

BLG317E - Database Systems. Term Project.

Başak Mehlepçi, ID: 150200008
DeniS Iurie Davidoglu, ID: 150200916
Mahmut Mert Özdemir, ID: 150200740

December 25, 2023

Table of Contents

Table of Contents	1
1 Responsibilities	2
2 Introduction	2
3 Software used	2
3.1 Database	2
3.2 Server	3
3.3 Client	3
4 Project folder structure	4
5 Database	5
5.1 Database sources	5
5.2 Importing from CSV	5
5.3 Table row descriptions	5
5.3.1 airlines	5
5.3.2 airports	6
5.3.3 routes	6
5.3.4 countries	6
5.3.5 planes	7
5.3.6 living_cost	7
5.3.7 users	7
5.3.8 user_history	7
5.3.9 airline_costs	7
5.4 Entity Relationship Diagram	8
6 Backend API	8
6.1 SQL scripts	8
6.1.1 Closest airport to point [Başak]	8
6.1.2 Map [Başak]	9
6.1.3 Distance between airports	9
6.1.4 Calculate indirect routes and show all details	9
6.1.5 Get airlines data from route	11
6.1.6 Get user history	11
6.2 Wrapper functions	11

6.3	HTTP requests	13
7	Frontend	14
7.1	Parent component	14
7.2	Header	15
7.3	Account	15
7.4	Map [Başak]	17
7.5	Calculator	19
7.6	Home [Başak]	20

1 Responsibilities

- Başak Mehlepçi: Home page, Map page, Map SQL scripts, Website Logo
- Denis Iuric Davidoglu: Everything else
- Mahmut Mert Özdemir: Nothing

2 Introduction

As our term project for the Database Systems course, we developed an interactive traveling cost estimation application. User can choose their start location and destination city on an interactive map, indicate the number of days, and an estimated cost of their travel including plane tickets and daily expenses is presented. The technology stack of this project is quite complex, requiring a database management system, server and client sides.

3 Software used

3.1 Database

MySQL is an open-source relational database management system, having support for all devices and operating systems. It is lightweight, easy to operate, and due to its popularity there are MySQL extensions for integrated development environments, or even dedicated graphical programs. During development, we mostly used MySQL Workbench and the terminal version of MySQL client. Some functions in the database, as well as integration with an HTTP server required some customization. After installing MySQL, the following commands must be run inside the command-line client:

```
mysql> FLUSH PRIVILEGES;
mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY '<R00tUser>';
mysql> SET GLOBAL log_bin_trust_function_creators = 1;
mysql> SET GLOBAL sql_mode=(SELECT REPLACE(@@sql_mode,'ONLY_FULL_GROUP_BY',''));
```

The Flush Privileges statement reloads the grant tables' privileges, ensuring that any changes made to user permissions are immediately applied without requiring a restart of the MySQL server. This is totally optional in our case; however, it is a good general practice. The second line contains authentication data, so that the backend server can connect to the database. The user name is set to "root", trusted address is "localhost" or "127.0.0.1", and password is "<R00tUser>". Connecting with these credentials gives full control over the database. The last two lines became necessary because some database queries and functions, important to our application, would not have worked otherwise.

3.2 Server

Initially, we thought of using Flask, which is a micro web framework written in Python. However, it had no apparent advantages over other frameworks when it comes to work with databases. Since the client side works with JavaScript, we decided to stick to this language and used it in the backend as well. Running JavaScript without a browser is possible with Node.js, an open-source runtime environment built on top of Google Chrome's V8 engine. In order to run the server, Node.js must be installed. Then, inside the **express_server** directory, **npm install** must be run from terminal. This command will install all of necessary dependencies:

- **express**: Node.js server framework for building web applications and APIs.
- **nodemon**: Development tool for hot-reloading Node.js applications during code changes.
- **mysql2**: Module for interacting with MySQL databases.
- **dotenv**: Loads environment variables from a .env file, instead of hard coding credentials.
- **shelljs**: Executes external programs and shell commands.
- **cors**: Provides Cross-Origin Resource Sharing support.
- **bcrypt**: Encrypts passwords securely.

After this installation, the server can be started on **localhost:8080** with the command **npm start**.

3.3 Client

The client code is located in **react_app** directory, and as its name suggests, it is build using React.js front-end library. In the same way as with the server, **npm install** will download all necessary packages:

- **react-router-dom**: A package used for page navigation in React applications.
- **react-helmet**: A package that allows for dynamically changing the title of a webpage.
- **react-native**: Primarily used for developing mobile applications using the React Native framework.
- **react-native-web**: Enables the use of React Native components in web applications.
- **react-horizontal-scrolling-menu**: Provides a horizontally scrolling menu component.
- **leaflet**: Used for creating interactive maps in web applications.
- **react-leaflet**: Offers React components specifically designed for working with maps in web applications.

npm start command opens the application inside the default browser running at **localhost:3000**.

