticket.
- **arrival_point** - place to where is ticket.
- **price** - price of the ticket from one place to another.
- **duration** - total transportation time from departure to arrival point.
- **departure_time** - time of departure from departure point.
- **arrival_time** - time of arrival to arrival point.
- **created_at** - saves timestamp that identifies addition of the trip record to the table.
- **updated_at** - saves timestamp which indicates when record was updated last time.

Flights and buses tables are meant to save data about transfers from departure point to arrival. For example, a flight change in Berlin can be performed to get from Warsaw to Paris. That means there is one stop on the way, and the flight to Paris consists of two flights. That is why data about all transportation from departure to arrival point is in separate tables. In addition to that, it is more convenient to keep flights and bus data separately from each other, but these tables have the same schema:

- **id** - unique id of the transportation possibility.
- **company_name** - name of the carrier.
- **departure_point** - place from where will be made transportation.
- **arrival_point** - place to where will be made transportation.
- **trip_id** - id of the trip from trips table.
- **departure_time** - time of departure from departure point.
- **arrival_time** - time of arrival to arrival point.
- **created_at** - saves timestamp that identifies addition of a certain transportation information.
- **updated_at** - saves timestamp which indicates when record was updated last time.

A background process will clean all outdated data; only appropriate possible tickets will remain.

The last table that is located in the tickets database is city codes. It contains information about city names and their codes retrieved from Kiwi API. The reason for creating such a table is that API calls to Kiwi API are limited, and it is suitable to save already retrieved data in the table and not to make the same API call. City codes are used to get flight data from one city to another from Kiwi API.

4.4.3 Motivation of choosing Postgres

Postgres is a relational database management system that ensures data resilience and is used to save structured data. In the case of the traveling platform, all gotten data is structured and can be saved in defined tables. One of the motivations for using a relational database is that it is easy to perform any changes such as inserts and updates of records. In addition to that, a relational database is a great choice when tables have relations between each other, and there should be performed joins of data. In the tickets' database trips table is connected to flights and buses tables through id to get more detailed information about transportation between cities. Moreover, a relational database supports ACID and ensures that data will be consistent and durable. This is important for saving users' data because it should not be lost.

4.5 Math model

To find optimal paths in the platform, there is used Travelling Salesman Problem represented as a linear integer optimization problem, and to solve it Genetic Algorithm is used. Here should be solved a typical TSP problem where the user should visit every city only once with the minimum total cost and get back to the initial point. But the formulation of the problem itself differs from the classic ones and has more constraints. Here is an emphasis on minimizing the total price of the tickets together with transportation time that defines a multi-objective problem but using weights, it was converted to a single-objective problem. In addition to that, in the platform are solved numerous such problems depending on the amount of found tickets. After finding all possible tickets for all possible dates, combinations are made from them, and created separate TSP problems that are then solved using Genetic Algorithm minimizing objective function taking into account constraints.

Objective function of the problem is defined as:

$$min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} wp_{ij}x_{ij} + wt_{ij}x_{ij}$$

where w is a weight of price and transportation time in equation and $w = 0.5$, that means that weight of price and transportation time are the same in the problem; p_{ij} - price to get from city i to city j in euros; t_{ij} - transportation time to get from city i to city j in hours;

There are such main constraints that are considered in Travelling Salesman Problem that should be observed:

$$x_{ij} \in \{0, 1\} \qquad i, j = 1, ..., n \quad (1)$$

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1 \qquad j = 1, ..., n \quad (2)$$

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1 \qquad i = 1, ..., n \quad (3)$$

(1) constraint shows that x can be equal to 0 or to 1, where 1 - path from city i to city j and 0 - shows that no path between city i to city j; (2) and (3) constraints ensure that traveler comes to a certain city only ones and departs from it also once

In addition to that, there are defined such additional constraints as the trip should always start from the departure city and end in this city, in each city must be spent a certain amount of days as the user of the platform wants, all the dates of tickets in the planned path should be the unique and total price of the trip has to be less or equal than maximum possible price.

For now, for problem solving is used Genetic Algorithm. This algorithm is a metaheuristic algorithm that uses natural selection as a foundation. In the iterations are modified populations using crossover and mutation processes until the perfect one is not found that will be greatly evaluated. Crossover makes from two parents one valid child with a selected modified part, and using mutation it is further modified by recombination of the selected part. (Talbi, *Metaheuristics*) All steps in the algorithm can be defined as follows:

