Full stack Development With MERN

A Project Documentaion

Project title:- Flight Finder

Team Members	Roles
Mattaparti Lavanya	Frontend Developer
Lakkakula Gayathri	Backend Developer and Database Developer
Kopparhti D V S Bharadwaja	Admin DashBoard Developer
Kurapati Sri Lakshmi	Documentation and testing

Flight Booking APP

2.project overview

Purpose:

SB Flights is a next-generation digital platform designed to transform the way you book and manage flight tickets. Whether you're a frequent flyer or an occasional traveler, SB Flights brings simplicity, speed, and convenience to your travel planning.

Our intuitive web application makes it easy to find and book the perfect flight. Simply enter your travel dates, destinations, number of passengers, and basic details — and receive instant ticket confirmation. No more long queues or confusing systems.

Imagine having comprehensive flight details right at your fingertips. From departure and arrival times to flight classes and available amenities, SB Flights provides all the essential information you need to make informed decisions. No more second-guessing or uncertainty—every aspect of your travel is made crystal clear, ensuring complete confidence in your booking.

The booking process itself is designed to be as simple and streamlined as possible. Just enter your name, age, preferred travel dates, departure and arrival cities, and the number of passengers. Once you submit your booking request, you'll receive instant confirmation of your reservation. Say goodbye to long queues and complex reservation systems—SB Flights makes booking your next journey quick, easy, and hassle-free.

Upon successful booking, you'll gain access to our dedicated Booking Details page, which becomes your personal travel companion. This page offers a comprehensive overview of all your current and previous bookings, enabling you to effortlessly manage your travel plans and stay organized. With SB Flights, your essential travel information is always just a click away, supporting a stress-free and well-managed journey.

But SB Flights isn't just built for travelers—it also includes powerful tools for flight service administrators. Our intuitive Admin Dashboard allows administrators to efficiently manage ticket reservations. They can easily view a list of all available flights open for booking, monitor ongoing and past reservations, and maintain complete control over the booking process. Each flight service has its own separate login and registration pages, ensuring privacy and security for both administrators and users.SB Flights is here to enhance your travel experience by providing a seamless and convenient way to book flight tickets. With our user-friendly interface, efficient booking management, and robust administrative features, we ensure a hassle-free and enjoyable flight ticket booking experience for both users and flight administrators alike.

Get ready to embark on a new era of flight travel with SB Flights – your ticket to effortless booking and unforgettable journeys.

Features of the Flight Finder Application

Flight Search:

- Search flights by source, destination, departure & return dates.
- Option to select one-way, round-trip, or multi-city.

Filter & Sort:

- Filter results by price, number of stops, airlines, flight duration, or departure time.
- Sort flights by lowest price, shortest duration, or earliest departure.

View Flight Details:

- See complete flight information including baggage rules, layover times, aircraft type, and cancellation policies.

Easy Booking:

- Book flights by entering passenger details and selecting seats if available.
- Secure payment gateway integration for safe transactions.

Booking Management:

- View upcoming and past bookings.
- Option to cancel or reschedule bookings based on airline policies.
- Download or email tickets.

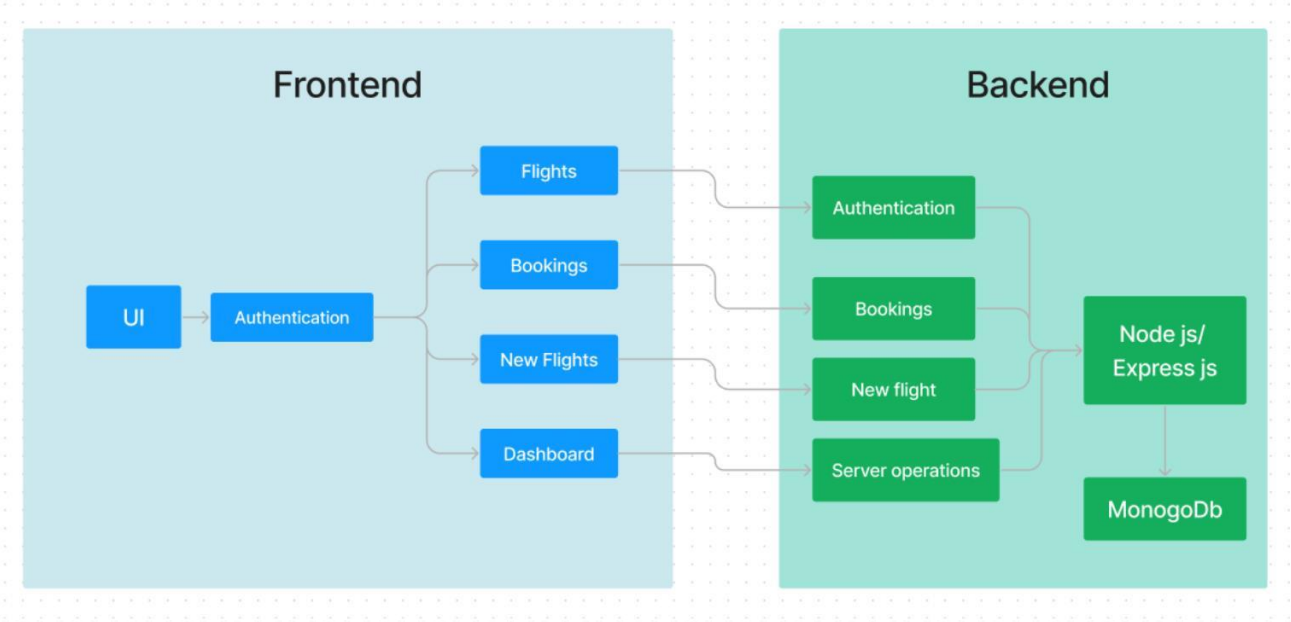
Notifications & Alerts:

- Email and in-app notifications for booking confirmations, schedule changes, or price drops

Optional Chat / Support:

- Real-time chat or helpdesk integration for customer support.

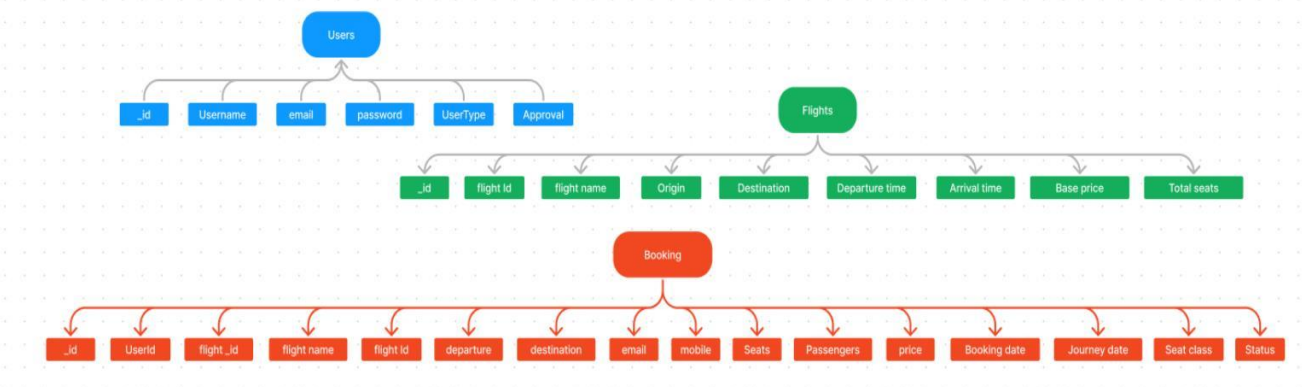
3. ARCHITECTURE:



In this architecture diagram:

- The frontend is represented by the "Frontend" section, including user interface components such as User Authentication, Flight Search, and Booking.
- The backend is represented by the "Backend" section, consisting of API endpoints for Users, Flights, Admin and Bookings. It also includes Admin Authentication and an Admin Dashboard.
- The Database section represents the database that stores collections for Users, Flights, and Flight Bookings.