4 Project folder structure

The project is organized in the following structure:

```
.
|-- express_server          Backend API server
    |-- database.js         Backend MySQL functions
    |-- index.js            HTTP request handling
    |-- package.json        Node.js dependency list
    |-- package-lock.json
|-- original_csv            Data imported to database
    |-- airlines.csv
    |-- airports.csv
    |-- cost_of_living_indices.csv
    |-- countries.csv
    |-- planes.csv
    |-- routes.csv
|-- react_app               Frontend server
    |-- package.json        Node.js dependency list
    |-- package-lock.json
    |-- public
        |-- airline_logos   Database of airline logos
        |-- favicon.ico     Application icon
        |-- index.html
        |-- manifest.json
    |-- src                 React source folder
        |-- App.css         Frontend CSS
        |-- App.js          React root component
        |-- components
            |-- footer.js
            |-- header.js    Header component with navigation buttons
        |-- images          Small images, part of pages' design
        |-- index.js
        |-- pages           React child components
            |-- account.js
            |-- admin.js
            |-- calculator.js
            |-- home.js
            |-- map.js
            |-- order.js
        |-- reportWebVitals.js
        |-- setupTests.js
|-- SCHEMA.sql              Script to create MySQL database and tables
|-- IMPORT.sql              Script to import all from CSV and create functions
```

5 Database

5.1 Database sources

Most of the tables in database come from OpenFlights.org. It contains **airlines.csv**, **airports.csv**, **countries.csv**, **routes.csv** and **planes.csv** files. The second source, which was supposed to be used for staying cost estimation, was Numbeo's Current Cost of Living Index. It compares cities across the world by several parameters, such as rent, groceries, restaurant and local purchasing power indices. Lastly, the **airlines.csv** from the first database is augmented with Airline Logos database, which has over 900 airline logos in PNG format. These datasets can be accessed from the links below:

- <https://openflights.org/data.html>
- https://www.numbeo.com/cost-of-living/rankings_current.jsp
- <https://github.com/sexym0nk3y/airline-logos>

5.2 Importing from CSV

By first running **SCHEMA.sql** and then **IMPORT.sql**, assuming that the CSV files were copied to the path accessible to MySQL, all required tables can be imported. **IMPORT.sql** also has a function for randomly generating a table called **airline_costs**.

5.3 Table row descriptions

According to the table row descriptions provided in this section, **SCHEMA.sql** was written.

5.3.1 airlines

id	Unique OpenFlights identifier for this airline.
name	Name of the airline.
alias	Alias of the airline. For example, All Nippon Airways is commonly known as "ANA".
iata	2-letter IATA code, if available.
icao	3-letter ICAO code, if available.
callsign	Airline callsign.
country	Country or territory where airport is located. See Countries to cross-reference to ISO 3166-1 codes.
active	"Y" if the airline is or has until recently been operational, "N" if it is defunct. This field is not reliable: in particular, major airlines that stopped flying long ago, but have not had their IATA code reassigned (eg. Ansett/AN), will incorrectly show as "Y".

5.3.2 airports

id	Unique OpenFlights identifier for this airport.
name	Name of airport. May or may not contain the City name.
city	Main city served by airport. May be spelled differently from Name.
country	Country or territory where airport is located. See Countries to cross-reference to ISO 3166-1 codes.
iata	3-letter IATA code. Null if not assigned/unknown.
icao	4-letter ICAO code. Null if not assigned.
latitude	Decimal degrees, usually to six significant digits. Negative is South, positive is North.
longitude	Decimal degrees, usually to six significant digits. Negative is West, positive is East.
altitude	In feet.
timezone	Hours offset from UTC. Fractional hours are expressed as decimals, eg. India is 5.5.
dst	Daylight savings time. One of E (Europe), A (US/Canada), S (South America), O (Australia), Z (New Zealand), N (None) or U (Unknown). See also: Help: Time
tz	Timezone in "tz" (Olson) format, eg. "America/Los_Angeles".
type	Type of the airport. Value "airport" for air terminals, "station" for train stations, "port" for ferry terminals and "unknown" if not known. In airports.csv, only type=airport is included.
source	Source of this data. "OurAirports" for data sourced from OurAirports, "Legacy" for old data not matched to OurAirports (mostly DAFIF), "User" for unverified user contributions. In airports.csv, only source=OurAirports is included.

5.3.3 routes

airline_name	2-letter (IATA) or 3-letter (ICAO) code of the airline.
airline_id	Unique OpenFlights identifier for airline (see Airline).
src_airport	3-letter (IATA) or 4-letter (ICAO) code of the source airport.
src_airport_id	Unique OpenFlights identifier for source airport (see Airport)
dest_airport	3-letter (IATA) or 4-letter (ICAO) code of the destination airport.
dest_airport_id	Unique OpenFlights identifier for destination airport (see Airport)
codeshare	"Y" if this flight is a codeshare (that is, not operated by Airline, but another carrier), empty otherwise.
stops	Number of stops on this flight ("0" for direct)
equipment	3-letter codes for plane type(s) generally used on this flight, separated by spaces

5.3.4 countries

name	Full name of the country or territory.
iso_code	Unique two-letter ISO 3166-1 code for the country or territory.
dafif_code	FIPS country codes as used in DAFIF. Obsolete and primarily of historical interest.

