Full Stack Development with MERN

Project Documentation

1. INTRODUCTION:

❖ SB Flights is a modern, full-stack web application designed to transform and simplify the way users book flight tickets. With its intuitive interface and streamlined features, SB Flights offers a smarter, faster, and more transparent flight booking experience. Whether you're a frequent flier or an occasional traveler, this platform ensures that your journey planning is seamless and stress-free.

❖ PROJECT TITLE:

FlightFinder: Navigating Your Air Travel Options

❖ TEAM MEMBERS:

Team Leader: Merugu Naveena - Frontend development

Team member: Shaik Mohammed Ashik - Backend development

Team member: Nakka Jaya Vishnu Vardhan Rao- Testing & deployment

Team member: Marella Hemanthi- Final documentation

2. PROJECT OVERVIEW:

SB Flights is a comprehensive flight booking system built using the MERN (MongoDB, Express.js, React.js, Node.js) stack. It enables users to search and book flights based on various preferences while providing administrators with powerful tools to manage flights and bookings.

The application offers two primary interfaces:

- A **user portal** for travelers to explore flights, make reservations, and manage their bookings.

- An **admin dashboard** for flight operators and administrators to manage flight listings, view booking details, and maintain system integrity with secure logins.

SB Flights prioritizes convenience, user control, and efficient booking processes by integrating frontend and backend components with real-time data handling.

❖ PURPOSE:

The main objectives of the SB Flights application are:

- To simplify the flight booking process for users through a responsive and easy-to-use interface.
- To provide clear, accessible, and detailed flight information so users can make informed decisions.
- To eliminate the traditional complexities of booking systems like long queues and delayed confirmations.
- To equip administrators with tools to efficiently manage flight data, user bookings, and system operations through a secure and intuitive dashboard.
- To support scalability and modular design for potential future enhancements such as payments, notifications, or third-party integrations.

❖ FEATURES:

For Users:

- **Flight Search and Listings**: Users can search for flights by selecting departure and destination cities, travel dates, and number of passengers.

- **Detailed Flight Information**: Every flight includes key details like departure and arrival times, class options, prices, and available amenities.

- **Quick Booking Process**: Users can easily book flights by filling in simple information like name, age, and travel preferences.

- **Instant Confirmation**: Upon booking, users receive immediate confirmation with all ticket details.

- **Booking Management Page**: A dedicated section to view, manage, and cancel current and past bookings.

For Administrators:

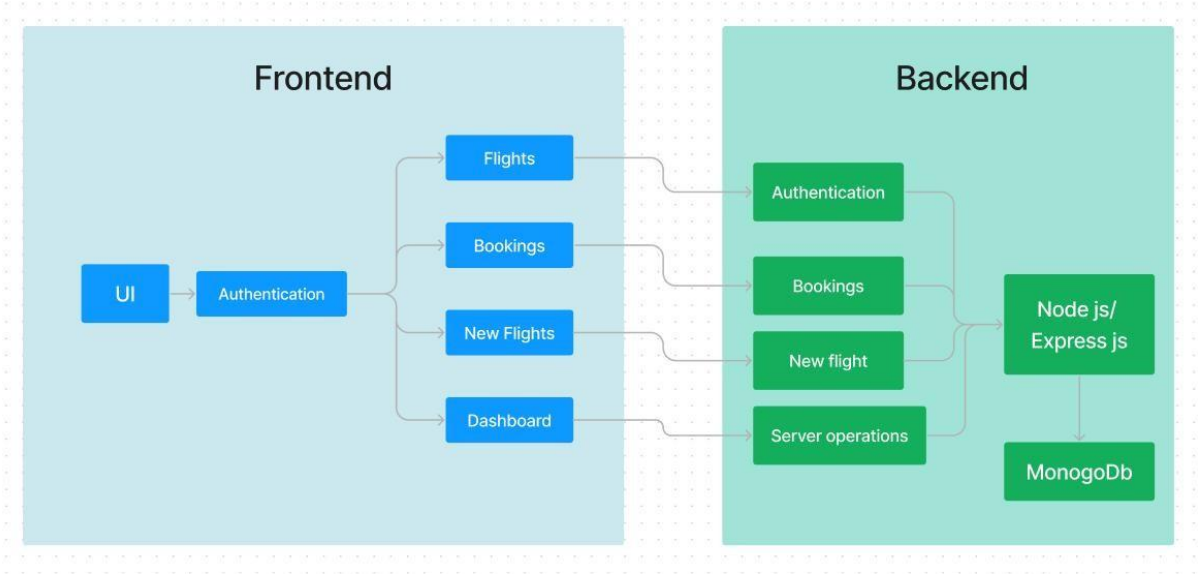
- **Admin Dashboard**: A centralized platform for managing flights, viewing bookings, and monitoring user activity.

- **Flight Management**: Add, update, or remove flights from the system.

- **Booking Oversight**: View user bookings, track flight occupancy, and ensure smooth operations.

- **Secure Admin Authentication**: Separate login and registration modules for admins and flight operators to maintain data security and role-based access.

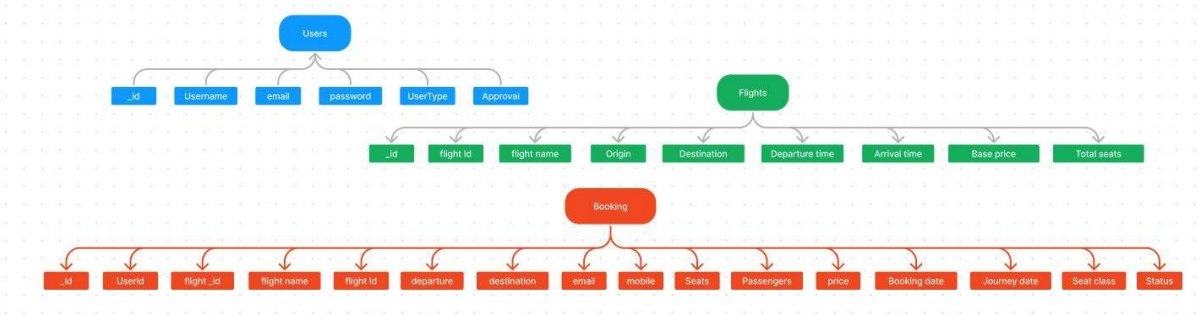
3. ARCHITECTURE:



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.

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:

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.

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.

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.

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.

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 userfriendly 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:

❖ PRE-REQUISTIC:

To develop a full-stack flight booking app using React JS, Node.js, and MongoDB, there are several prerequisites 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 development machine. 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 or use 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 install express**

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 to connect 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 products listings, booking forms, and user interfaces for the admin dashboard.

Version Control: Use Git for version control, enabling collaboration and tracking changes throughout the development 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 your preferences, 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/Ashik1029/FlightFinder-Navigating-Your-Air-Travel-

Options**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 port configuration in the .env file if needed.