ER DIAGRAM:



The flight booking ER-diagram represents the entities and relationships involved in a flight booking system. It illustrates how users, bookings, flights, passengers, and payments are interconnected. Here is a breakdown of the entities and their relationships:

USER: Represents the individuals or entities who book flights. A customer can place multiple bookings and make multiple payments.

BOOKING: Represents a specific flight booking made by a customer. A booking includes a particular flight details and passenger information. A customer can have multiple bookings.

FLIGHT: Represents a flight that is available for booking. Here, the details of flight will be provided and the users can book them as much as the available seats.

ADMIN: Admin is responsible for all the backend activities. Admin manages all the bookings, adds new flights, etc.,

Features:

1. **Extensive Flight Listing:** SB Flights offers an extensive list of flight services, providing a wide range of routes and options for travelers. You can easily browse through the list and explore different flight journeys, including departure and arrival times, flight classes, and available amenities, to find the perfect travel option for your journey.

2. **Book Now Button:** Each flight listing includes a convenient "Book Now" button. When you find a flight journey that suits your preferences, simply click on the button to proceed with the reservation process.

3. **Booking Details**: Upon clicking the "Book Now" button, you will be directed to a booking details page. Here, you can provide relevant information such as your preferred travel dates, departure and arrival stations, the number of passengers, and any special requirements you may have.

4. **Secure and Efficient Booking Process:** SB Flights ensures a secure and efficient booking process. Your personal information will be handled with the utmost care, and we strive to make the reservation process as quick and hassle-free as possible.

5. **Confirmation and Booking Details Page:** Once you have successfully made a reservation, you will receive a confirmation message. You will then be redirected to a booking details page, where you can review all the relevant information about your booking, including your travel dates, departure and arrival stations, the number of passengers, and any special requirements you specified.

In addition to these user-facing features, SB Flights provides a powerful admin dashboard, offering administrators a range of functionalities to efficiently manage the system. With the admin dashboard, admins can add and manage multiple flight services, view the list of available flights, monitor user activity, and access booking details for all flight journeys.

SB Flights is designed to enhance your flight travel experience by providing a seamless and user-friendly way to book flight tickets. With our efficient booking process, extensive flight listings, and robust admin dashboard, we ensure a convenient and hassle-free flight ticket booking experience for both users and flight administrators alike.

4.Setup Instructions

PREREQUISITES AND INSTALLATION:

To develop a full-stack flight booking app using React JS, Node.js, and MongoDB, there are severalprerequisites you should consider. Here are the key prerequisites for developing such an application:

Node.js and npm: Install Node.js, which includes npm (Node Package Manager), on your developmentmachine. Node.js is required to run JavaScript on the server side.

- Download: https://nodejs.org/en/download/
- Installation instructions: https://nodejs.org/en/download/package-manager/

MongoDB: Set up a MongoDB database to store hotel and booking information. Install MongoDB locally oruse a cloud-based MongoDB service.

- Download: https://www.mongodb.com/try/download/community
- Installation instructions: https://docs.mongodb.com/manual/installation/

Express.js: Express.js is a web application framework for Node.js. Install Express.js to handle server-side routing,middleware, and API development.

- Installation: Open your command prompt or terminal and run the following command: **npm installexpress**

React.js: React.js is a popular JavaScript library for building user interfaces. It enables developers to create interactive and reusable UI components, making it easier to build dynamic and responsive web applications. To install React.js, a JavaScript library for building user interfaces, follow the installation guide: https://reactjs.org/docs/create-a-new-react-app.html

HTML, CSS, and JavaScript: Basic knowledge of HTML for creating the structure of your app, CSS for styling,and JavaScript for client-side interactivity is essential.

Database Connectivity: Use a MongoDB driver or an Object-Document Mapping (ODM) library like Mongoose toconnect your Node.js server with the MongoDB database and perform CRUD (Create, Read, Update, Delete) operations.

Front-end Framework: Utilize Angular to build the user-facing part of the application, including productslistings, booking forms, and user interfaces for the admin dashboard.

Version Control: Use Git for version control, enabling collaboration and tracking changes throughout thedevelopment process. Platforms like GitHub or Bitbucket can host your repository.

- Git: Download and installation instructions can be found at: https://git-scm.com/downloads

Development Environment: Choose a code editor or Integrated Development Environment (IDE) that suits yourpreferences, such as Visual Studio Code, Sublime Text, or WebStorm.

- Visual Studio Code: Download from https://code.visualstudio.com/download
- Sublime Text: Download from https://www.sublimetext.com/download
- WebStorm: Download from https://www.jetbrains.com/webstorm/download

To Connect the Database with Node JS go through the below provided link:

- Link: https://www.section.io/engineering-education/nodejs- mongoosejs-mongodb/

To run the existing Flight Booking App project downloaded from github:

Follow below steps:

Clone the repository:

- Open your terminal or command prompt.
- Navigate to the directory where you want to store the e-commerce app.
- Execute the following command to clone the repository:

  Git clone: https://github.com/L-Gayathri/Flight_Finder

Install Dependencies:

- Navigate into the cloned repository directory:
  cd Flight-Booking-App-MERN

- Install the required dependencies by running the following command:
  npm install

Start the Development Server:

- To start the development server, execute the following command:
  npm run dev or npm run start

- The e-commerce app will be accessible at http://localhost:3000 by default. You can change the portconfiguration in the .env file if needed.

Access the App:

- Open your web browser and navigate to http://localhost:3000.
- You should see the flight booking app's homepage, indicating that the installation and setup weresuccessful.

You have successfully installed and set up the flight booking app on your local machine. You can now proceedwith further customization, development, and testing as needed.