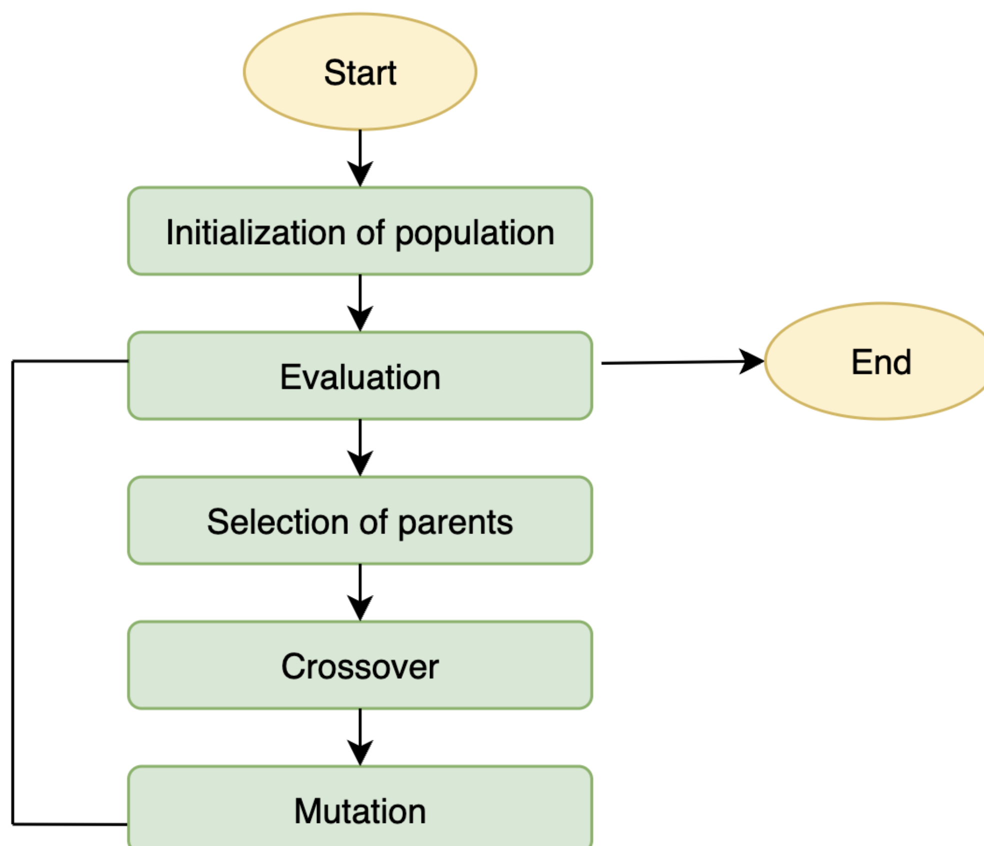

FIGURE 4.5: Genetic algorithm steps

In chapter 2.2 were described three other algorithms suitable for such kind of problem, namely Simulated Annealing, Tabu Search, and Ant Colony Optimization. In the article written by Míča1, "Comparison of metaheuristic methods by solving travelling salesman problem" Genetic algorithm was also tested and with fewer nodes it performed well but with more ones solution was inaccurate. But the motivation for choosing this algorithm to solve the problem of finding an optimal

path in this thesis is that it is suitable to solve such a problem and is easy to understand. In the platform will be performed search not for a big amount of cities at one time, that is why this algorithm will be quite enough to solve the problem. This is somehow a random algorithm so that a solution can be found faster. In the platform this algorithm is represented using *pymoo* library in Python, an optimization library that represents different algorithms for solving optimization problems.

4.6 User Interface

A simple interface of the platform was created to make it usable for users. The design was created using *HTML* and *CSS*.

On the home page, the user can go directly to search after signing up and signing in to the platform. To go further, there should be clicked *search* button. (figure 4.6)

To get more information about the platform user can click the *About* button on the header. There are described the purpose of this platform and the main benefits.(figure 4.7)

Sign up (figure 4.8) and Sign in (figure 4.9) are represented as forms. Users should register first, then sign in to be able to search for optimal paths.

After signing in user can search for optimal trip paths. Search is represented as a form where the user should enter requirements regarding the trip. (figure 4.10, 4.11) All the inputs are redirected to the main application service and, after that, are sent to the Search service for processing. There should be sent all the inputs required by the Search service. On these two figures, 4.10 and 4.11 are also shown sample inputs in the form.

Figure 4.12 represents the sample output result of optimal path search. It consists of one main table and n more detailed tables about found paths. The first main table represents all found paths with their total price in euros and total transportation time in hours. Other tables contain information about every optimal path separately with details about every transportation option, cost, transportation time, carrier, local departure time, and local arrival time.