5.3.5 planes

name	Full name of the aircraft.
iata	Unique three-letter IATA identifier for the aircraft.
icao	Unique four-letter ICAO identifier for the aircraft.

5.3.6 living_cost

city	City
country	Country
slug	Short name
currency	Currency code in three characters
avg_index	Overall living index (0%-100%)
rent_index	Rent Index (0%-100%)
groceries_index	Groceries Index (0%-100%)
restaurant_index	Restaurant Price Index (0%-100%)
purchasing_index	Local Purchasing Power Index (0%-100%)
id	Unique identifier for each city

5.3.7 users

email	User's email, primary key
password_hash	User's encrypted password
first_name	User's first name
last_name	User's last name
age	User's age
interests	Each bit of this integer indicates the presence of a particular interest

5.3.8 user_history

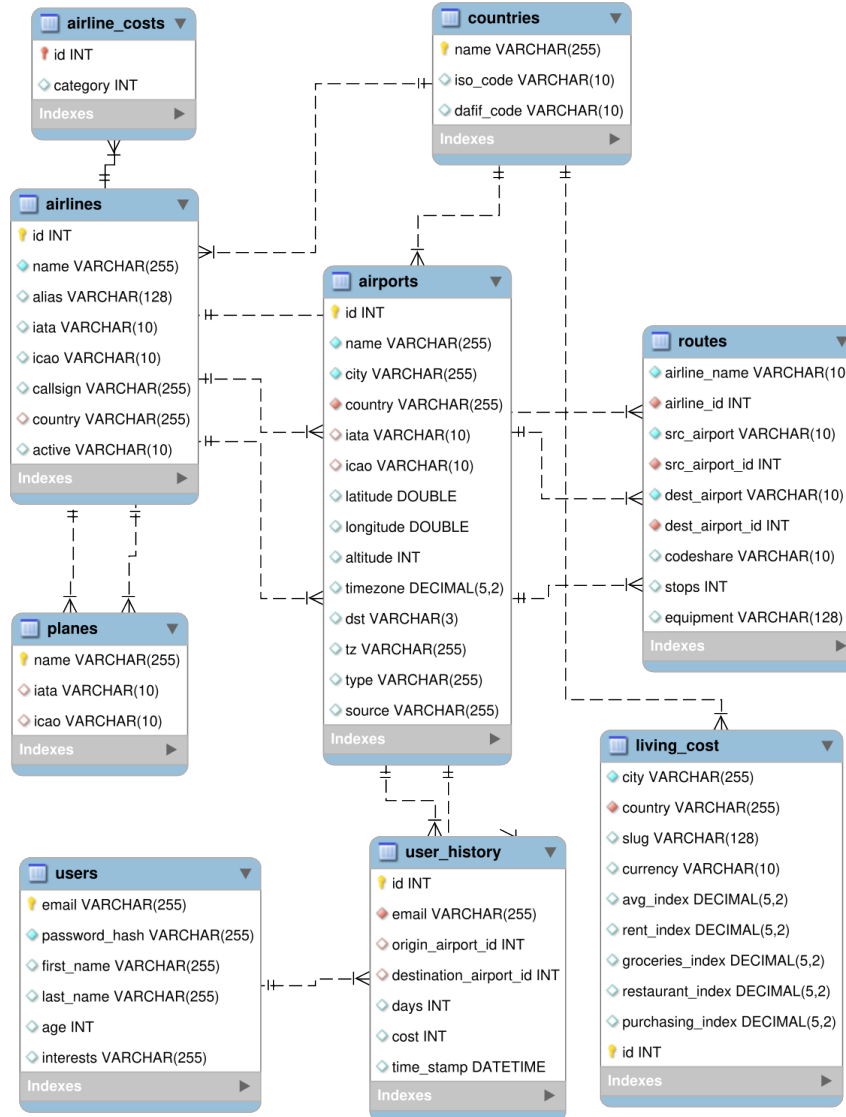
id	Unique id for each history entry
email	Email referring to a registered user
origin_airport_id	Origin airport id, referring to airports table
destination_airport_id	Destination airport id, referring to airports table
days	Number of days of stay
cost	Estimated cost of traveling
time_stamp	Date and time of saving the history entry

5.3.9 airline_costs

id	Unique identifier referring to airlines table
category	Number from 1 to 5, where less means more expensive airline

5.4 Entity Relationship Diagram

In total, there are 9 tables, interconnected in such a way:



6 Backend API

6.1 SQL scripts

6.1.1 Closest airport to point [Başak]

```
delimiter $$
```

```
create function closest_airport(p_lat double, p_long double) returns int
deterministic
begin
return (select id
from airports
order by ST_Distance_Sphere(
point(longitude, latitude),
point(p_long, p_lat)
) limit 1);
```



```
end$$
delimiter ;
```