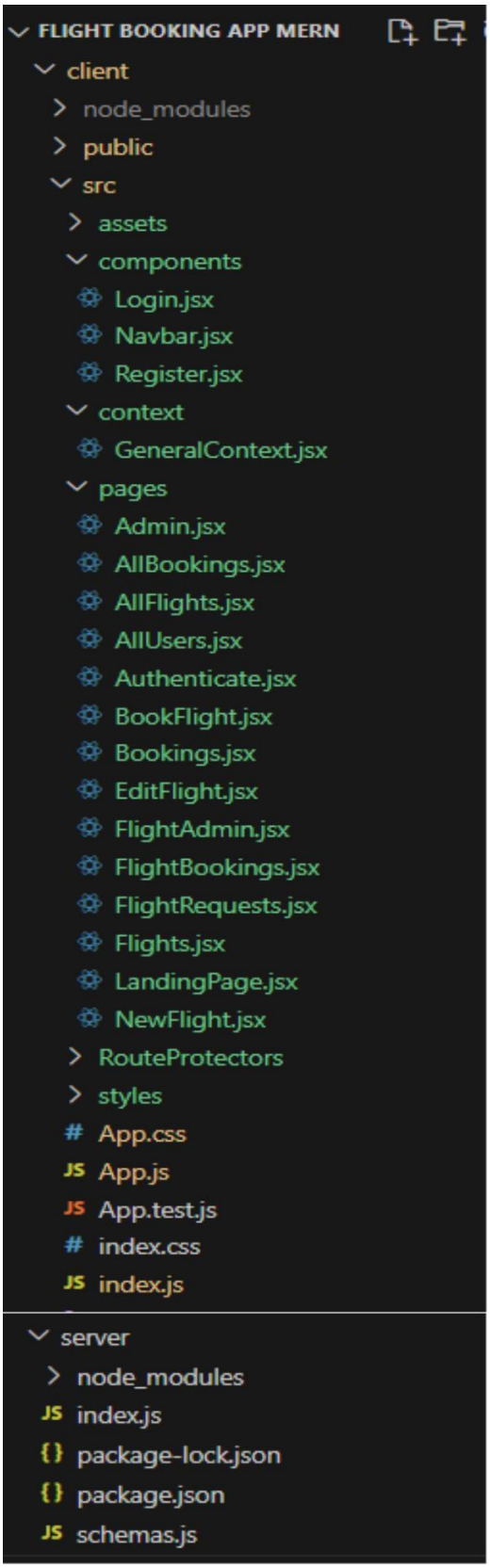
USER & ADMIN FLOW:

1. **User Flow:**
    - Users start by registering for an account.
    - After registration, they can log in with their credentials.
    - Once logged in, they can check for the availability of flights in their desired route and dates.
    - Users can select a specific flight from the list.
    - They can then proceed by entering passenger details and other required data.
    - After booking, they can view the details of their booking.

2. **Flight Operator Flow:**
    - Flight operator start by logging in with their credentials.
    - Once logged in, they are directed to the Flight operator Dashboard.
    - Flight Operator can access the Dashboard, where they can view bookings, add new flight routes, etc.,

3. **Admin Flow:**
    - Admins start by logging in with their credentials.
    - Once logged in, they are directed to the Admin Dashboard.
    - Admins can access the Flight Booking Admin Dashboard, where they can view bookings, approve new flight operators, etc.,

5.FOLDER STRUCTURE:



This structure assumes a React app and follows a modular approach. Here's a brief explanation of the main directories and files:

- src/components: Contains components related to the application such as, register, login, home, bookings, etc..
- src/pages has the files for all the pages in the application.

Project Flow:

Milestone 1: Project Setup and Configuration:

1. **Install required tools and software:**
   - Node.js.
   - MongoDB.
   - React Js.
   - Git.
2. **Create project folders and files:**
   - Client folders.
   - Server folders

Milestone 2: Backend Development:

1. **Setup express server:**
   - Install express.
   - Create index.js file.
   - Define API’s
2. **Configure MongoDB:**
   - Install Mongoose.
   - Create database connection.
3. **Implement API end points:**
   - Implement CRUD operations.
   - Test API endpoints.

Milestone 3: Web Development:

1. **Setup React Application:**
    - Create React app in client folder.
    - Install required libraries
    - Create required pages and components and add routes.
2. **Design UI components:**
    - Create Components.
    - Implement layout and styling.
    - Add navigation.
3. **Implement frontend logic:**
    - Integration with API endpoints.
    - Implement data binding.

Create database in cloud video link:-
https://drive.google.com/file/d/1CQil5KzGnPvkVOPWTLP0h-Bu2bXhq7A3/view?usp=sharing

Backend:

1. **Set Up Project Structure:**
    - Create a new directory for your project and set up a package.json file using npm init command.
    - Install necessary dependencies such as Express.js, Mongoose, and other required packages.

2. **Database Configuration:**
    - Set up a MongoDB database either locally or using a cloud-based MongoDB service like MongoDB Atlas or use locally with MongoDB compass.
    - Create a database and define the necessary collections for flights, users, bookings, and other relevant data.

3. **Create Express.js Server:**
    - Set up an Express.js server to handle HTTP requests and serve API endpoints.
    - Configure middleware such as body-parser for parsing request bodies and cors for handling cross-origin requests.

4. **Define API Routes:**
    - Create separate route files for different API functionalities such as flights, users, bookings, and authentication.
    - Define the necessary routes for listing flights, handling user registration and login, managing bookings, etc.
    - Implement route handlers using Express.js to handle requests and interact with the database.

5. **Implement Data Models:**
    - Define Mongoose schemas for the different data entities like flights, users, and bookings.
    - Create corresponding Mongoose models to interact with the MongoDB database.
    - Implement CRUD operations (Create, Read, Update, Delete) for each model to perform database operations.
    -

6. **User Authentication:**
    - Create routes and middleware for user registration, login, and logout.
    - Set up authentication middleware to protect routes that require user authentication.

7. **Handle new Flights and Bookings:**
    - Create routes and controllers to handle new flight listings, including fetching flight data from the database and sending it as a response.
    - Implement booking functionality by creating routes and controllers to handle booking requests, including validation and database updates.

8. **Admin Functionality:**
    - Implement routes and controllers specific to admin functionalities such as adding flights, managing user bookings, etc.
    - Add necessary authentication and authorization checks to ensure only authorized admins can access these routes.

9. **Error Handling:**
    - Implement error handling middleware to catch and handle any errors that occur during the API requests.
    - Return appropriate error responses with relevant error messages and HTTP status codes.

Schema usecase:

1. **User Schema:**
   - Schema: userSchema
   - Model: ‘User’
   - The User schema represents the user data and includes fields such as username, email, and password.
   - It is used to store user information for registration and authentication purposes.
   - The email field is marked as unique to ensure that each user has a unique email address**.**

2. **Flight Schema:**
   - Schema: flightSchema
   - Model: ‘Flight’
   - The Flight schema represents the hotel data and includes fields such as Flight Name, Flight Id, Origin, Destination, Price, seats, etc.,
   - It is used to store information about flights available for bookings.

3. **Booking Schema:**
   - Schema: BookingsSchema
   - Model: ‘Booking’
   - The Booking schema represents the booking data and includes fields such as userId, flight Name, flight Id, Passengers, Coach Class, Journey Date, etc.,
   - It is used to store information about the flight bookings made by users.
   - The user Id field is a reference to the user who made the booking.