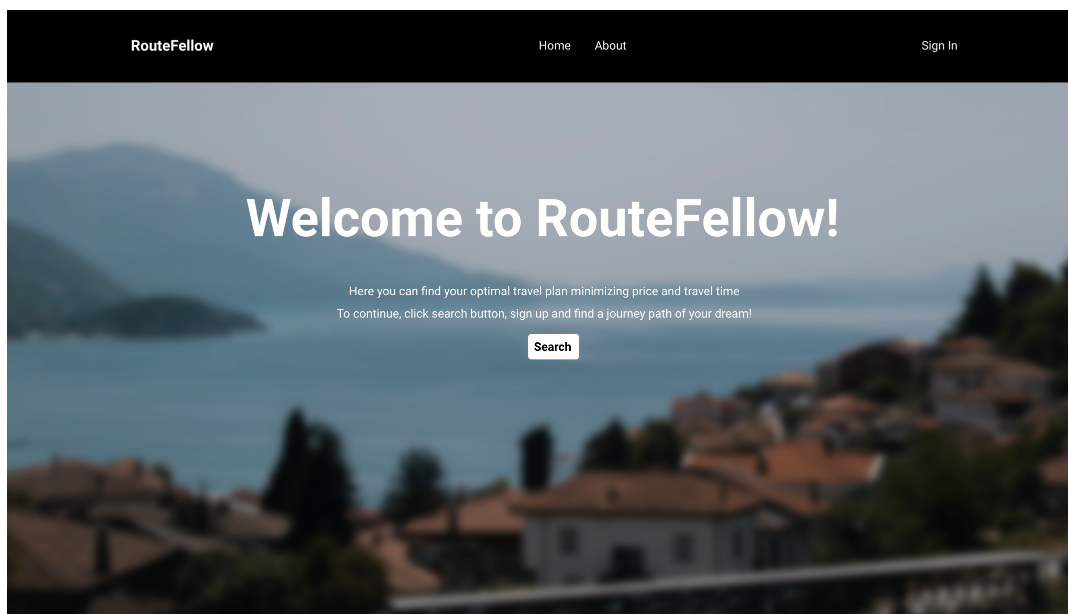

FIGURE 4.6: Home page of the platform

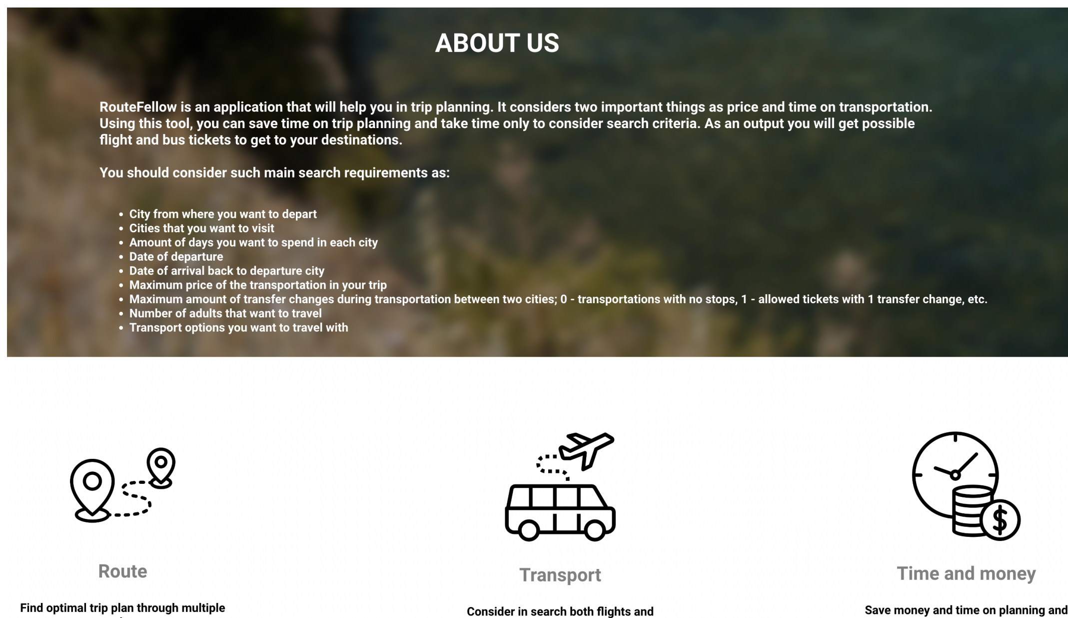

FIGURE 4.7: About section on the platform

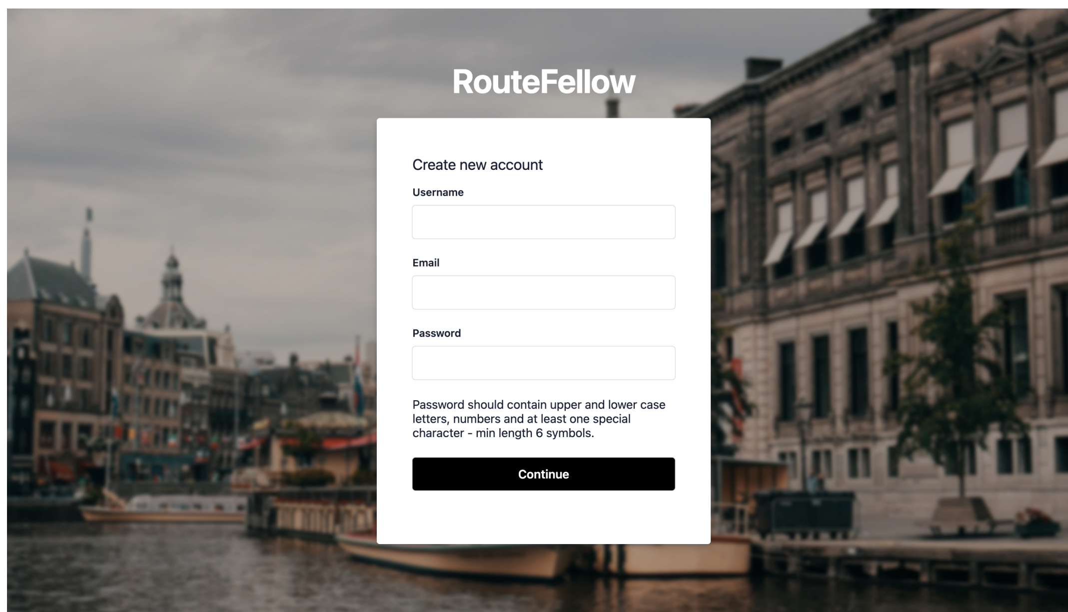

FIGURE 4.8: Sign up page on the platform

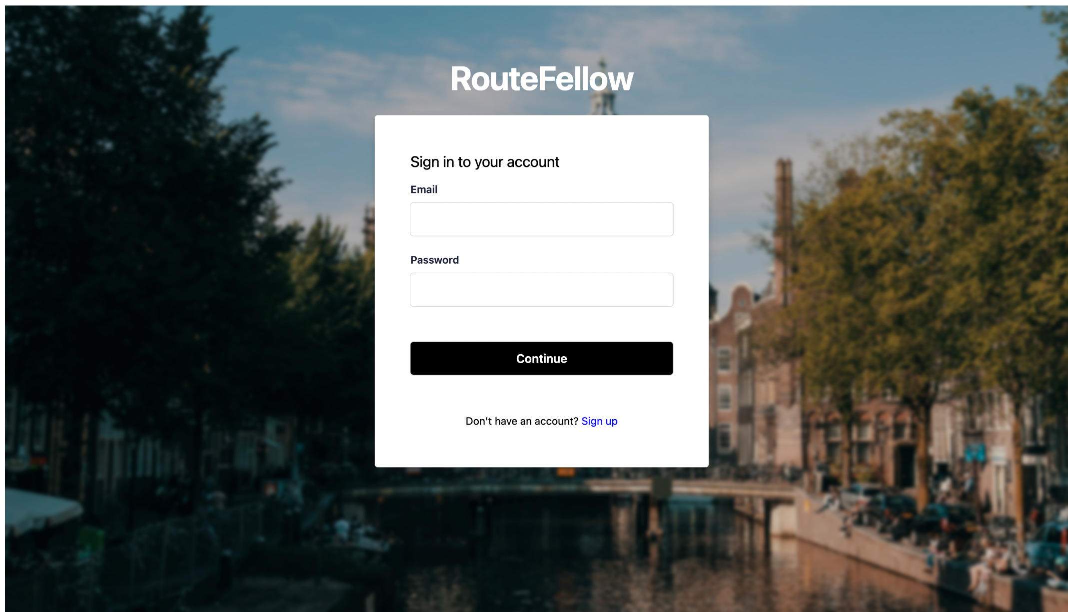

FIGURE 4.9: Sign in page on the platform

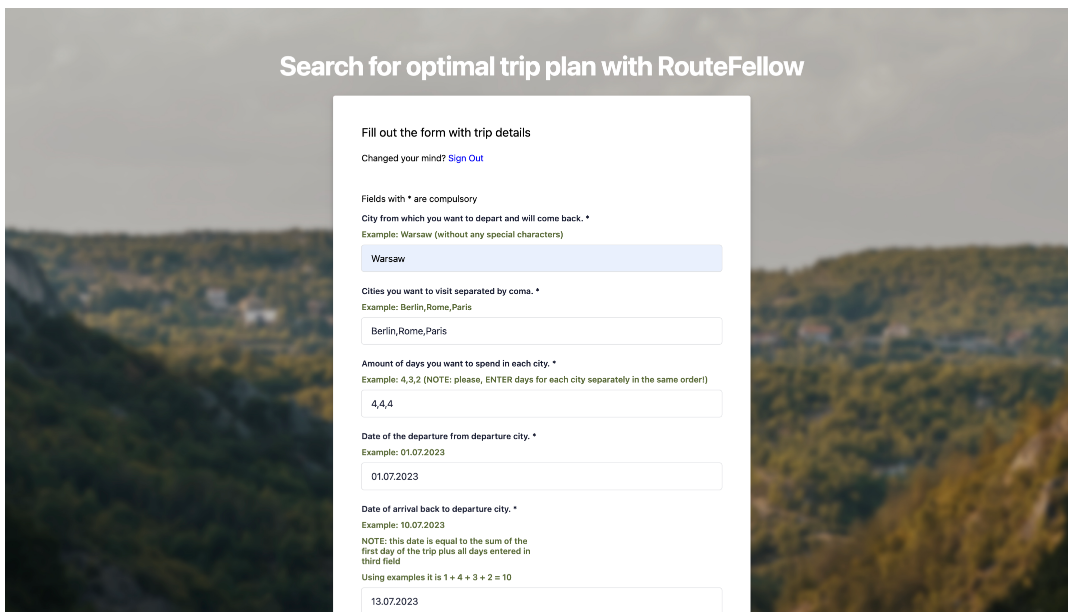

FIGURE 4.10: Search page of the platform with example, part 1

Maximum price of the total trip. *

Example: 300 (NOTE: in Euro)

300

Maximum transfer change amount during transportation from one city to another. *

Example: 1

This field means what is the maximum allowable value of transfer change between two cities

0 - all transportations with no stops on the way

1 - allowed tickets with max 1 transfer change in transportation between two cities.

2 - allowed tickets with max 2 transfer changes in transportation between two cities, etc.

1

Number of adults that want to travel. *

Example: 2

2

Transport options you want travel with. *

Possible options: bus, flight or both.

both

Airlines to exclude in search separated by coma.

Example: Ryanair,Air France.

Find optimal paths

FIGURE 4.11: Search page of the platform with example, part 2

Optimal paths results