6.1.2 Map [Başak]

```
select id, latitude as lat, longitude as lng, name
from airports
where id = closest_airport(47, 28.9);
```

6.1.3 Distance between airports

The script creates a function which takes two airport identifiers (a and b) and returns the geographic distance between them. **ST_Distance_Sphere(Point, Point)** is MySQL's native function for calculating distance between geographic points. Given latitude and longitude as double-precision values, they can be transformed into **Point** data type using **point(long, lat)** function. Because all that is given is airport IDs, their corresponding latitude and longitude coordinates are retrieved using the **SELECT** statement.

```
delimiter $$
create function distance_between_airports(a int, b int) returns int
begin
return (select ST_Distance_Sphere(
point((select longitude from airports where id = a),
      (select latitude from airports where id = a)),
point((select longitude from airports where id = b),
      (select latitude from airports where id = b))
));
end$$
delimiter ;
```

6.1.4 Calculate indirect routes and show all details

The following script is quite long and it generates results with a noticeable delay. However, its output represents a very rich information, specifically the list of direct and indirect routes. It outputs airport IDs that are part of the route, city and country names where airports are located and the route's total distance.

First, the script generates an intermediate table inside the **WITH** clause. Regardless of the number of stops, all routes are in the same table of 5 columns: **a0**, **a1**, **a2**, **a3** and **distance**, where **ax** is airport **x**'s identifier. For direct flights, only two airport IDs will be shown, and the places for other airports will get an invalid value, -1. This is done using **CASE** expressions. The number of airports is always at least two, so there is nothing to be checked for **a0** and **a1**. For **a2**, the ID is irrelevant when the route ends on airport 1. The case expression for **a2** can be read like this: in case when the destination airport has been found at level 0 (direct flight), then **a2** is irrelevant and takes -1; otherwise, the airport found at level 1 (indirect flight with 1 exchange) is relevant. For **a3** to be included, airports found at level 0 and level 1 must not be equal to final destination. **a3** is the result of level 2 evaluation (indirect flight with 2 exchanges).

The last column is distance, which is calculated again according to the relevancy of airport IDs. The **CASE** expression switches between different formulas. When all airport IDs are relevant, distances between all adjacent points are included, and when the flight is direct, the total distance is the distance between two airports. Distance is important in sorting rows, because shortest paths are usually cheaper and more relevant.

Different levels are obtained by joining the **routes** table onto itself two times. **lvl0** instance is left-joined with **routes** to create **lvl1**, which is again left-joined with another **routes** instance, resulting in **lvl2**. Out of this big table, the only rows relevant are those which lead to **@destination**, on any level.

Since irrelevant airports are hidden with -1, there can be multiple same rows. To handle this issue, **SELECT DISTINCT** is used. The rows are ordered by distance and are limited, because there can be too many results.

In second part, the intermediate table is expanded with city and country names of each stop. This final version is what the web application will show to the user: routes consisting of readable sequences of cities and countries.

```
select 344 into @source;
select 1688 into @destination;

with intermediate as
(select distinct
lvl0.src_airport_id as a0,
  lvl1.src_airport_id as a1,
  case when (
    lvl0.dest_airport_id = @destination
  ) then -1
  else lvl1.dest_airport_id end as a2,
  case when (
    lvl0.dest_airport_id = @destination or
    lvl1.dest_airport_id = @destination
  ) then -1
  else lvl2.dest_airport_id end as a3,
  case
when lvl0.dest_airport_id = @destination then (
select distance_between_airports(lvl0.src_airport_id, lvl0.dest_airport_id)
)
when lvl1.dest_airport_id = @destination then (
(select distance_between_airports(lvl0.src_airport_id, lvl0.dest_airport_id)) +
(select distance_between_airports(lvl1.src_airport_id, lvl1.dest_airport_id))
)
when lvl2.dest_airport_id = @destination then (
(select distance_between_airports(lvl0.src_airport_id, lvl0.dest_airport_id)) +
(select distance_between_airports(lvl1.src_airport_id, lvl1.dest_airport_id)) +
(select distance_between_airports(lvl2.src_airport_id, lvl2.dest_airport_id))
)
else 2147483647
  end as distance
from routes as lvl0
left join routes as lvl1 on lvl0.dest_airport_id = lvl1.src_airport_id
left join routes as lvl2 on lvl1.dest_airport_id = lvl2.src_airport_id
where (lvl0.src_airport_id = @source and lvl0.dest_airport_id = @destination) or
  (lvl0.src_airport_id = @source and lvl1.dest_airport_id = @destination) or
  (lvl0.src_airport_id = @source and lvl2.dest_airport_id = @destination)
order by distance
limit 20)
```

```

select intermediate.distance,
    a0.city as 'airport0_city', a0.country as 'airport0_country',
    a1.city as 'airport1_city', a1.country as 'airport1_country',
    a2.city as 'airport2_city', a2.country as 'airport2_country',
    a3.city as 'airport3_city', a3.country as 'airport3_country',
    intermediate.a0 as 'airport0_id',
    intermediate.a1 as 'airport1_id',
    intermediate.a2 as 'airport2_id',
    intermediate.a3 as 'airport3_id'
from airports a0, airports a1, airports a2, airports a3, intermediate
where (a0.id = intermediate.a0 and
    a1.id = intermediate.a1 and
    a2.id = intermediate.a2 and
    a3.id = intermediate.a3);