Access the App:

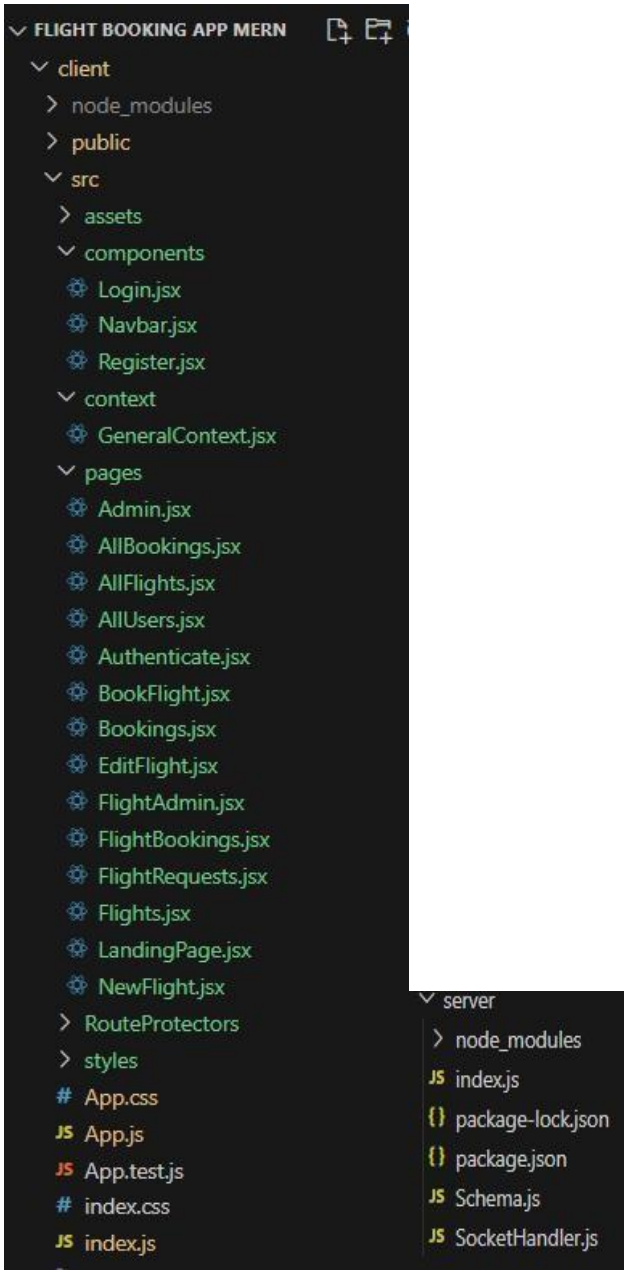
• Open your web browser and navigate to http://localhost:3000.
• You should see the flight booking app's homepage, indicating that the installation and setup were successful.

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

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.



6. RUNNING THE APPLICATION:

❖ Backend Development

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. UserAuthentication:

- 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:

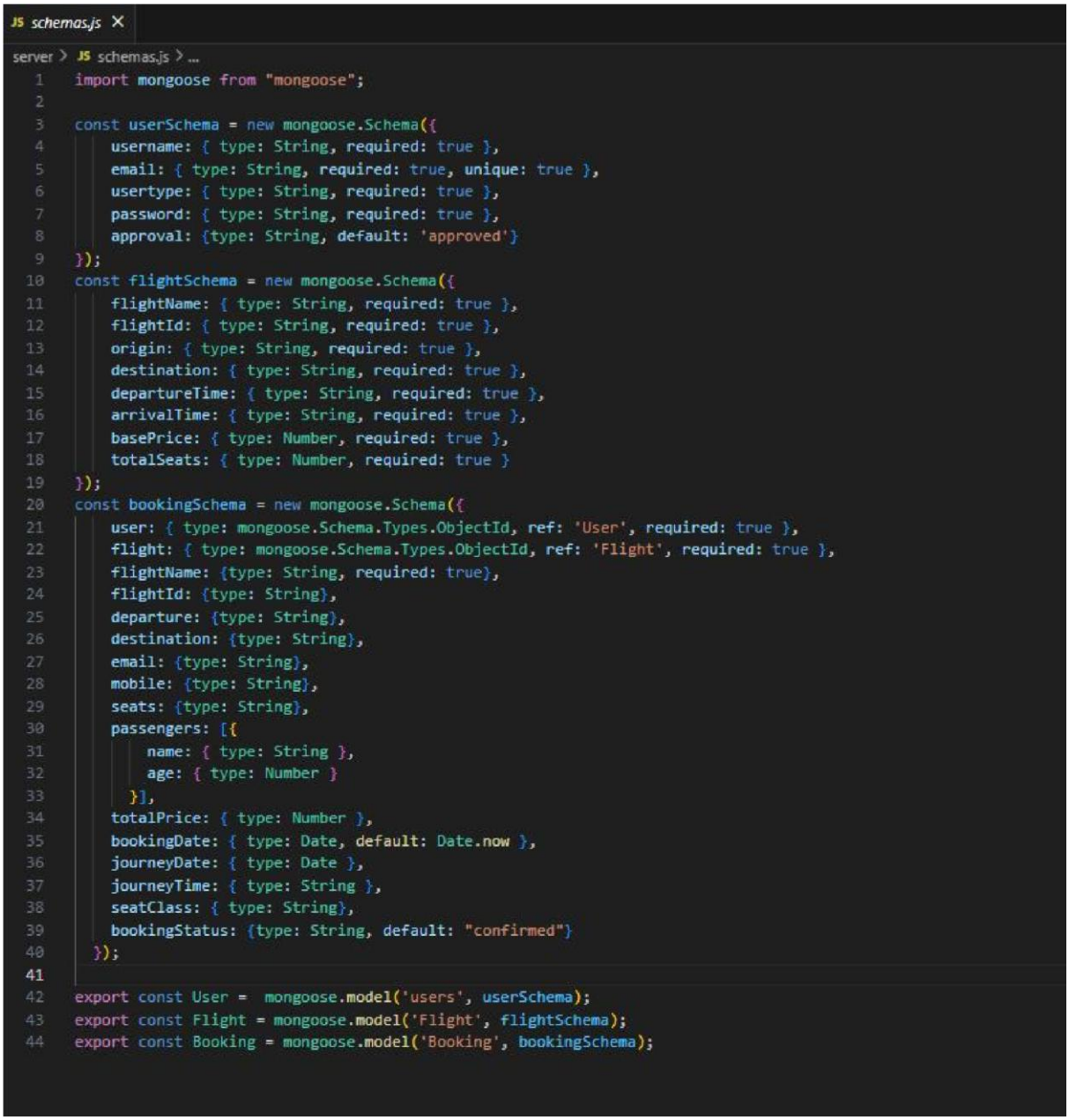
- Schema: flightSchema
- Model: ‘Flight’
- The Flight schema represents the hotel data and includes fields such as Flight Name,

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.



```
JS schemas.js
server > JS 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);
```

❖ Frontend development

1. Setting the Stage

The **Flight Finder** frontend is powered by React.js. To kick off development:

We'll create the initial React application setup using `create-react-app`.