Do you want to search for more paths? Search

Do not want to search for any other paths? Sign Out

Num	Cities order	Total price (EUR)	Total transportation time (Hours)
1	Warsaw-Rome-Paris-Berlin-Warsaw	170.48	31.08
2	Warsaw-Berlin-Rome-Paris-Warsaw	208.49	14.33
3	Warsaw-Berlin-Paris-Rome-Warsaw	209.96	77.16

Warsaw-Rome-Paris-Berlin-Warsaw

Path	Price (EUR)	Transportation time (Hours)	Carrier	Local departure time	Local arrival time
Warsaw-Rome	63.5	2.25	Ryanair	2023-07-01 14:40:00	2023-07-01 16:55:00
Rome-Paris	29.0	2.17	Wizz Air	2023-07-05 06:50:00	2023-07-05 09:00:00
Paris-Berlin	49.99	18.83	Flixbus	2023-07-09 01:10:00	2023-07-09 20:00:00
Berlin-Warsaw	27.99	7.83	Flixbus	2023-07-13 14:00:00	2023-07-13 21:50:00

Warsaw-Berlin-Rome-Paris-Warsaw

FIGURE 4.12: Sample result of the optimal path search

4.7 AWS architecture of the platform

Infrastructure as a Code approach to deploy application on Amazon Web Services was used to make the platform available for users. **Amazon Web Services** is a cloud computing platform (*AWS documentation*) that can be used for hosting websites and has many tools to choose from. All the architecture launched on AWS was written using Terraform (*Terraform documentation*) - Infrastructure as a Code tool.