```

6.1.5 Get airlines data from route

This script works on results obtained from the indirect routes script. It gets the list of airline names, ICAO codes and cost category numbers, for a pair of airports. This script is not combined with already hefty route finding script, it is instead used by the backend server. The backend program has more control and is able to select only a portion of routes, thus limiting the quickly growing airline list.

```

select airlines.name, airlines.icao, airline_costs.category
from routes, airlines, airline_costs
where airlines.id = routes.airline_id and
    airline_costs.id = routes.airline_id and
    routes.src_airport_id=344 and
    routes.dest_airport_id=1688;

```

6.1.6 Get user history

The user should be able to access their own history data in the account page, and this script is exactly for this. It prepares the data in a readable format, with origin and destination columns being in "CITY, COUNTRY" format, containing the number of days, cost and even the time stamp formatted nicely. In this way, every piece of data the database holds is provided, respecting the users' freedom.

```

select CONCAT(src.city, ", ", src.country) as origin,
    CONCAT(dest.city, ", ", dest.country) as destination,
    days, cost, date_format(time_stamp, '%Y-%m-%d %H:%i:%s') as "time_stamp"
from user_history, airports src, airports dest
where (src.id = origin_airport_id and
    dest.id = destination_airport_id and
    email = 'example@gmail.com');

```

6.2 Wrapper functions

On top of the SQL scripts described above, the backend API uses some trivial select and insert commands, which were omitted. However, the shorter scripts can be still mentioned in the context of wrapper functions. **express_server/database.js** is the file where all communication

between the backend and MySQL server takes place. It connects to the database using environmental variables inside **express_server/.env**:

```
MYSQL_USER='root'  
MYSQL_HOST='localhost'  
MYSQL_PASSWORD='<R00tUser>'  
MYSQL_DATABASE='traveling_cost'
```

Instead of keeping only one connection, **database.js** creates a pool of connections using these credentials, which allows more flexible querying. Every query function is structured similar to this example:

```
export async function getCityCountry(airport_id) {  
  try {  
    const [result] = await pool.query(  
      `select city, country from airports  
      where id = ?`,  
      airport_id  
    );  
    return result;  
  } catch(e) {  
    console.log(e.message);  
    return false;  
  }  
}
```

Some functions retrieve or post user's sensitive data, and additional checks must be performed to confirm the someone's identity. Before that, let's have a look at a new user creation. This function check for the password length, hashes the password and inserts everything to the database:

```
export async function newUser(user_data) {  
  if (user_data[1].length < 5)  
    throw new Error("Password shorter than 5 characters");  
  const hashed_password = await hashPassword(user_data[1]);  
  const [result] = await pool.query(  
    `insert into users values  
    (?, ?, ?, ?, ?, ?);`,  
    [user_data[0], hashed_password, user_data[2], user_data[3],  
     parseInt(user_data[4]), parseInt(user_data[5])]  
  );  
  return result;  
}
```

Then, there is a login function, used to verify someone's identity. The password provided is hashed and compared with the one in the database:

```
export async function login(login_data) {  
  const [[result]] = await pool.query(  
    `select password_hash from users where email=?;`,  
    login_data[0]  
  );  
  const isMatch = await bcrypt.compare(login_data[1], result["password_hash"]);  
  return isMatch;  
}
```

Finally, each function that accesses, uploads or deletes user's private data goes through the authentication process, as in this example:

```
export async function getUser(login_data) {
  const authenticated = await login(login_data);
  if (authenticated) {
    const [[result]] = await pool.query(
      `select email, first_name, last_name, age, interests
       from users where email=?`;
    login_data[0]
  );
  return result;
}
```

The biggest function in the **express_server/database.js** is **getPlaneOffers(source_id, destination_id)**. It calls the indirect route calculation script, and as long as the number of saved offers is not exceeded, each route gets expanded into concrete offers by airlines. For direct flights, there is no limit, but for indirect ones there is a limit of 5 + 5 offers (for one exchange and two exchanges). The selection of indirect flights is done randomly. The return value of this function is an array of offers, where each offer consists of number of stops, airline name, airline ICAO code, list of cities through which the route passes, and the price. Price is estimated based on the airline's cost category and distance between airports.