6.Running the Application

- **Frontend:**npm start
- **Backend:**npm start

7.API Documentation

Code Explanation:

Server setup:

Let us import all the required tools/libraries and connect the database.

index.js

server > index.js > then() callback > app.post('/register') callback

```
import express from 'express';
import bodyParser from 'body-parser';
import mongoose from 'mongoose';
import cors from 'cors';
import bcrypt from 'bcrypt';
import { User, Booking, Flight } from './schemas.js';

const app = express();

app.use(express.json());
app.use(bodyParser.json({limit: "30mb", extended: true}))
app.use(bodyParser.urlencoded({limit: "30mb", extended: true}));
app.use(cors());

// mongoose setup

const PORT = 6001;
mongoose.connect('mongodb://localhost:27017/FlightBookingMERN', {
        useNewUrlParser: true,
        useUnifiedTopology: true,
    }
).then(()=>{


    // All the client-server activites

```

Schemas:

Now let us define the required schemas

schemas.js

server > schemas.js > ...

```
import mongoose from "mongoose";

const userSchema = new mongoose.Schema({
    username: { type: String, required: true },
    email: { type: String, required: true, unique: true },
    usertype: { type: String, required: true },
    password: { type: String, required: true },
    approval: {type: String, default: 'approved'}
});
const flightSchema = new mongoose.Schema({
    flightName: { type: String, required: true },
    flightId: { type: String, required: true },
    origin: { type: String, required: true },
    destination: { type: String, required: true },
    departureTime: { type: String, required: true },
    arrivalTime: { type: String, required: true },
    basePrice: { type: Number, required: true },
    totalSeats: { type: Number, required: true }
});
const bookingSchema = new mongoose.Schema({
    user: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
    flight: { type: mongoose.Schema.Types.ObjectId, ref: 'Flight', required: true },
    flightName: {type: String, required: true},
    flightId: {type: String},
    departure: {type: String},
    destination: {type: String},
    email: {type: String},
    mobile: {type: String},
    seats: {type: String},
    passengers: [{
        name: { type: String },
        age: { type: Number }
      }],
    totalPrice: { type: Number },
    bookingDate: { type: Date, default: Date.now },
    journeyDate: { type: Date },
    journeyTime: { type: String },
    seatClass: { type: String},
    bookingStatus: {type: String, default: "confirmed"}
  });

export const User =  mongoose.model('users', userSchema);
export const Flight = mongoose.model('Flight', flightSchema);
export const Booking = mongoose.model('Booking', bookingSchema);
```

User Authentication:

- **Backend**

  Now, here we define the functions to handle http requests from the client for authentication.

index.js

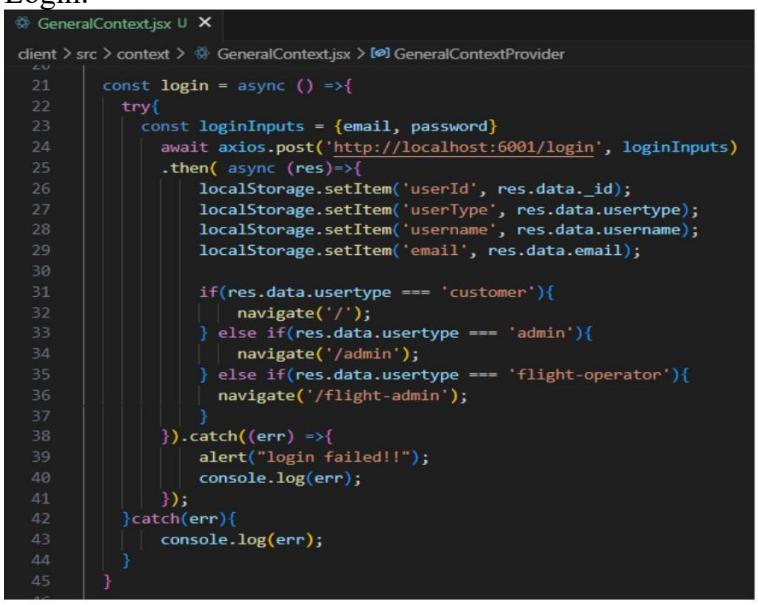
server > index.js > then() callback

```
26
27
28  app.post('/register', async (req, res) => {
29      const { username, email, usertype, password } = req.body;
30      let approval = 'approved';
31      try {
32          const existingUser = await User.findOne({ email });
33          if (existingUser) {
34              return res.status(400).json({ message: 'User already exists' });
35          }
36          if(usertype === 'flight-operator'){
37              approval = 'not-approved'
38          }
39          const hashedPassword = await bcrypt.hash(password, 10);
40          const newUser = new User({
41              username, email, usertype, password: hashedPassword, approval
42          });
43          const userCreated = await newUser.save();
44          return res.status(201).json(userCreated);
45      } catch (error) {
46        console.log(error);
47        return res.status(500).json({ message: 'Server Error' });
48      }
49  });
50
51  app.post('/login', async (req, res) => {
52      const { email, password } = req.body;
53      try {
54          const user = await User.findOne({ email });
55          if (!user) {
56              return res.status(401).json({ message: 'Invalid email or password' });
57          }
58          const isMatch = await bcrypt.compare(password, user.password);
59          if (!isMatch) {
60              return res.status(401).json({ message: 'Invalid email or password' });
61          } else{
62              return res.json(user);
63          }
64      } catch (error) {
65        console.log(error);
66        return res.status(500).json({ message: 'Server Error' });
67      }
68  });
```

8.Authentication

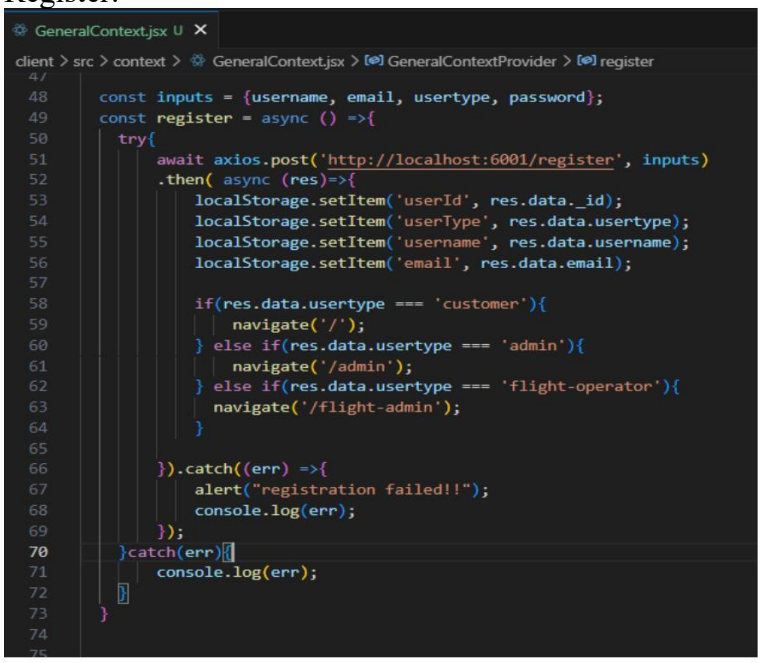
- **Frontend**

Login:



```
GeneralContext.jsx U
client > src > context > GeneralContext.jsx > GeneralContextProvider
21  const login = async () =>{
22    try{
23      const loginInputs = {email, password}
24        await axios.post('http://localhost:6001/login', loginInputs)
25        .then( async (res)=>{
26            localStorage.setItem('userId', res.data._id);
27            localStorage.setItem('userType', res.data.usertype);
28            localStorage.setItem('username', res.data.username);
29            localStorage.setItem('email', res.data.email);
30
31            if(res.data.usertype === 'customer'){
32              navigate('/');
33            } else if(res.data.usertype === 'admin'){
34              navigate('/admin');
35            } else if(res.data.usertype === 'flight-operator'){
36              navigate('/flight-admin');
37            }
38        }).catch((err) =>{
39            alert("login failed!!");
40            console.log(err);
41        });
42    }catch(err){
43        console.log(err);
44    }
45  }
```