For hosting microservices of the platform was chosen **Elastic Container Service** - container orchestration tool using which can be created services that run tasks isolated from each other. In task definitions are defined Docker images that should start running as containers on task launch. One of the advantages of this service is that it provides a serverless mode of task running, and there is no need to manage used servers. In addition to that, when tasks are deployed as services, they can be automatically relaunched in case of failure inside the Docker container.

To run platform services on AWS ECS, there were created five separate repositories in AWS **Elastic Container Registry** for **main**, **sign up**, **sign in**, **search** services and **background process** for inserting and updating tickets data to save their Docker images. The Elastic Container Registry service is used to store and manage Docker images. All repositories for the platform were created using Terraform, and only the push of images was performed manually.

All microservices are deployed inside one **Virtual Private Cloud** - an isolated section on AWS that can be interpreted as an isolated virtual network. This is used to make an application more secure and save it isolated. In addition, every microservice has its own **security group** - a firewall that controls all incoming and outcoming traffic by defined rules. Separate security groups were created to provide excellent security for every microservice. In addition, all platform services have their own **load balancers** to provide resource distribution between microservices instances and greater availability and performance. Every load balancer has its security group with provided inbound and outbound traffic rules.

For deploying Postgres servers to save application data was chosen **Relational Database Service** with Postgres engine. The reason for using Relational Database Service is that it provides a more straightforward setup and operation of databases used in the platform.

Figure 4.13 shows AWS architecture in detail.

4.7.1 Motivation of choosing AWS and Terraform

The main motivation for choosing AWS is that this cloud computing provider is great for website hosting as it can provide high availability and scalability. In addition to that, many different services can be used to build an infrastructure and provide excellent performance and security.

The motivation for choosing Terraform as infrastructure as a code tool is that it provides the possibility to provision AWS resources and is easy to use. Using one command all the architecture can be deployed in one click. Moreover, deleting and updating already created resources is easy as the entire state is remembered in the Terraform state file.

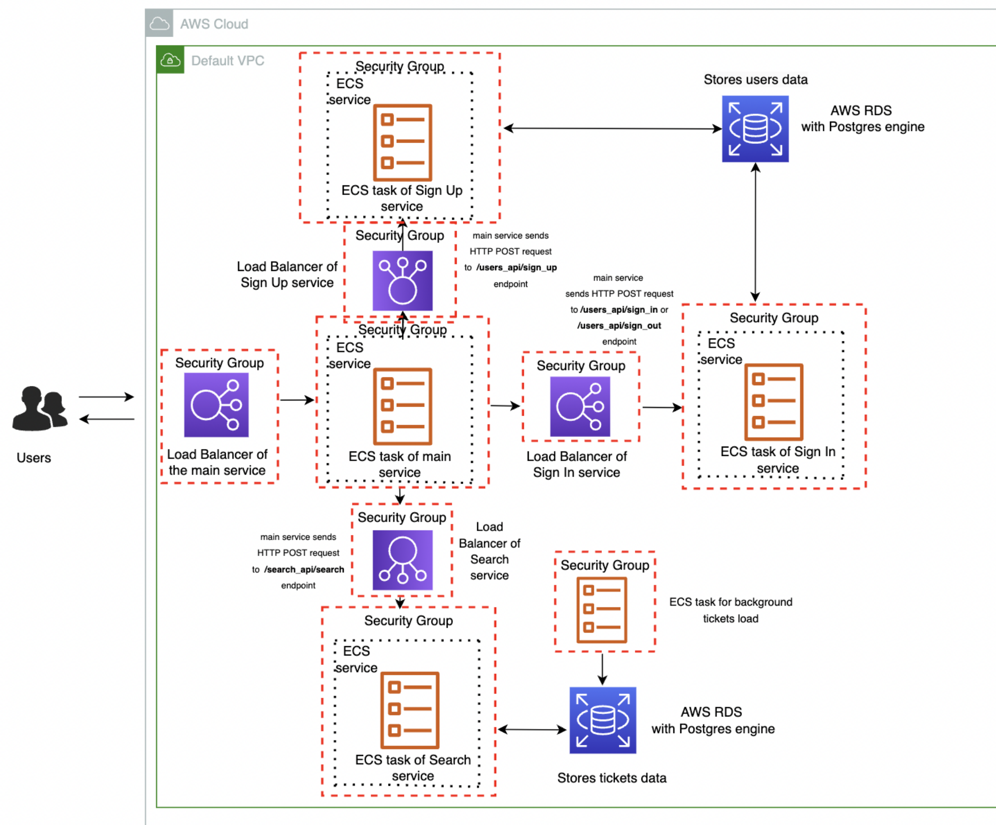

FIGURE 4.13: Platform architecture on AWS

Chapter 5

Experiments and Results

This chapter describes experiments performed on the developed platform using different search criteria and testing on real users.

5.1 Testing different search criteria on platform

Several experiments were conducted to check how well ticket filtering and optimal path search work on the platform.

1. Search for the trip plan from Lviv to specific destinations.

   After performing such an experiment, it was found that the search from Lviv is poorly performing as for now there are available only buses from this point, and not to all locations are buses from Lviv. But in general it is possible to find paths from Lviv to other cities.

2. Check the performance of the search part for different amounts of destinations from 2 to 10 cities.

   After checking the different amount of destinations to go to, there was investigated that when the amount of cities grows, the computation time also increases. Starting from 4 cities the user wants to visit as particular destinations, the computation speed starts to be very bad, and the user should wait a long time to get the results. This happens because the search service works on loading a significant amount of data from API and works on solving all the possible combinations of optimization problems. Starting from 5 or more destinations, it is almost impossible to wait all the time for the results.

3. Check the performance of the search part for the different number of days to stay in other cities.

   There was found that when there are many cities to visit for 1 or 2 days, performance worsens as there are many date combinations on which these cities can be visited. Because of that, there is performed a long search for tickets, together with solutions of optimization problems. But the search performs rather well when the number of days to spend in different cities correlates from 4, and the number of destinations is not so big.

