

Flight Booking APP

| | |
|---------------|---|
| Team ID | LTVIP2026TMIDS47032 |
| Project Title | Flight finder: navigating your air travel options |

INTRODUCTION

Introducing SB Flights, the ultimate digital platform designed to revolutionize the way you book flight tickets. With SB Flights, your flight travel experience will be elevated to new heights of convenience and efficiency.

Our user-friendly web app empowers travelers to effortlessly discover, explore, and reserve flight tickets based on their unique preferences. Whether you're a frequent commuter or an occasional traveler, finding the perfect flight journey has never been easier.

Imagine accessing comprehensive details about each flight journey at your fingertips. From departure and arrival times to flight classes and available amenities, you'll have all the information you need to make informed decisions. No more guessing or uncertainty – SB Flights ensures that every aspect of your flight travel is crystal clear.

The booking process is a breeze. Simply provide your name, age, and preferred travel dates, along with the departure and arrival cities, and the number of passengers. Once you submit your booking request, you'll receive an instant confirmation of your ticket reservation. No more waiting in long queues or dealing with complicated reservation systems – SB Flights makes it quick and hassle-free.

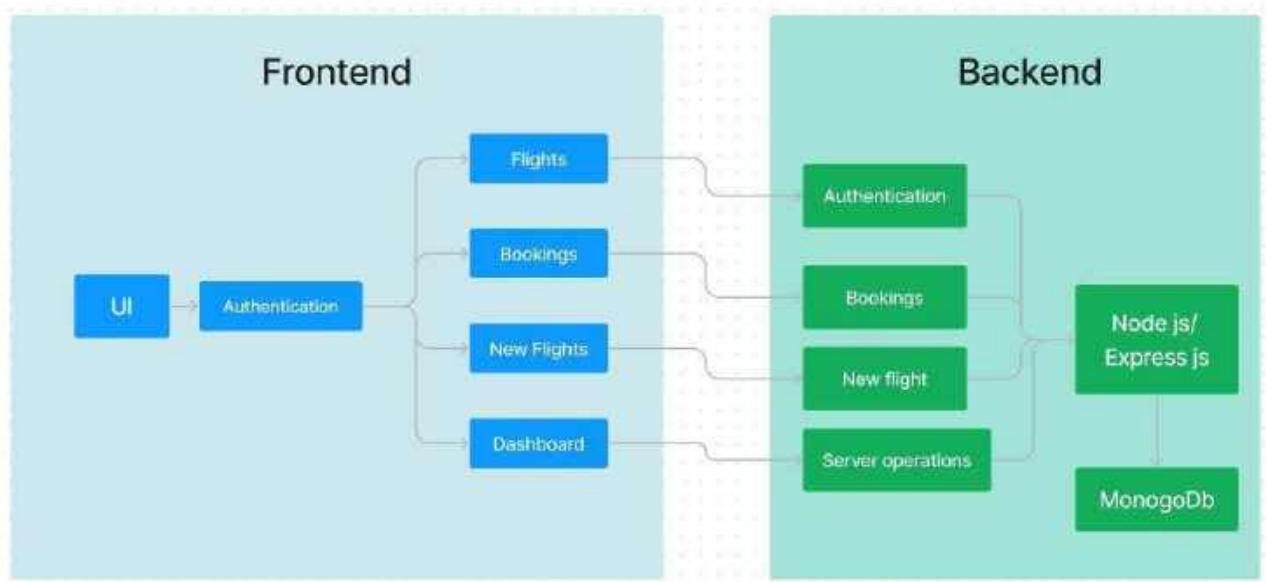
Once your booking is confirmed, our dedicated booking details page becomes your travel companion. It provides a comprehensive overview of your current and previous bookings, allowing you to effortlessly manage your travel plans and stay organized. With SB Flights, you'll have all your essential travel information at your fingertips, ensuring a stress-free journey.

But SB Flights isn't just for travelers. Flight administrators also benefit from our intuitive admin dashboard. This specially designed dashboard empowers administrators to efficiently manage and oversee ticket reservations for their flight service. They can easily view the list of available flights for booking and monitor the bookings made by users. With separate login and registration pages for each flight service, privacy and security are always maintained.

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.