6.3 HTTP requests

To communicate with client, Express.js framework is used to handle various HTTP requests. In **express_server/index.js**, a couple of services are defined, used as API by the client. In this example, server process a GET request for plane offers, taking source and destination airport IDs as parameters. **db** is object containing all functions exported from **express_server/database.js**. If database responds without errors, client is transmitted the result together with status 200 (OK). If something goes wrong, client is informed with status 400 (Bad Request).

```
app.get("/plane_offers/:source/:destination", async function(req, res) {
  console.log("/plane_offers" + req.params.source + " " + req.params.destination);
  let result = await db.getPlaneOffers(req.params.source, req.params.destination);
  if (result === false)
    res.status(400).send(result);
  else
    res.status(200).send(result);
});
```

Here is a complete list of all backend API:

Method	URL	Description
POST	http://localhost:8080/register	Register new account
POST	http://localhost:8080/login	Login into existing account
DELETE	http://localhost:8080/delete_user	Delete account
POST	http://localhost:8080/user_data	Get all user data except password
POST	http://localhost:8080/user_history	Get all user history
GET	http://localhost:8080/plane_offers/:source/:destination	Get plane offers between two airports
GET	http://localhost:8080/city_country/:airport	Get airport's city and country names
POST	http://localhost:8080/post_history	Save a history entry
POST	http://localhost:8080/closest_airport	Get closest airport to a point on map

7 Frontend

7.1 Parent component

React parent component is defined in **react_app/src/App.js**. It includes all child component files, routes to different pages, keeps global states, provides inter-page communication and even can make fetch requests to backend server. The way the application renders is defined in the return statement of the **App()** functional component:

```
return (
  <div>
    <Router>
      <Header account_name={account_name}/>
      <Routes>
        <Route path="/" element={<Home/>}/>
        <Route path="/home" element={<Home/>}/>
        <Route path="/map" element=
          {<Map onMarker1={(result) => {setAirport1(result)}}
            onMarker2={(result) => {setAirport2(result)}}/>}/>
        <Route path="/calculator" element=
          {<Calculator origin={airport1} destination={airport2}
            user_credentials={user_credentials}/>}/>
        <Route path="/account" element=
          {<Account user_data={user_data} user_credentials={user_credentials}
            onLogin={(result) => {setUserCredentials(result)}}/>}/>
      </Routes>
    </Router>
  </div>
);
```

The contents of Router tag are a static Header component, containing the navigation buttons, and a dynamic portion which is replaced by a page, based on URL. There are a couple of states here. **account_name** state is passed to Header to render user's name, purely for design purposes. This state is updated every time Account component is triggered to update user credentials, on an **onLogin** event. **user_credentials** is a state on its own, shared between the Parent, Account, and Calculator components. Map component can update Parent's **airport1** and **airport2** states on marker change events. These states are needed for Calculator. In this design, map communicates the airport IDs to the calculator page by means of the Parent. Calculator also needs **user_credentials**,

because it saves calculations to the database, and since it is private data, our backend would demand authentication.

7.2 Header

Header component resides **react_app/src/components/header.js**. It only renders navigation buttons and the website logo. On the **Account** button, text is changed from "Account" to the user's first name once logged in.

```
<img src={voyify_logo} />
<nav>
  <ul>
    <li><NavLink className="button" to="home"><img/>Home</NavLink></li>
    <li><NavLink className="button" to="map"><img/>Map</NavLink></li>
    <li><NavLink className="button" to="calculator"><img/>Calculator</NavLink></li>
    <li><NavLink className="button" to="order"><img/>Order</NavLink></li>
    <li><NavLink className="button" to="admin"><img/>Admin</NavLink></li>
    <li><NavLink className="button_img" to="account"><img src={user_icon}/>
      {props.account_name}</NavLink></li>
  </ul>
</nav>
```

Logo, designed by Başak:



Account button change after logging in:



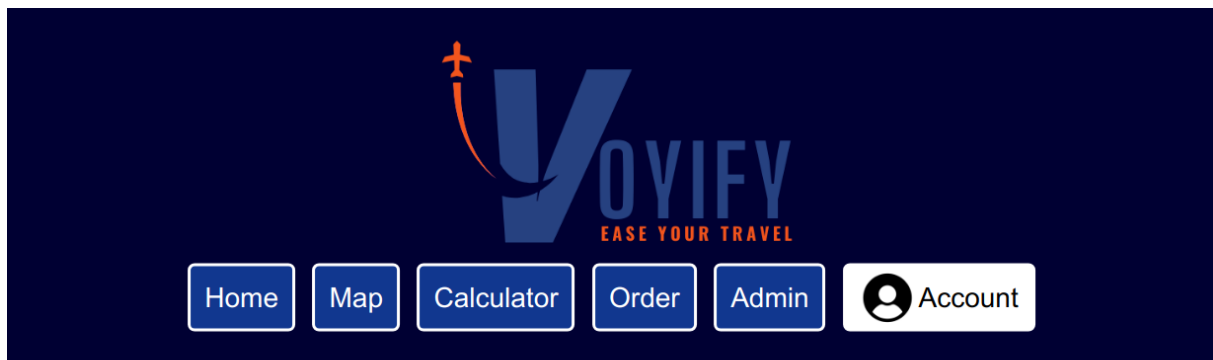
7.3 Account

Account page can be in one of two states: registration/login state or logged in state, where user's history can be examined. Account can make fetch requests to the backend API using the following methods:

- registerSubmit(e)
- loginSubmit(e)
- deleteAccount()