4. Check the search for close dates and those in the distant future.

   Search for close dates performs poorly as no places on buses and flights can be available, but it differs significantly from dates and destinations. As for the search for a distant feature, it performs rather well, and tickets are mostly found, but it all depends on search parameters.

5. Check for different price limits.

   There were checked different price limits. When using a price limit of 1000, there is a more significant possibility of finding more diverse optimal paths. But in general, it is often possible to find optimal paths with a total price less and equal to 300. Everything depends solely on the entered destinations.

6. Check for the possibility of traveling only using direct transportation and with a transfer change.

   After performing experiments, it was found that there are possibilities to travel with transfer changes as well as using direct transportation. The amount of such found paths depends on entered destinations, dates, and price limits. In most cases, the direct transportation is more expensive than with transfer changes.

7. Check different transportation options: flight, bus, and both.

   These three transportation options were checked, and for all of them search part performed relatively well. But everything also depends on the entered parameters, and paths can occasionally not be found.

8. Check of work of certain airlines exclude.

   Excluding certain airlines works rather well, and airlines with which users do not want to travel are excluded. The ones that were entered were not present in the results. But when the airline is incorrectly entered, it does not work correctly.

After performing experiments, conclusions about the general work of the optimal paths search were made. First of all, when finding the optimal trip plan for a larger number of destinations, the work speed gets worse a lot because there should be found a lot of tickets and many combinations of optimization problems that should be solved. In addition, when choosing many destinations together with a small number of days to spend there as 1 or 2, there will be even more possible combinations because there are many potential dates to travel. However, when some tickets are saved in the database, the work of the search part slightly improves, and also when there are a small number of destinations in the search. But in the general search part works rather well and finds optimal paths, although it works for quite a long time. In addition to that, all the results depend on the entered parameters, and the user should be attentive when entering them.

5.2 Testing on real users

Platform testing was performed on real users, and all the feedback was gathered anonymously through Google Forms. There was gathered feedback from 28 users.

Testing was performed on the age category from 19 to 30, and a few answers were gathered from the 30+ age category. An important thing to mention is that the target audience was chosen primarily people interested in traveling and who have experience using popular applications for trip planning. The purpose of choosing such a people category is to get real feedback on the usefulness of this platform and things that can be improved. The age distribution is shown in more detail in the figure 5.1.

The Google form for feedback consisted of a few questions, and a detailed analysis of the answers will be made here.

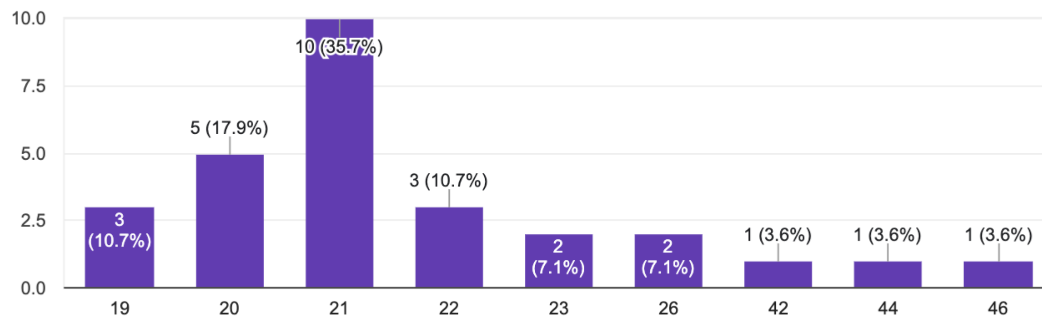

FIGURE 5.1: Diagram of age of real users

The first question was whether this trip planning application is helpful in trip planning, and suggested answers consisted of evaluation from 1 to 5: 1 - it is not useful at all, 5 - it is beneficial. This diagram 5.2 shows that this application can be useful in trip planning for most users that tested the platform.

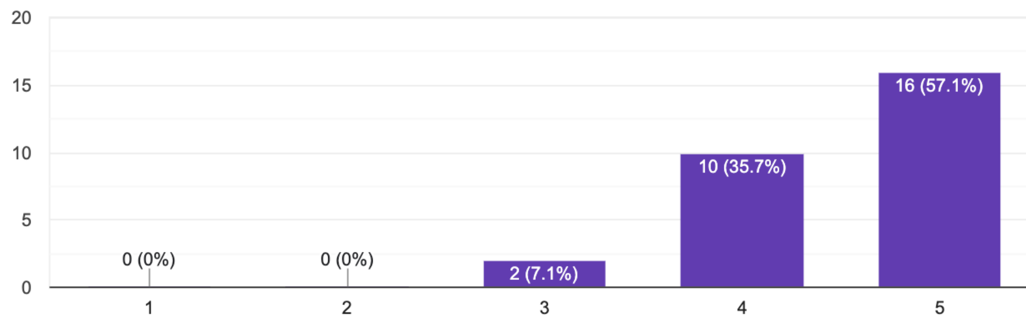

FIGURE 5.2: Evaluation of usefulness of this platform

Another three questions aimed to get feedback on separate parts of the application, namely Sign Up, Sign In, and Search from a user experience perspective. Answer choices consisted of evaluation from 1 to 5: 1 - particular application part works poorly, 5 - rather excellent. The figure 5.3 shows the results of the review of the Sign Up service that most users rated as 4. In general, from all the answers can be concluded that it works rather well, but for some real users, it did not work satisfactorily.