Install essential libraries such as Axios for API calls, React Router for navigation, and Tailwind CSS or Bootstrap for styling.

Organize components, pages, and utilities into a clean project structure for efficient development.
This solid foundation ensures a streamlined workflow as we begin building the intuitive and responsive Flight Finder interface.

---

2. Crafting the User Experience

Next, we shift focus to the **user interface (UI)** — the heart of user interaction:

Design reusable components such as search bars, filter dropdowns, flight cards,andbooking modals.

Define a consistent layout and styling to ensure a clean, modern, and responsive interface across devices.

Implement navigation elements for easy movement between pages like Home, Search Results, and Booking History.

These elements collectively offer a smooth and user-friendly experience for travelers exploring and booking flights.

---

3. Bridging the Gap

In the final phase, we connect the frontend with the backend services:

Integrate API endpoints for fetching available flights, submitting bookings, and retrieving user history.

Implement state management and data binding to ensure real-time updates based on user interactions.

This integration completes the frontend development of the **Flight Finder** platform, delivering a fully interactive and dynamic flight booking experience.

7. API DOCUMENTATION:

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

```
1   import express from 'express';
2   import bodyParser from 'body-parser';
3   import mongoose from 'mongoose';
4   import cors from 'cors';
5   import bcrypt from 'bcrypt';
6   import { User, Booking, Flight } from './schemas.js';
7
8   const app = express();
9
10  app.use(express.json());
11  app.use(bodyParser.json({limit: "30mb", extended: true}))
12  app.use(bodyParser.urlencoded({limit: "30mb", extended: true}));
13  app.use(cors());
14
15  // mongoose setup
16
17  const PORT = 6001;
18  mongoose.connect('mongodb://localhost:27017/FlightBookingMERN', {
19          useNewUrlParser: true,
20          useUnifiedTopology: true,
21      }
22  ).then(()=>{
23
24
25      // All the client-server activites
26
```

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

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.

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

index.js

server > index.js > then() callback > app.put('/cancel-ticket/:id') callback

```
// 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);
    }
})
```

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

```
const [flightName, setFlightName] = useState(localStorage.getItem('username'));

const [flightId, setFlightId] = useState('');
const [origin, setOrigin] = useState('');
const [destination, setDestination] = useState('');
const [startTime, setStartTime] = useState('');
const [arrivalTime, setArrivalTime] = useState('');
const [totalSeats, setTotalSeats] = useState(0);
const [basePrice, setBasePrice] = useState(0);

const handleSubmit = async () =>{

  const inputs = {flightName, flightId, origin, destination,
                  departureTime: startTime, arrivalTime, basePrice, totalSeats};

  await axios.post('http://localhost:6001/add-Flight', inputs).then(
    async (response)=>{
      alert('Flight added successfully!!');
      setFlightName('');
      setFlightId('');
      setOrigin('');
      setStartTime('');
      setArrivalTime('');
      setDestination('');
      setBasePrice(0);
      setTotalSeats(0);
    }
  )
}
```

- Backend

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

```
// Add flight

app.post('/add-flight', async (req, res)=>{
    const {flightName, flightId, origin, destination, departureTime,
                                arrivalTime, basePrice, totalSeats} = req.body;
    try{
        const flight = new Flight({flightName, flightId, origin, destination,
                                departureTime, arrivalTime, basePrice, totalSeats});
        const newFlight = flight.save();
        res.json({message: 'flight added'});
    }catch(err){
        console.log(err);
    }
})
```

Update Flight:

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

Fetching user bookings:

• Frontend

```
const handleSubmit = async () =>{

  const inputs = {_id: id,flightName, flightId, origin, destination,
    departureTime: startTime, arrivalTime, basePrice, totalSeats};

  await axios.put('http://localhost:6001/update-flight', inputs).then(
    async (response)=>{
      alert('Flight updated successfully!!');
      setFlightName('');
      setFlightId('');
      setOrigin('');
      setStartTime('');
      setArrivalTime('');
      setDestination('');
      setBasePrice(0);
      setTotalSeats(0);
    }
  )
}
```

Backend:

index.js X

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

8. Authentication:

- backend

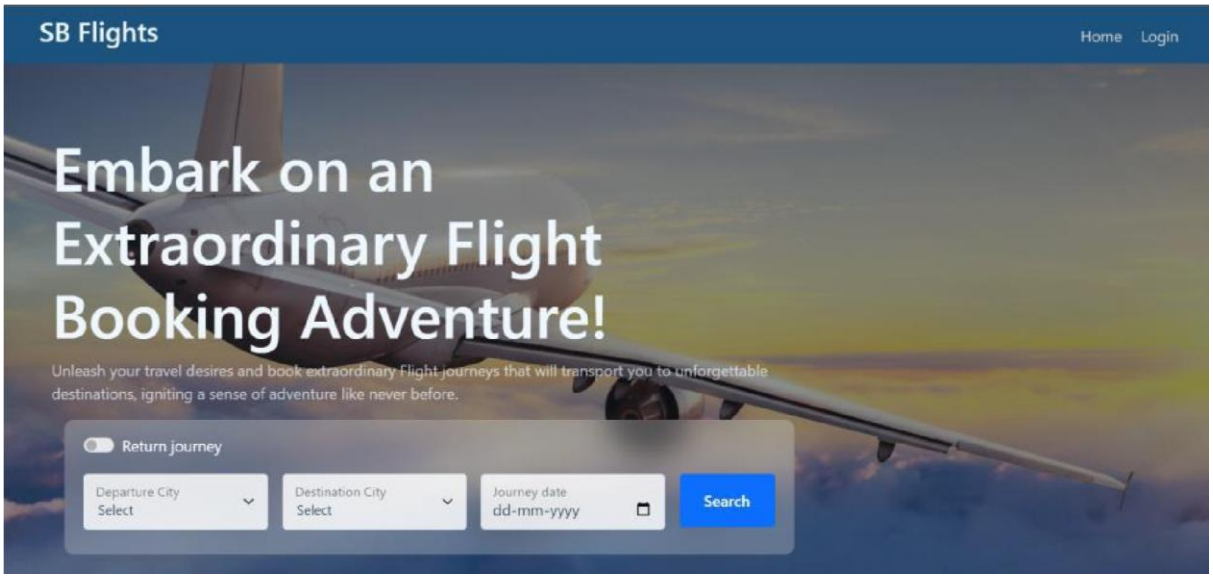
Now, here we define the functions to handle http requests from the client for authentication