- `getHistory()`

Here are screenshots of the Account page:



Register

General

Email: *

Password: *

First name: *

Last name: *

Age: *

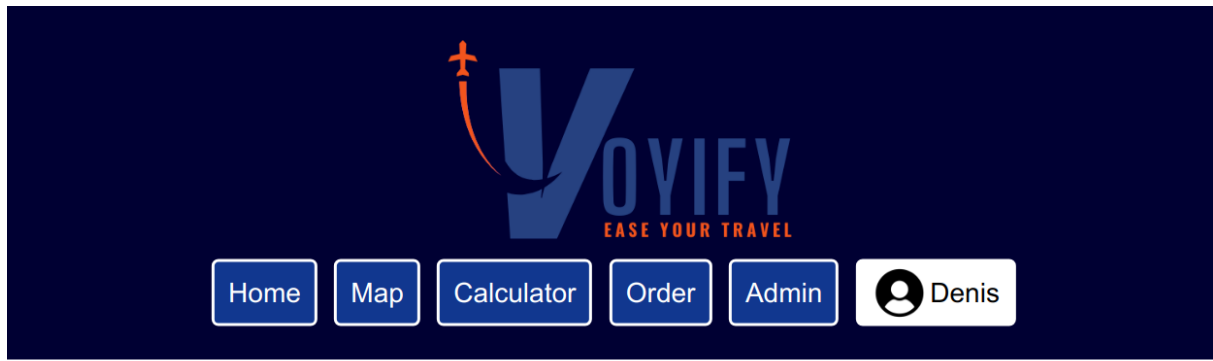
Login

Email: *

Password: *

Interests

- ☐ Museums
- ☐ World Cuisines
- ☐ Architecture
- ☐ Cultural Events
- ☐ Nature
- ☐ Shopping
- ☐ Alcohol
- ☐ Music Festivals
- ☐ History
- ☐ Art Galleries
- ☐ Cycling Adventures



Welcome, Denis Davidoglu

History

Origin	Destination	Days	Cost	Time stamp
Karlovy Vary, Czech Republic	Cluj- napoca, Romania	0	3750	2023-12-25 04:53:59
Ontario, United States	Ottawa, Canada	32	3176	2023-12-25 04:56:19
Casablanca, Morocco	Simferopol, Ukraine	3	1606	2023-12-25 04:56:47
Ashkhabad, Turkmenistan	Phnom-penh, Cambodia	5	5520	2023-12-25 04:57:14
Belem, Brazil	Rio Negro, Colombia	2	4034	2023-12-25 04:58:16

Logout

Delete account

7.4 Map [Başak]

On the Map page, we have developed a frontend-centric interface. Turning our attention to the SQL side, we utilized the ‘closest_airport(p_lat double, p_long double)’ function. This function takes latitude (‘p_lat’) and longitude (‘p_long’) as parameters and retrieves information from the ‘airports’ table to find the closest airport to the specified location. The closest_airport(p_lat DOUBLE, p_long DOUBLE) and distance_between_airports(a INT, b INT) SQL functions are responsible for distance calculations between airports and finding the closest airport to a given geographical point. ‘distance_between_airports’ calculates the distance between two airports based on their coordinates. ‘closest_airport’ identifies the nearest airport to a specified latitude and longitude.

```
; React Map Component
import React, { useState, useEffect, useRef } from 'react';
import { MapContainer, TileLayer, Marker, Popup, Polyline } from 'react-leaflet';
; Other imports...

function Map(props) {
  ; Component implementation...
}
```

```
export default Map;
```

This React component uses Leaflet for mapping features. It provides an interactive map where users can set markers and visualize travel routes. The component uses state variables and effects for managing marker positions and fetching airport data.

```
; Event Handling - Marker Interaction
const LocationFinderDummy = () => {
  const map = useMapEvents({
    click(e) {
      ; Marker setting logic...
    },
  });
  return null;
};

; Data Fetching - closest_airport Function
useEffect(() => {
  const data = { position: { lat: marker1[0], lng: marker1[1] } };
  fetch('http://localhost:8080/closest_airport', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(data)
  }).then(response => response.json()).then(data => {
    setData1(data);
    props.onMarker1(data["id"]);
    const marker = markerRef1.current;
    if (marker) {
      marker.openPopup();
    }
  });
}, [marker1]);
```