Register:



```
GeneralContext.jsx U
client > src > context > GeneralContext.jsx > GeneralContextProvider > register
48  const inputs = {username, email, usertype, password};
49  const register = async () =>{
50    try{
51        await axios.post('http://localhost:6001/register', inputs)
52        .then( async (res)=>{
53            localStorage.setItem('userId', res.data._id);
54            localStorage.setItem('userType', res.data.usertype);
55            localStorage.setItem('username', res.data.username);
56            localStorage.setItem('email', res.data.email);
57
58            if(res.data.usertype === 'customer'){
59              navigate('/');
60            } else if(res.data.usertype === 'admin'){
61              navigate('/admin');
62            } else if(res.data.usertype === 'flight-operator'){
63              navigate('/flight-admin');
64            }
65
66        }).catch((err) =>{
67            alert("registration failed!!");
68            console.log(err);
69        });
70    }catch(err){
71        console.log(err);
72    }
73  }
74
```

Logout:

```
GeneralContext.jsx U
client > src > context > GeneralContext.jsx > GeneralContextProvider > login
73    const logout = async () =>{
74
75      localStorage.clear();
76      for (let key in localStorage) {
77        if (localStorage.hasOwnProperty(key)) {
78          localStorage.removeItem(key);
79        }
80      }
81
82      navigate('/');
83    }
84
```

9.User Interface

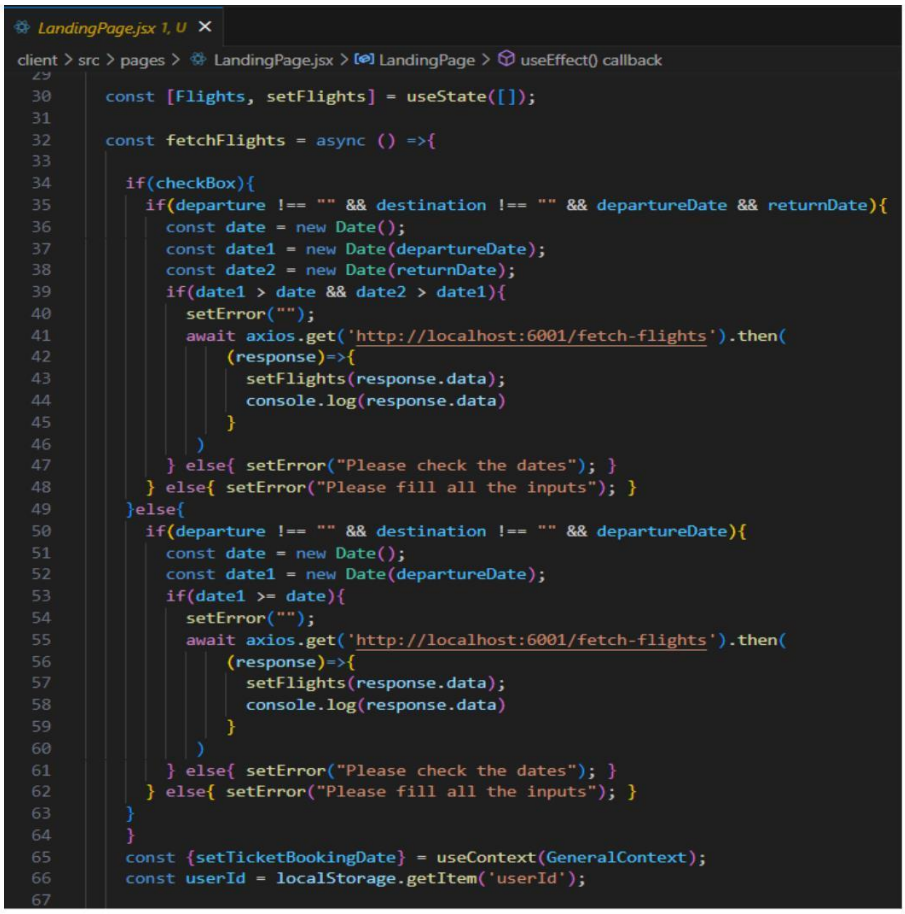
Flight Booking (User):

- **Frontend**

  In the frontend, we implemented all the booking code in a modal. Initially, we need to implement flight searching feature with inputs of Departure city, Destination, etc.,

  Flight Searching code:

  With the given inputs, we need to fetch the available flights. With each flight, we add a button to book the flight, which re-directs to the flight-Booking page.



On selecting the suitable flight, we then re-direct to the flight-booking page.

- **Backend**

In the backend, we fetch all the flights and then filter them in the client side.

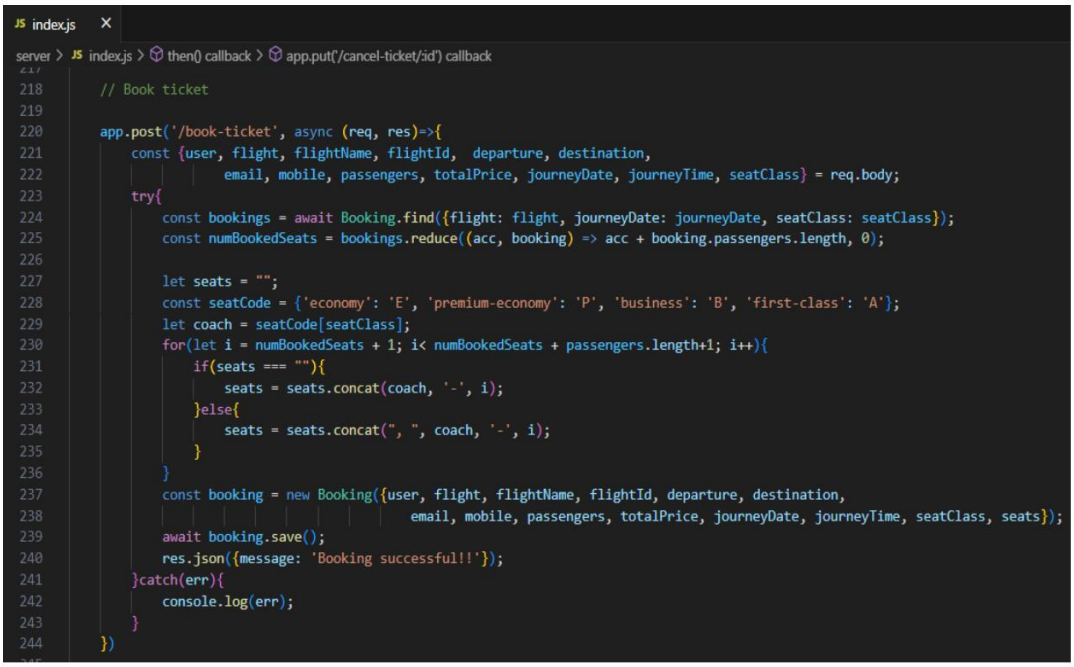
```
// fetch trains

app.get('/fetch-trains', async (req, res)=>{

    try{
        const trains = await Train.find();
        res.json(trains);

    }catch(err){
        console.log(err);
    }
})
```

Then, on confirmation, we book the flight ticket with the entered details.



```
// Book ticket

app.post('/book-ticket', async (req, res)=>{
    const {user, flight, flightName, flightId,  departure, destination,
            email, mobile, passengers, totalPrice, journeyDate, journeyTime, seatClass} = req.body;
    try{
        const bookings = await Booking.find({flight: flight, journeyDate: journeyDate, seatClass: seatClass});
        const numBookedSeats = bookings.reduce((acc, booking) => acc + booking.passengers.length, 0);

        let seats = "";
        const seatCode = {'economy': 'E', 'premium-economy': 'P', 'business': 'B', 'first-class': 'A'};
        let coach = seatCode[seatClass];
        for(let i = numBookedSeats + 1; i< numBookedSeats + passengers.length+1; i++){
            if(seats === ""){
                seats = seats.concat(coach, '-', i);
            }else{
                seats = seats.concat(", ", coach, '-', i);
            }
        }
        const booking = new Booking({user, flight, flightName, flightId, departure, destination,
                                     email, mobile, passengers, totalPrice, journeyDate, journeyTime, seatClass, seats});
        await booking.save();
        res.json({message: 'Booking successful!!'});
    }catch(err){
        console.log(err);
    }
})
```