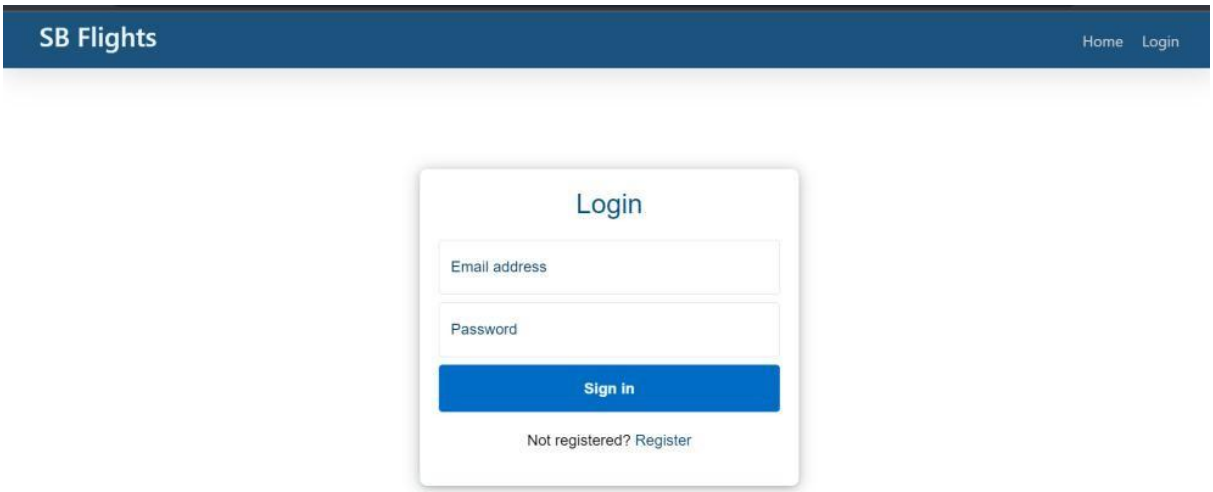
```
index.js
server > index.js > then() callback

app.post('/register', async (req, res) => {
    const { username, email, usertype, password } = req.body;
    let approval = 'approved';
    try {
        const existingUser = await User.findOne({ email });
        if (existingUser) {
            return res.status(400).json({ message: 'User already exists' });
        }
        if(usertype === 'flight-operator'){
            approval = 'not-approved'
        }
        const hashedPassword = await bcrypt.hash(password, 10);
        const newUser = new User({
            username, email, usertype, password: hashedPassword, approval
        });
        const userCreated = await newUser.save();
        return res.status(201).json(userCreated);
    } catch (error) {
      console.log(error);
      return res.status(500).json({ message: 'Server Error' });
    }
});

app.post('/login', async (req, res) => {
    const { email, password } = req.body;
    try {
        const user = await User.findOne({ email });
        if (!user) {
            return res.status(401).json({ message: 'Invalid email or password' });
        }
        const isMatch = await bcrypt.compare(password, user.password);
        if (!isMatch) {
            return res.status(401).json({ message: 'Invalid email or password' });
        } else{
            return res.json(user);
        }
    } catch (error) {
      console.log(error);
      return res.status(500).json({ message: 'Server Error' });
    }
});
```

9. USER INTERFACE:

- Providing screenshots or GIFs showcasing different UI features.

Landing page:





10.TESTING:

• Describe the testing strategy and tools used.

Testing Strategy:

To ensure the reliability and quality of the application, a combination of **unit testing**, **integration testing**, and **manual testing** was used:

1. **Unit Testing:**
    - Focused on testing individual functions, models, and API controllers (e.g., user authentication, profile updates).
    - Ensured each component works as expected in isolation.
2. **Integration Testing:**

- Verified end-to-end functionality of APIs (e.g., user registration to project hiring).
- Checked interactions between components like database, middleware, and routes.

3. **Manual Testing (UI & UX):**

- Tested all core user flows (e.g., login, posting projects, hiring freelancers).
- Used for quick verification during development and identifying UI/UX bugs.

✓ **Installation of required tools:**

1. Open the frontend folder to install necessary tools

   For frontend, we use:

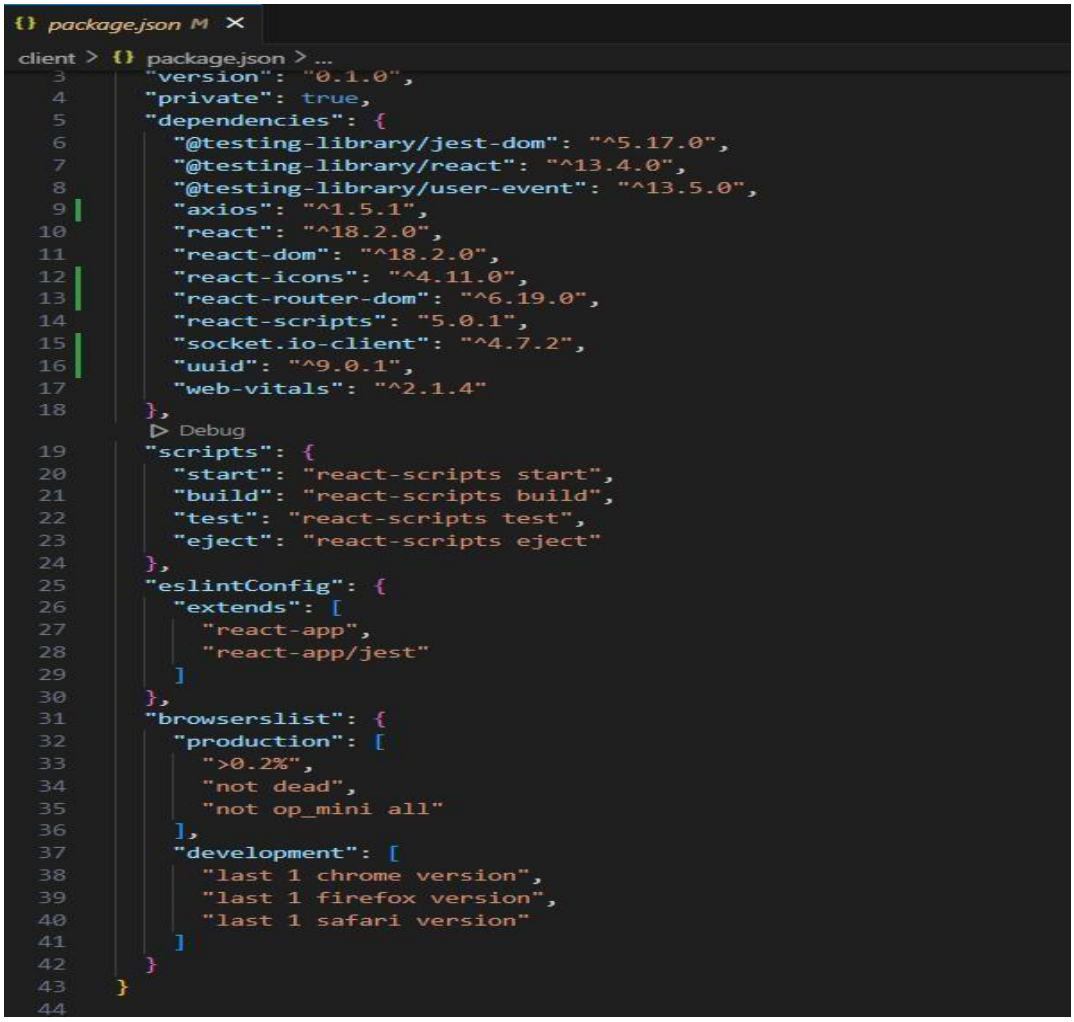
   - React
   - Bootstrap
   - Material UI
   - Axios
   - react-bootstrap