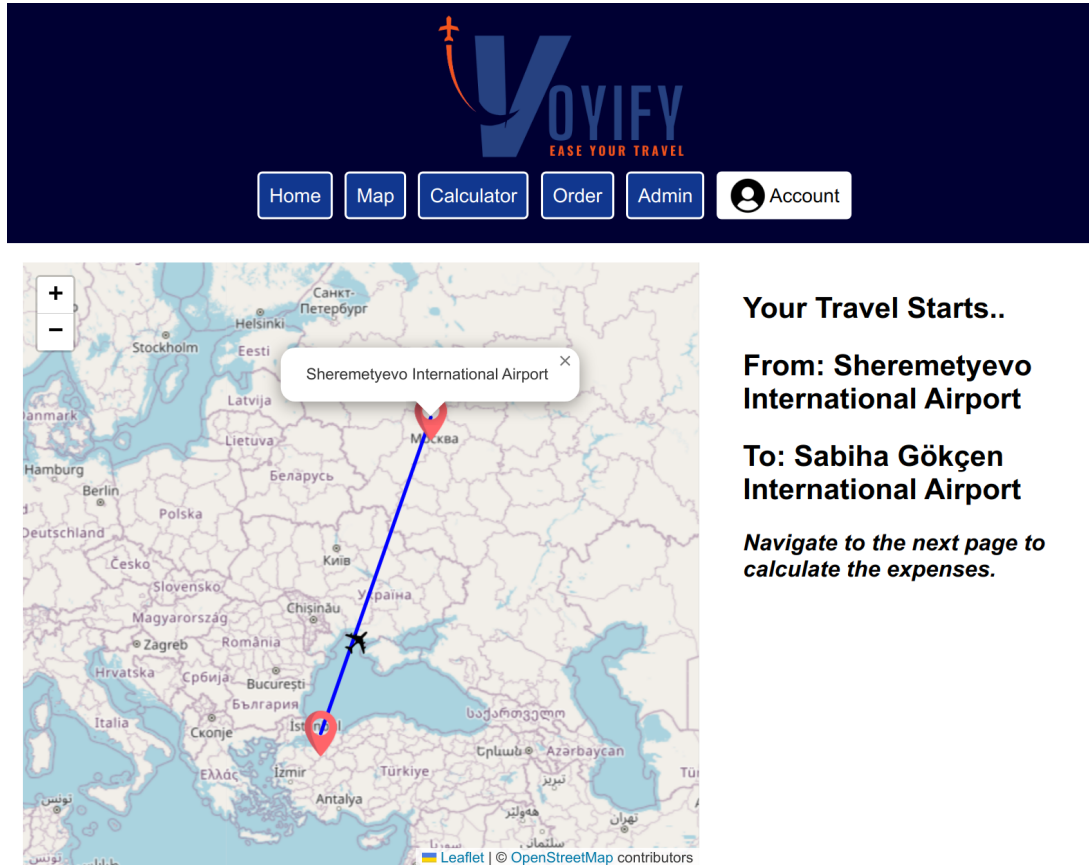
The event handling logic enables users to set markers on the map. The ‘LocationFinderDummy’ component uses the ‘useMapEvents’ hook to detect click events. Data fetching is demonstrated using the ‘closest_airport’ function, which is triggered when the marker position changes.

```
; Map Rendering - Leaflet Components
<MapContainer>
  ; Other components...
  <Marker icon={customIcon} position={marker1} ref={markerRef1}>
    <Popup>{data1["name"]}</Popup>
  </Marker>
  ; Other components...
</MapContainer>

; User Information Display
<div>
  <h2>Your Travel Starts..</h2>
  <h2>From: {data1 && data1["name"]}</h2>
```

```
<h2>To: {data2 && data2["name"]} </h2>
</div>
```

This section involves rendering the map and markers using Leaflet components. Custom icons are used for markers. Popup information is displayed for each marker. User information is dynamically updated based on selected airports.



7.5 Calculator


Calculator takes two airport ID numbers, and displays a list of possible flights with their corresponding airlines. In a horizontal scrolling menu, cards with all details, such as airline logos, airlines names, stops and price are displayed. Number of days can be entered here as well. In the bottom, there is a submit button, which saves the calculated result.

Calculator

Bucharest, Romania - Chisinau, Moldova



Number of days:

Airline: Tarom
Direct flight





Price: \$380

Airlines: Tarom,Air Moldova
Exchange at Istanbul

Price: \$1243

Airlines: Tarom,Turkish Airlines
Exchange at Istanbul

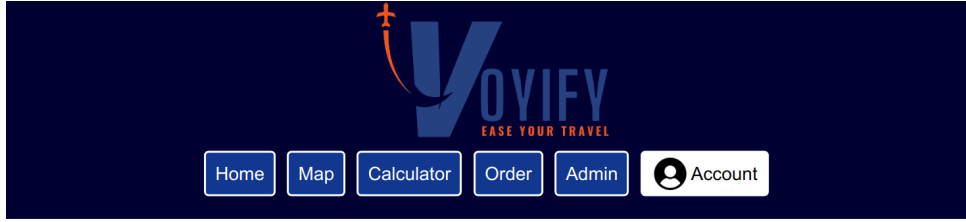



Price: \$728

Total: \$383

Save

7.6 Home [Başak]



Welcome to Voyify!

Start by getting registered or logging in.



Embark on a journey of seamless travel planning with Voyify, where your travel dreams come to life! Say goodbye to the hassle of coordinating itineraries and let Voyify transform your ideas into unforgettable adventures. Whether you're a solo explorer, a couple seeking a romantic getaway, or a group of friends ready for a new escapade, Voyify is your go-to app for personalized travel experiences.