10.Testing

Fetching user bookings:

- **Frontend**

In the bookings page, along with displaying the past bookings, we will also provide an option to cancel that booking.

```
Bookings.jsx U
client > src > pages > Bookings.jsx > Bookings
  const [bookings, setBookings] = useState([]);

  const userId = localStorage.getItem('userId');

  useEffect(()=>{
    fetchBookings();
  }, [])

  const fetchBookings = async () =>{
    await axios.get('http://localhost:6001/fetch-bookings').then(
      (response)=>{
        setBookings(response.data);
      }
    )
  }
  const cancelTicket = async (id) =>{
    await axios.put(`http://localhost:6001/cancel-ticket/${id}`).then(
      (response)=>{
        alert("Ticket cancelled!!");
        fetchBookings();
      }
    )
  }
```

- **Backend**

  In the backend, we fetch all the bookings and then filter for the user. Otherwise, we can fetch bookings only for the user.

```
index.js
server > index.js > then() callback
app.get('/fetch-bookings', async (req, res)=>{

    try{
        const bookings = await Booking.find();
        res.json(bookings);

    }catch(err){
        console.log(err);
    }
})
```

Then we define a function to delete the booking on cancelling it on client side.

```
index.js
server > index.js > ...
app.put('/cancel-ticket/:id', async (req, res)=>{
    const id = await req.params.id;
    try{
        const booking = await Booking.findById(req.params.id);
        booking.bookingStatus = 'cancelled';
        await booking.save();
        res.json({message: "booking cancelled"});

    }catch(err){
        console.log(err);
    }
})
```

Add new flight:

Now, in the admin dashboard, we provide a functionality to add new flight.

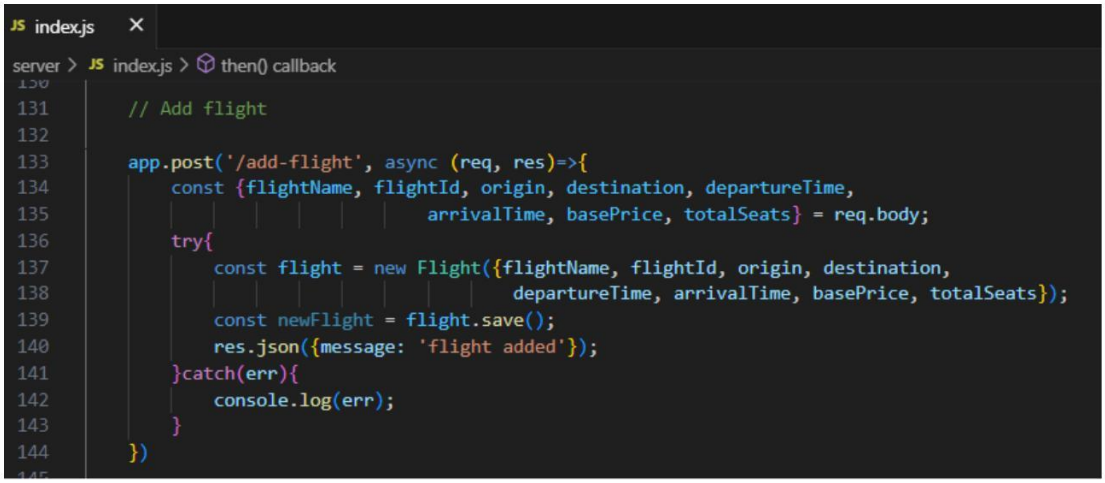
- **Frontend**

  We create a html form with required inputs for the new flight and then send an http request to the server to add it to database.

```
NewFlight.jsx U
client > src > pages > NewFlight.jsx > NewFlight
30
31    const [flightName, setFlightName] = useState(localStorage.getItem('username'));
32
33    const [flightId, setFlightId] = useState('');
34    const [origin, setOrigin] = useState('');
35    const [destination, setDestination] = useState('');
36    const [startTime, setStartTime] = useState('');
37    const [arrivalTime, setArrivalTime] = useState('');
38    const [totalSeats, setTotalSeats] = useState(0);
39    const [basePrice, setBasePrice] = useState(0);
40
41    const handleSubmit = async () =>{
42
43      const inputs = {flightName, flightId, origin, destination,
44                      departureTime: startTime, arrivalTime, basePrice, totalSeats};
45
46      await axios.post('http://localhost:6001/add-Flight', inputs).then(
47        async (response)=>{
48          alert('Flight added successfully!!');
49          setFlightName('');
50          setFlightId('');
51          setOrigin('');
52          setStartTime('');
53          setArrivalTime('');
54          setDestination('');
55          setBasePrice(0);
56          setTotalSeats(0);
57        }
58      )
59    }
60
```

- **Backend**

  In the backend, on receiving the request from the client, we then add the request body to the flight schema.



```
index.js
server > index.js > then() callback
131    // Add flight
132
133    app.post('/add-flight', async (req, res)=>{
134        const {flightName, flightId, origin, destination, departureTime,
135                                arrivalTime, basePrice, totalSeats} = req.body;
136        try{
137            const flight = new Flight({flightName, flightId, origin, destination,
138                                departureTime, arrivalTime, basePrice, totalSeats});
139            const newFlight = flight.save();
140            res.json({message: 'flight added'});
141        }catch(err){
142            console.log(err);
143        }
144    })
```

Update Flight:

Here, in the admin dashboard, we will update the flight details in case if we want to make any edits to it

- **Frontend:**

EditFlight.jsx 1, U

client > src > pages > EditFlight.jsx > EditFlight

```
61    const handleSubmit = async () =>{
62
63      const inputs = {_id: id,flightName, flightId, origin, destination,
64        departureTime: startTime, arrivalTime, basePrice, totalSeats};
65
66      await axios.put('http://localhost:6001/update-flight', inputs).then(
67        async (response)=>{
68          alert('Flight updated successfully!!');
69          setFlightName('');
70          setFlightId('');
71          setOrigin('');
72          setStartTime('');
73          setArrivalTime('');
74          setDestination('');
75          setBasePrice(0);
76          setTotalSeats(0);
77        }
78      )
79    }
```