2. Open the backend folder to install necessary tools

   For backend, we use:

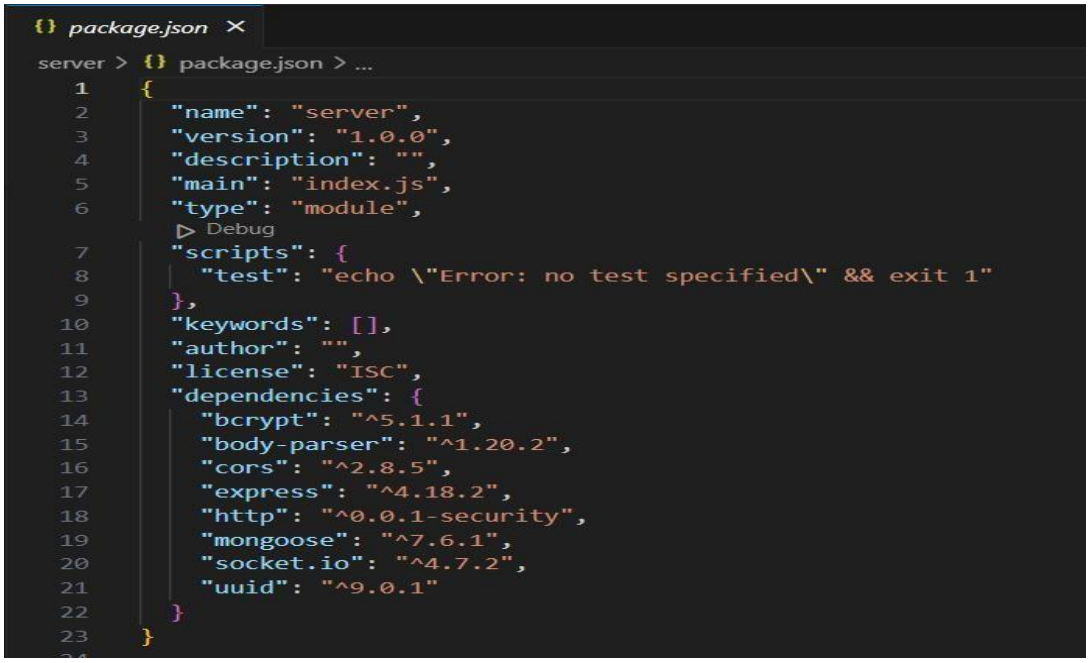
   - Express Js
   - Node JS
   - MongoDB
   - Mongoose
   - Cors
   - Bcrypt

After the installation of all the libraries, the package.json files for the frontend looks like the one mentioned below.



```
{} package.json M
client > {} package.json > ...
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.17.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "axios": "^1.5.1",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-icons": "^4.11.0",
    "react-router-dom": "^6.19.0",
    "react-scripts": "5.0.1",
    "socket.io-client": "^4.7.2",
    "uuid": "^9.0.1",
    "web-vitals": "^2.1.4"
  },
  ▷ Debug
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

After the installation of all the libraries, the package.json files for the backend looks like the one mentioned below.