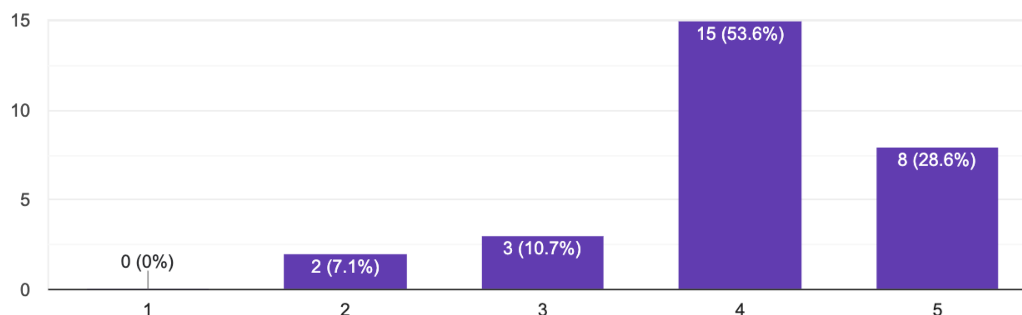

FIGURE 5.3: Evaluation of Sign Up part

In the figure 5.4 are described results of the evaluation of Sign In service that as Sign Up is mainly rated as 4. It can be concluded that it worked well for most users that tested the platform.

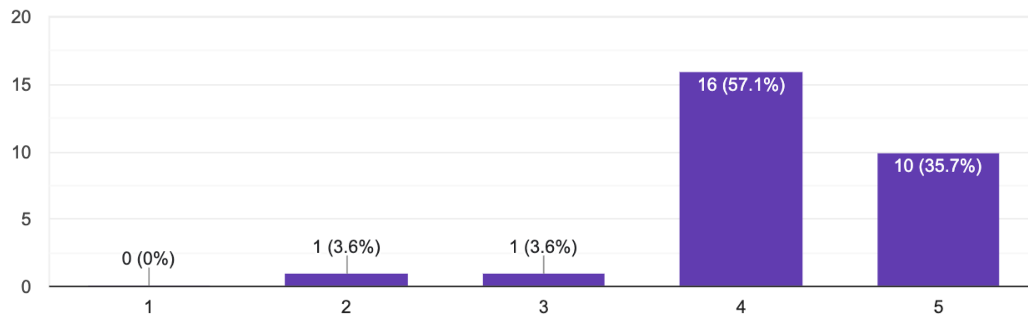

FIGURE 5.4: Evaluation of Sign In part

Evaluation of the Search part in the figure 5.5 shows that users' opinions somewhat differ, but most are rated as 3, 4, and 5. For four users that rated it as 1 and 2 search part worked rather badly. This means that the work of this part of the application should be reviewed and improved.

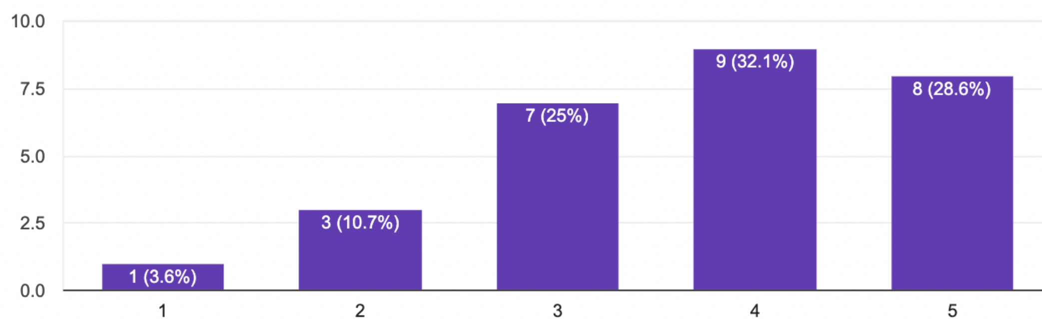

FIGURE 5.5: Evaluation of Search part

The next question was to rate the platform's speed from 1 to 5: 1 - speed is awful, and 5 - speed is rather great. The figure 5.6 shows results that can be interpreted as the speed of the application work is not so bad, but it is not also great. For some users speed is critical, and at the same time, for others is normal to wait for a generated path for some time.

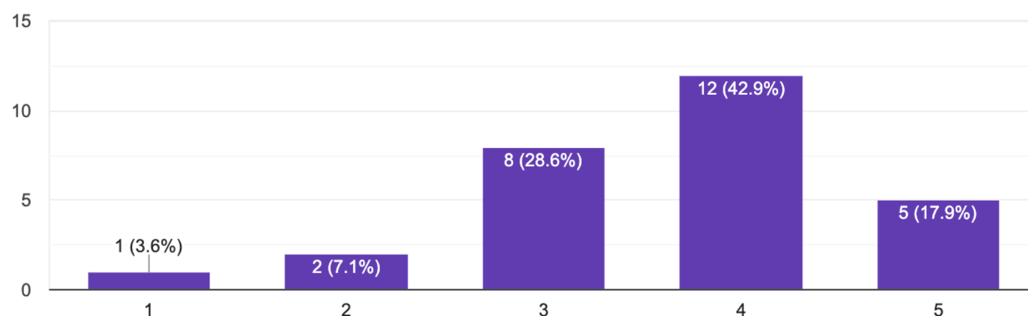

FIGURE 5.6: Evaluation of application's speed

In addition to all those questions, there was a question to evaluate the user interface from 1 to 5: 1 - the user interface is terrible, 5 - looks rather excellent. In the figure 5.7 are shown results that indicate that most users like the simple user interface developed. Most rated it as 4 and 5.