- **Backend:**

index.js

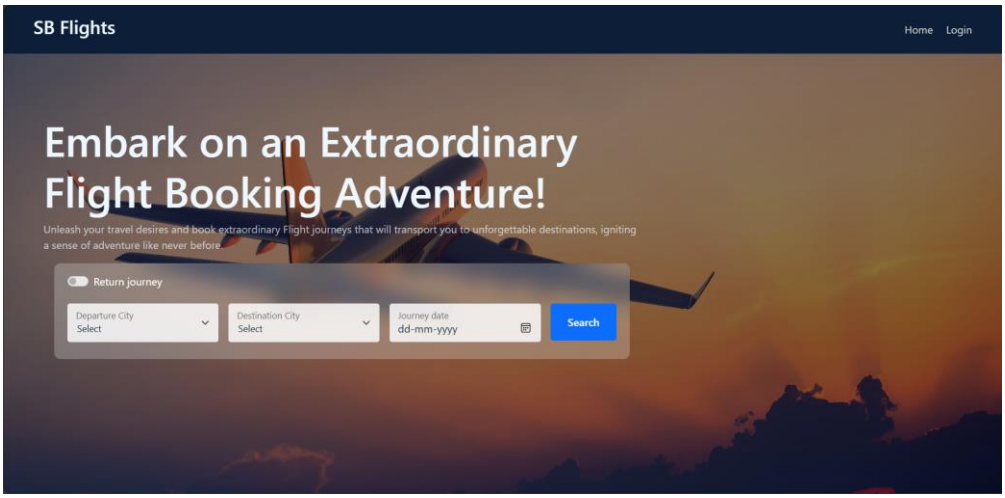
server > index.js > then() callback

```
147   // update flight
148
149   app.put('/update-flight', async (req, res)=>{
150       const {_id, flightName, flightId, origin, destination,
151                   departureTime, arrivalTime, basePrice, totalSeats} = req.body;
152       try{
153           const flight = await Flight.findById(_id)
154
155           flight.flightName = flightName;
156           flight.flightId = flightId;
157           flight.origin = origin;
158           flight.destination = destination;
159           flight.departureTime = departureTime;
160           flight.arrivalTime = arrivalTime;
161           flight.basePrice = basePrice;
162           flight.totalSeats = totalSeats;
163
164           const newFlight = flight.save();
165           res.json({message: 'flight updated'});
166
167       }catch(err){
168           console.log(err);
169       }
170   })
171
```

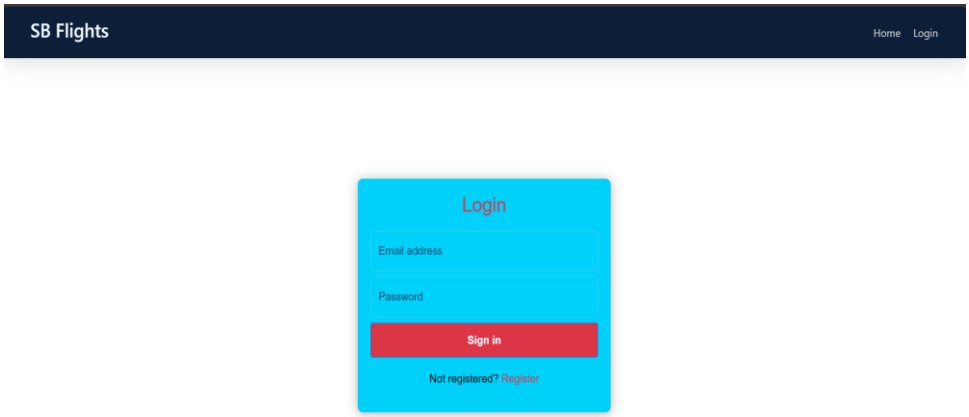
Along with this, implement additional features to view all flights, bookings, and users in admin dashboard.