```
{} package.json
server > {} package.json > ...
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcrypt": "^5.1.1",
    "body-parser": "^1.20.2",
    "cors": "^2.8.5",
    "express": "^4.18.2",
    "http": "^0.0.1-security",
    "mongoose": "^7.6.1",
    "socket.io": "^4.7.2",
    "uuid": "^9.0.1"
  }
}
```

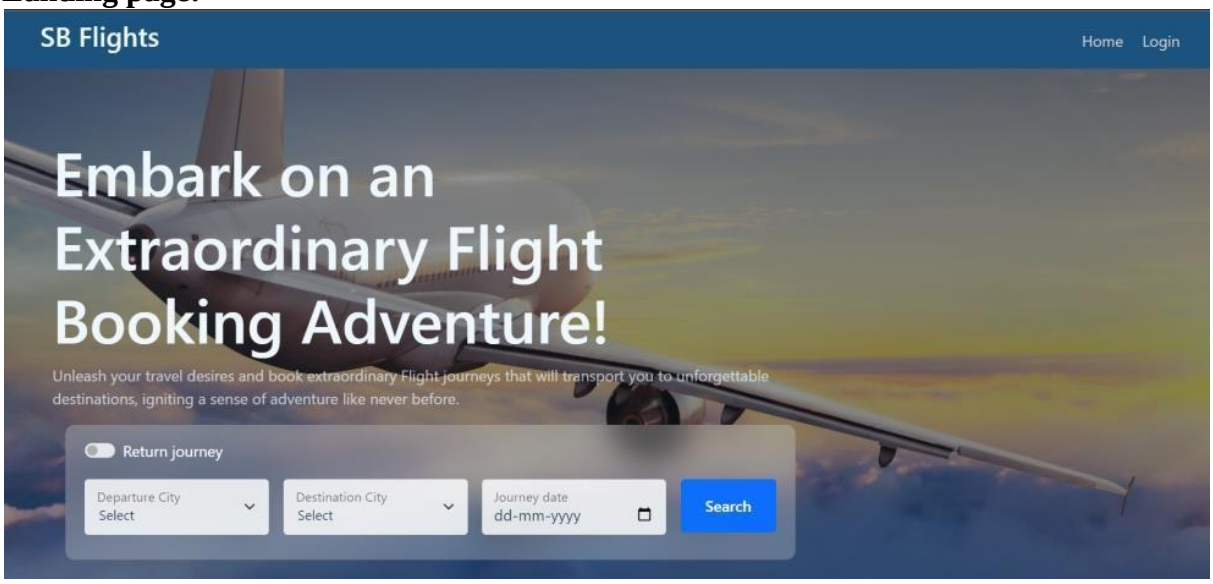
11. SCREENSHOTS OR DEMO:

• Providing screenshots or a link to a demo to showcase the application.

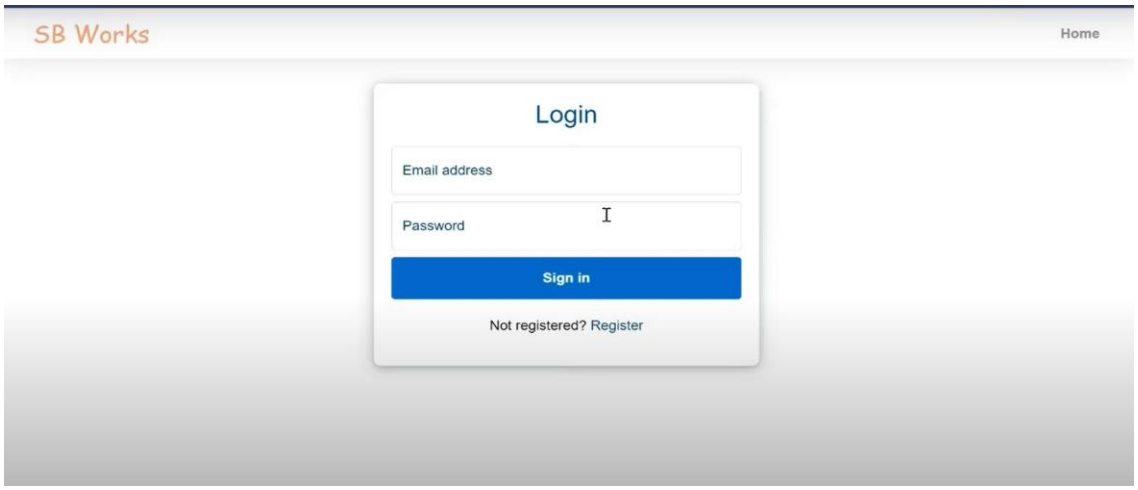
PROJECT IMPLEMENTATION;

On completing the development part, we then run the application one last time to verify all the functionalities and look for any bugs in it. The user interface of the application looks a bit like the images provided below.

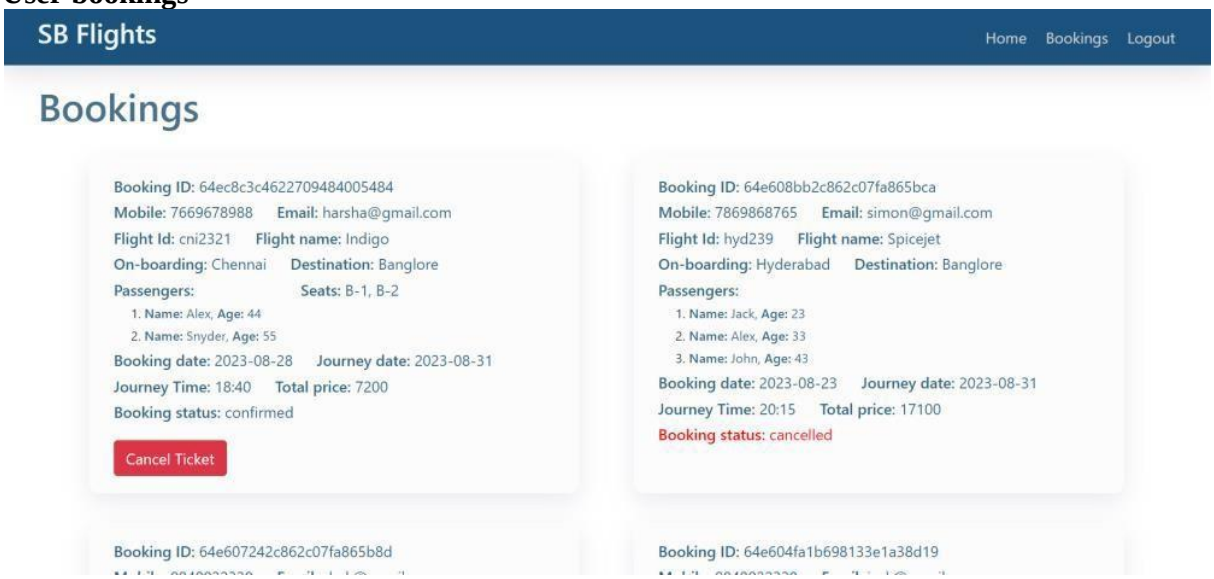
Landing page:



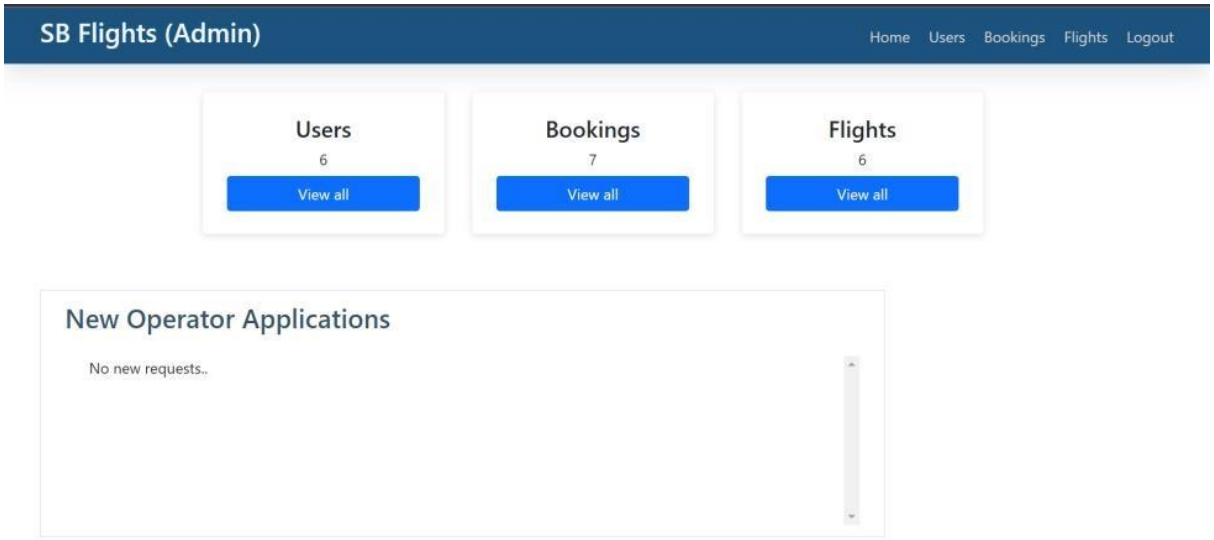
Authentication:



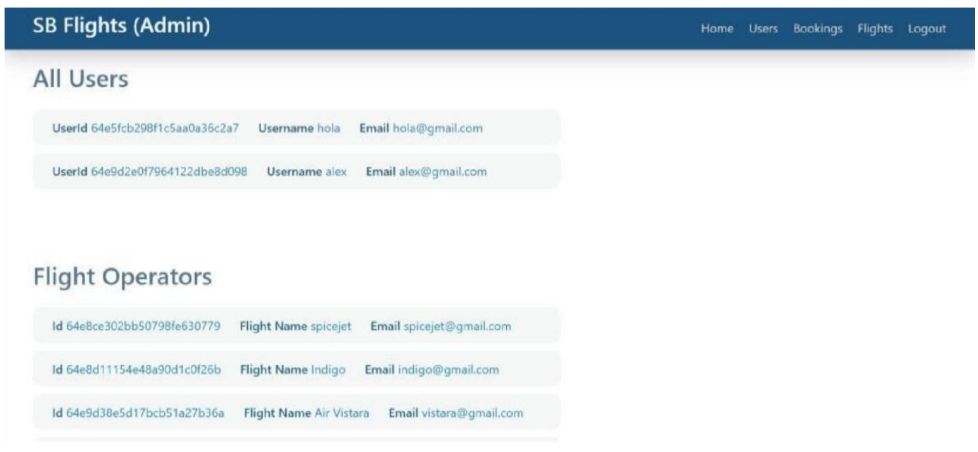
User bookings



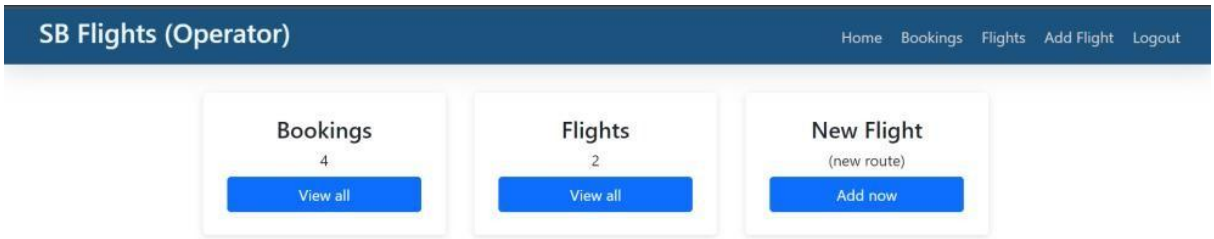
• Admin Dashboard



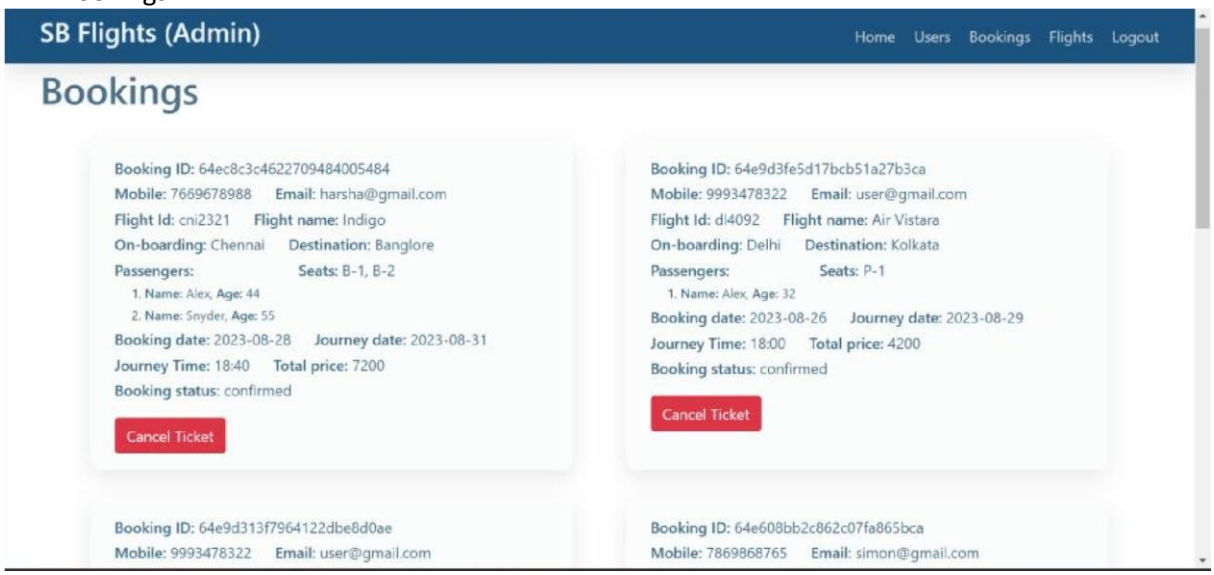
• All users



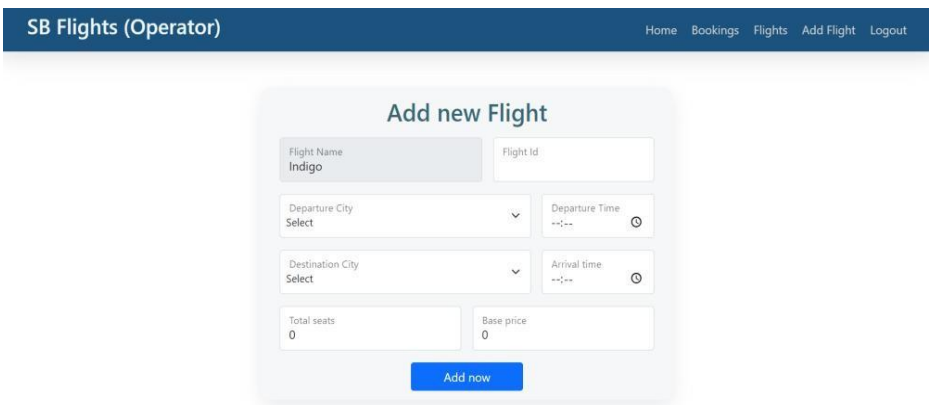
• Flight Operator



• All Bookings



• New Flight



12. KNOWN ISSUES:

1. Data Accuracy & Real-Time Updates

Problem: Flight schedules, delays, and pricing are highly dynamic.

Impact: Users may see outdated or incorrect results.

Cause: Static or infrequently updated data sources; lack of real-time API integration.

Solution: Integrate with reliable real-time APIs (e.g., Amadeus, Skyscanner, AviationStack).

---

2. API Limitations and Rate Limits

Problem: Free or low-tier APIs have request limits.

Impact: App may stop working or return partial data.

Solution: Use caching, pagination, and upgrade plans as needed.

---

3. Incomplete or Weak Search Filters

Problem: Users can't refine searches well (by stops, airline, time, price).

Impact: Poor user experience; overwhelming result sets.

Solution: Implement advanced filters with intuitive UI.

---

4. Slow Load Times for Results

Problem: Flight data fetching is slow, especially with multiple API calls.

Impact: High bounce rate and user frustration.

Solution: Use async requests, loading spinners, server-side caching, and pagination.

---

5. No or Insecure Payment Handling (if booking is enabled)

Problem: Payment gateway may not be integrated or secured properly.

Impact: Users can't book, or risk of data leaks.

Solution: Use secure, PCI-DSS compliant payment systems like Stripe or Razorpay.

---

6. Testing & Error Handling Gaps

Problem: Unexpected failures during booking, cancellations, or search.

Impact: Crashes, broken features.

Solution: Add comprehensive validation and error boundaries; test with edge cases.

---

7. Authentication and Data Privacy

Problem: Weak or no login/session security.

Impact: Unauthorized access, data leaks.

Solution: Use JWT, encrypted sessions, and HTTPS.

---

8. Poor Mobile Responsiveness

Problem: Layout breaks on mobile.

Impact: Bad UX for mobile users.