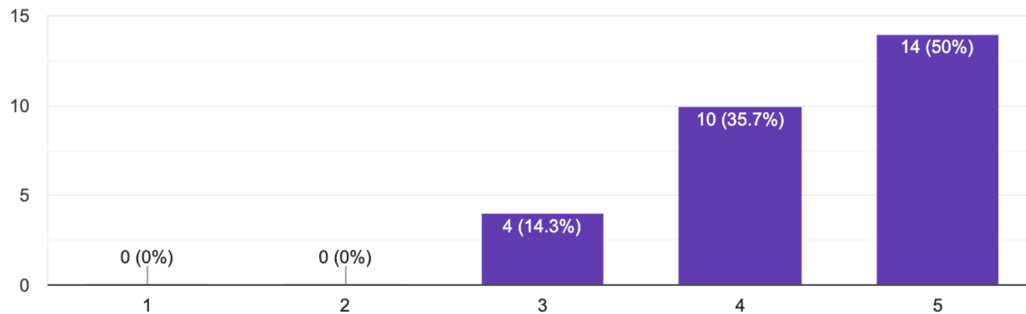

FIGURE 5.7: Evaluation of UI

The last important question with multiple choice represented for users was whether they would like to use this application. It is a major one because answers can show whether there exist potential users for such a product. In the figure 5.8 are shown that there are no negative answers. Sixty percent of users wish to use this platform, thirty percent will maybe use it, and seven percent are more inclined to use this application.

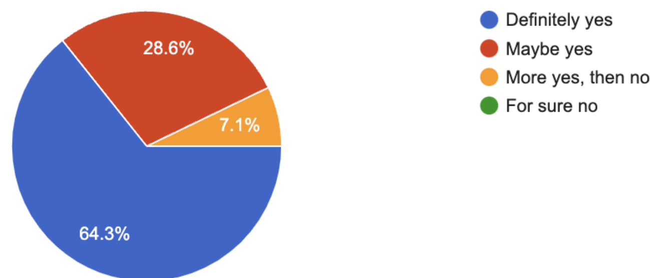

FIGURE 5.8: Possible usage of application

Moreover, there were two descriptive questions. The first one compares this platform with similar ones, such as Skyscanner, Kiwi, Rome2rio, etc. The users' answers differ, but some similar thoughts were found. First of all, some users liked that there are more filters that can help generate the trip they want to, and all nuances are taken into account. In addition to that, users liked the possibility of getting bus options in optimal path results. Some answers described the great design and straightforward flow of the application.

The second question covers things that users like and do not like in the application. The answer to it was not mandatory, but mostly all users wrote their feedback. Most of them liked the idea behind the application and the possibility of building complex trips. As for things that users do not like is the UI of the search part because not for all it was apparent, and also, there were no appropriate error messages about what parameter was entered incorrectly. Some users had problems in receiving results after tries of different search parameters. In addition to that, users complained about the relatively long time spent waiting for optimal path results. One more

thing: there was advice on what can be improved in other parts of the application from the user experience perspective.

In general, there can be concluded that most users like the idea of such an application, but there are some nuances in the work, and many things can still be improved. All the feedback was considered, and some drawbacks were already fixed, namely more deep descriptions of the fields to enter and better results output.

Chapter 6

Conclusions

6.1 Result Summary

In this thesis work were reached all the goals set at the beginning, especially the main aim to design and develop platform architecture. First of all, there were reviewed related works and algorithms that can be used to solve optimization problems. There was implemented microservices system with defined architecture, data, and math model. In addition to that, a simple UI was developed for possible user usage. The application was hosted on Amazon Web Services using its services with a defined infrastructure. The developed platform allows users to search for optimal trip plans with a simple user interface. Users can register on the application, log in, search for optimal trip plans using specific requirements, and get results.

After completing all these steps, several experiments were performed to test the system's work. These experiments showed that in the general system works and gives path results back, but some things can be improved, especially work speed. In addition, testing on real users showed that the application idea is rather great, but more deep improvements of the search service should be made.

6.2 Future improvements and work

After performing experiments on the platform and testing on real users, it was found that improving the system's operation is necessary. There should be made such future work and improvements:

- Work on increasing search part speed, improving the data load from carriers together with an algorithm that solves optimization problems faster.
- More deep exploration of algorithms for solving optimization problems for getting better results.
- Development of better error handling on the whole platform.
- Improvement of UI to make the search part more convenient for users.
- Addition of more bus providers and integration of train ones.
- Implementation of features related to the profile as the possibility of saving optimal trip plans.

Bibliography

Airlines, Airport, and Flight Routes datasets. https://www.kaggle.com/datasets/elmoallistair/airlines-airport-and-routes?select=airlines.csv.

AWS documentation. https://docs.aws.amazon.com/.

Flixbus API Reference. https://rapidapi.com/tipsters/api/flixbus/details.

Kiwi API provided by Tequila Reference. https://tequila.kiwi.com/portal/docs/tequila_api/search_api.

Kiwi website. https://www.kiwi.com/.

Marek Antosiewicz Grzegorz Koloch, Bogumił Kamiński. "Choice of best possible metaheuristic algorithm for the travelling salesman problem with limited computational time: quality, uncertainty and speed". In: *Journal of Theoretical and Applied Computer Science* 7.1 (2013), pp. 46–55.

Míča1, Ondřej. "Comparison of metaheuristic methods by solving travelling salesman problem". In: *The International Scientific Conference INPROFORUM 2015* (2015).

Omio website. https://www.omio.com/.

Rome2Rio website. https://www.rome2rio.com/.

Skyscanner website. https://www.skyscanner.com/.

Talbi, El-Ghazali. *Metaheuristics*. Wiley, 2009.

Terraform documentation. https://developer.hashicorp.com/terraform/docs.