11.Screenshots or Demo

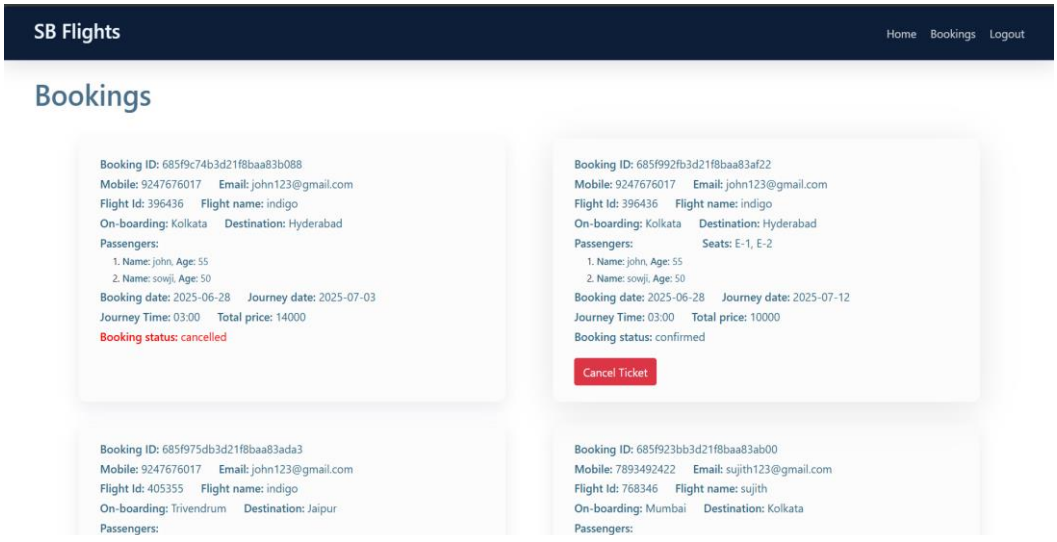
- **Landing page**



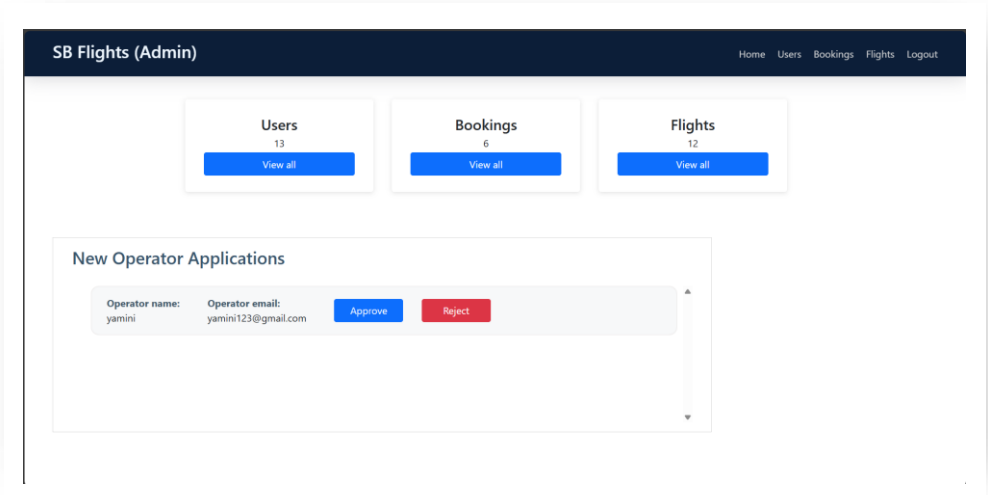
- **Authentication**



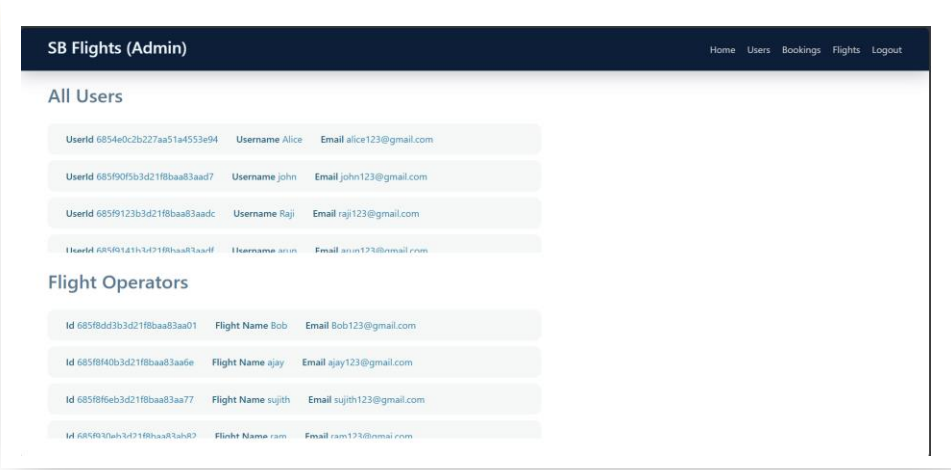
- **User bookings**



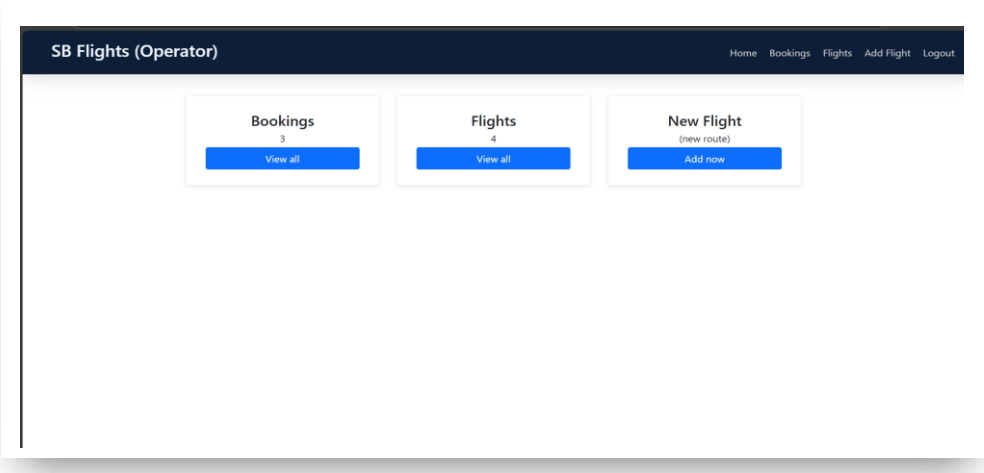
- **Admin Dashboard**



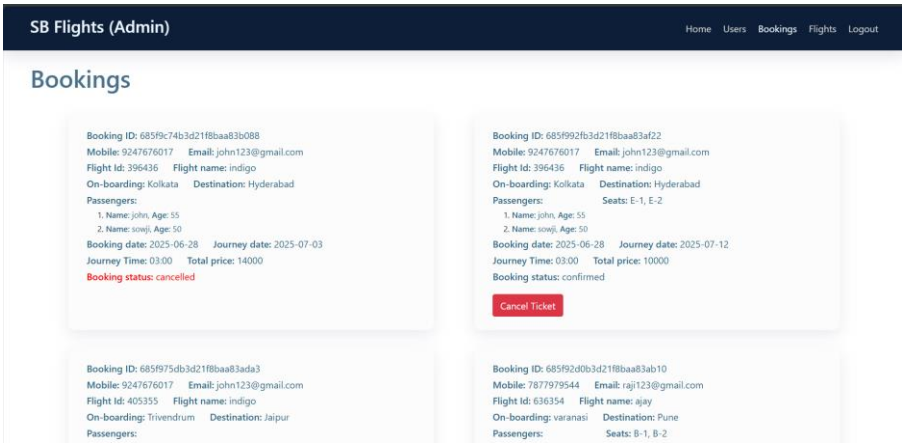
- **All users**



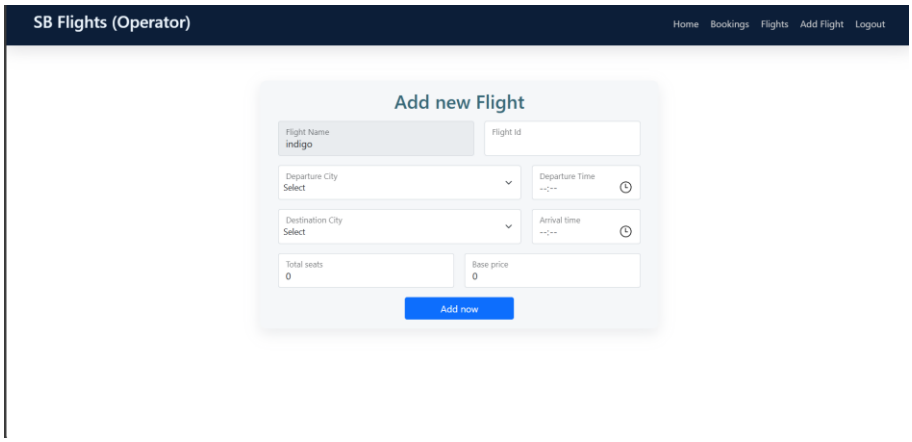
- **Flight Operator**



- **All Bookings**



- **New Flight**



For any further doubts or help, please consider the GitHub repo,

https://github.com/L-Gayathri/Flight_Finder

The demo of the app is available at:
https://drive.google.com/file/d/1MzaPO59qKIVO0g4XAbonCJnNJGzg-ymg/view?usp=drive_link

12.Known Issues

User-Side Issues (Travelers)

1. Inaccurate or Incomplete Flight Information

- Outdated schedules, missing layover details, or hidden fees can frustrate users.
- Lack of transparency around baggage allowance or seat availability.

2. Complex User Interfaces

- Confusing navigation, too many input fields, or unclear steps can lead to booking abandonment.
- Poor mobile optimization affects usability.

3. Limited Filtering Options

- Users often struggle to sort flights by preferred criteria (e.g. time, price, duration, stops, airlines).
- Inadequate customization leads to longer search time.

Admin-Side Issues (Flight Operators / Airlines)

1. Manual Management Overload

- Without automation, managing a high volume of bookings becomes error-prone and time-consuming.

2. Lack of Real-Time Sync with Inventory

- Seat availability may not update quickly, leading to overbooking or false availability.
- Changes to flight schedules might not reflect instantly on the user side.

3. Limited Reporting and Analytics

- No easy access to booking trends, passenger demographics, or cancellation patterns.

4. Security Risks

- Inadequate protection of user data (e.g. unencrypted PII or payment data).
- Admin portals vulnerable to unauthorized access.

13.Future Enhancements

User Experience Improvements

- **Advanced Flight Filtering & Sorting:**
  Allow users to filter flights by price, duration, layovers, airlines, time of day, and class.
- **Seat Selection Feature:**
  Enable users to choose specific seats during booking, with real-time seat map updates.
- **Multi-Language & Multi-Currency Support:**
  Add internationalization features to support global users with localized content and pricing.
- **Passenger Profile Management:**
  Allow users to create accounts and save personal details, travel preferences, and frequent flyer info for faster booking.

Flight Search & Real-Time Data

- **Live Flight Pricing and Availability:**
  Integrate with real-time airline APIs (e.g., Amadeus, Sabre, or Skyscanner) for accurate price and seat data.
- **Dynamic Pricing Engine:**
  Implement an algorithm to adjust pricing based on demand, availability, or booking window.
- **Live Flight Tracking:**
  Allow users to track live flight status (delays, gate changes, weather) via third-party API.