Solution: Use responsive design (e.g., Tailwind CSS breakpoints or media queries).

---

9. No Location-Based Suggestions

Problem: No auto-suggestions for nearest airports or saved preferences.

Impact: More typing for users; less personalized experience.

Solution: Add location-based APIs (e.g., Google Places) and local storage for recent searches.

---

10. Lack of Booking History or Cancellation Option

Problem: Users can’t view or manage their bookings.

Impact: No transparency or control for users.

Solution: Add user dashboard with booking management features.

13. FUTURE ENHANCEMENTS:

1. Data Accuracy & Real-Time Updates

Problem: Flight schedules, delays, and pricing are highly dynamic.

Impact: Users may see outdated or incorrect results.

Cause: Static or infrequently updated data sources; lack of real-time API integration.
Solution: Integrate with reliable real-time APIs (e.g., Amadeus, Skyscanner, AviationStack).

---

2. API Limitations and Rate Limits

Problem: Free or low-tier APIs have request limits.

Impact: App may stop working or return partial data.

Solution: Use caching, pagination, and upgrade plans as needed.

---

3. Incomplete or Weak Search Filters

Problem: Users can't refine searches well (by stops, airline, time, price).

Impact: Poor user experience; overwhelming result sets.

Solution: Implement advanced filters with intuitive UI.

---

4. Slow Load Times for Results

Problem: Flight data fetching is slow, especially with multiple API calls.

Impact: High bounce rate and user frustration.

Solution: Use async requests, loading spinners, server-side caching, and pagination.

---

5. No or Insecure Payment Handling (if booking is enabled)

Problem: Payment gateway may not be integrated or secured properly.

Impact: Users can't book, or risk of data leaks.

Solution: Use secure, PCI-DSS compliant payment systems like Stripe or Razorpay.

---

6. Testing & Error Handling Gaps

Problem: Unexpected failures during booking, cancellations, or search.

Impact: Crashes, broken features.

Solution: Add comprehensive validation and error boundaries; test with edge cases.

---

7. Authentication and Data Privacy

Problem: Weak or no login/session security.

Impact: Unauthorized access, data leaks.

Solution: Use JWT, encrypted sessions, and HTTPS.

---

8. Poor Mobile Responsiveness

Problem: Layout breaks on mobile.

Impact: Bad UX for mobile users.

Solution: Use responsive design (e.g., Tailwind CSS breakpoints or media queries).

---

9. No Location-Based Suggestions

Problem: No auto-suggestions for nearest airports or saved preferences.

Impact: More typing for users; less personalized experience.

Solution: Add location-based APIs (e.g., Google Places) and local storage for recent searches.

---

10. Lack of Booking History or Cancellation Option

Problem: Users can't view or manage their bookings

Impact: No transparency or control for users.

Solution: Add user dashboard with booking management features.

Future Enhancements for Flight Finder Project (SB Flights)

1. Smart Search with Autocomplete

Feature: City/airport name autocomplete using APIs.

Tools: Google Places API, Algolia, or custom dataset.
Benefit: Speeds up user input and reduces errors.

---

2. Dynamic Price Trends & Prediction

Feature: Show price trends over time or predict future prices.

Tools: Machine learning (e.g., time series forecasting with Prophet).

Benefit: Helps users decide the best time to book.

---

3. **AI-Powered Recommendation**

Feature: Recommend best flights based on user behavior/preferences.

Tools: User profiling + ML models.

Benefit: Personalized flight suggestions.

---

4. **Interactive Seat Selection**

Feature: Visual seat map selection during booking.

Benefit: Improves UX; adds realism.

---

5. **Flight Alerts & Notifications**

Feature: Notify users about:

Price drops

Delays or cancellations

Upcoming flight reminders

Tools: Push notifications, email/SMS (Twilio, OneSignal).

Benefit: Keeps users engaged and informed.

---

6. **Integrated Payment Gateway**

Feature: Secure online payment (Stripe, Razorpay, PayPal).
Enhancement: Add support for UPI, net banking, cards.

Benefit: Enables end-to-end booking functionality.

---

7. **Multi-language & Multi-currency Support**

Feature: Internationalization (i18n) and currency conversion.

Benefit: Reaches a global audience.

---

8. Mobile App Version (React Native or Flutter)

Feature: Native or hybrid mobile app with same backend.

Benefit: Better performance and usability on mobile devices.

---

9. Admin Dashboard

Feature: Admin panel to manage:

- Flight listings
- Bookings
- Users
- API connections

Tech: React.js + Express.js dashboard

Benefit: Full control and monitoring capabilities.

---

10. Booking History & Cancellation Module

Feature: Users can:

- View previous bookings
- Cancel or modify bookings
- Download invoices

Benefit: Enhances user trust and usability.

---

11. Advanced Testing & CI/CD

Feature: Add test suites (unit + integration + UI) and automated deployment.

Tools: Jest, Cypress, GitHub Actions, Docker.

Benefit: Ensures stability and production readiness.

---

12. Offline Booking Support / Progressive Web App (PWA)

Feature: Let users search or view recent results offline.

Benefit: Improves experience in low-connectivity areas

---

13. Enhanced Security

Feature: 2FA, encrypted user data, role-based access control.

Benefit: Increases data safety and regulatory compliance.

---

14. Map-Based Flight Search

Feature: Show available flights on a world map interface.

Tools: Leaflet.js, Mapbox, or Google Maps.

Benefit: Visual search adds an engaging touch.

---

15. Partner Airline Integration

Feature: Direct tie-ups with airlines for real-time seat and fare availability.

Benefit: Removes third-party dependency and adds reliability.

Github link : https://github.com/Ashik1029/FlightFinder-Navigating-Your-Air-Travel-Options
Demo links : https://drive.google.com/drive/folders/1HxqGxNo84_xKrTyA9Ecu3HWaa0QllRZI