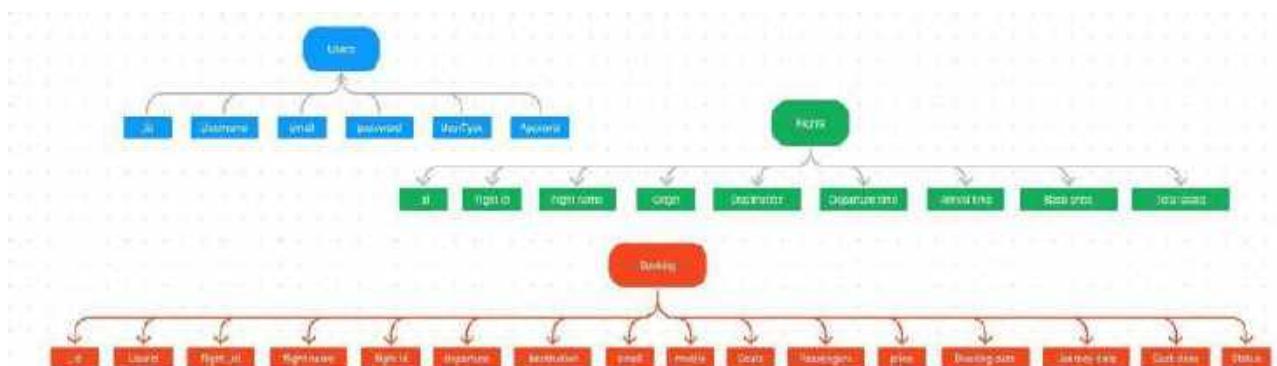
TECHNICAL ARCHITECTURE:



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.

PREREQUISITES:

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 serverside 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 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/harsha-vardhan-reddy-07/Flight-Booking-App-MERN>

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.

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.,

PROJECT STRUCTURE:

```
FLIGHT BOOKING APP MERN
```

```
client
  node_modules
  public
  src
    assets
    components
      Login.jsx
      Navbar.jsx
      Register.jsx
    context
      GeneralContext.jsx
    pages
      Admin.jsx
      AllBookings.jsx
      AllFlights.jsx
      AllUsers.jsx
      Authenticate.jsx
      BookFlight.jsx
      Bookings.jsx
      EditFlight.jsx
      FlightAdmin.jsx
      FlightBookings.jsx
      FlightRequests.jsx
      Flights.jsx
      LandingPage.jsx
      NewFlight.jsx
    RouteProtectors
    styles
    # App.css
  JS App.js
  JS App.test.js
  # index.css
  JS index.js
server
  node_modules
  JS index.js
  package-lock.json
  package.json
  JS schemas.js
```

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/1CQil5KzGnPvkVOPWTLPOh-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. **Code Explanation:**

Server setup:

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

```
js index.js  ×
server > js 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/lightbookingDB', [
 19   useNewUrlParser: true,
 20   useUnifiedTopology: true,
 21 ]
 22 ).then(()=>{
 23
 24
 25   // All the client-server activites
 26
```

Schemas:

Now let us define the required schemas

```
JS schemas.js ×
server > JS schemas.js > ...
1 import mongoose from "mongoose";
2
3 const userSchema = new mongoose.Schema({
4   username: { type: String, required: true },
5   email: { type: String, required: true, unique: true },
6   usertype: { type: String, required: true },
7   password: { type: String, required: true },
8   approval: {type: String, default: 'approved'}
9 });
10 const flightSchema = new mongoose.Schema({
11   flightName: { type: String, required: true },
12   flightId: { type: String, required: true },
13   origin: { type: String, required: true },
14   destination: { type: String, required: true },
15   departureTime: { type: String, required: true },
16   arrivalTime: { type: String, required: true },
17   basePrice: { type: Number, required: true },
18   totalSeats: { type: Number, required: true }
19 });
20 const bookingSchema = new mongoose.Schema({
21   user: { type: mongoose.Schema.Types.ObjectId, ref: "User", required: true },
22   flight: { type: mongoose.Schema.Types.ObjectId, ref: "Flight", required: true },
23   flightName: {type: String, required: true},
24   flightId: {type: String},
25   departure: {type: String},
26   destination: {type: String},
27   email: {type: String},
28   mobile: {type: String},
29   seats: {type: String},
30   passengers: [
31     {
32       name: { type: String },
33       age: { type: Number }
34     }
35   ],
36   totalPrice: { type: Number },
37   bookingDate: { type: Date, default: Date.now },
38   journeyDate: { type: Date },
39   journeyTime: { type: String },
40   seatClass: { type: String },
41   bookingStatus: {type: String, default: "confirmed"}
42 });
43 export const User = mongoose.model('users', userSchema);
44 export const Flight = mongoose.model('Flight', flightSchema);
45 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.

```
JS index.js ×
server > JS 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 });


```

- **Frontend**

Login:

```

  GeneralContext.js X
client > src > context > GeneralContext.js > [!1] GeneralContextProvider
  21 const login = async () => {
  22   try{
  23     const loginInputs = {email, password}
  24     await axios.post('http://localhost:5001/login', loginInputs)
  25     then( 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.js X
client > src > context > GeneralContext.js > [!2] GeneralContextProvider > [!3] register
  47
  48   const inputs = {username, email, usertype, password};
  49   const register = async () =>{
  50     try{
  51       await axios.post('http://localhost:5001/register', inputs)
  52       .then( 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 }

```

Logout:

```

  GeneralContext.js X
client > src > context > GeneralContext.js > [!3] GeneralContextProvider > [!4] login
  72
  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 }

```

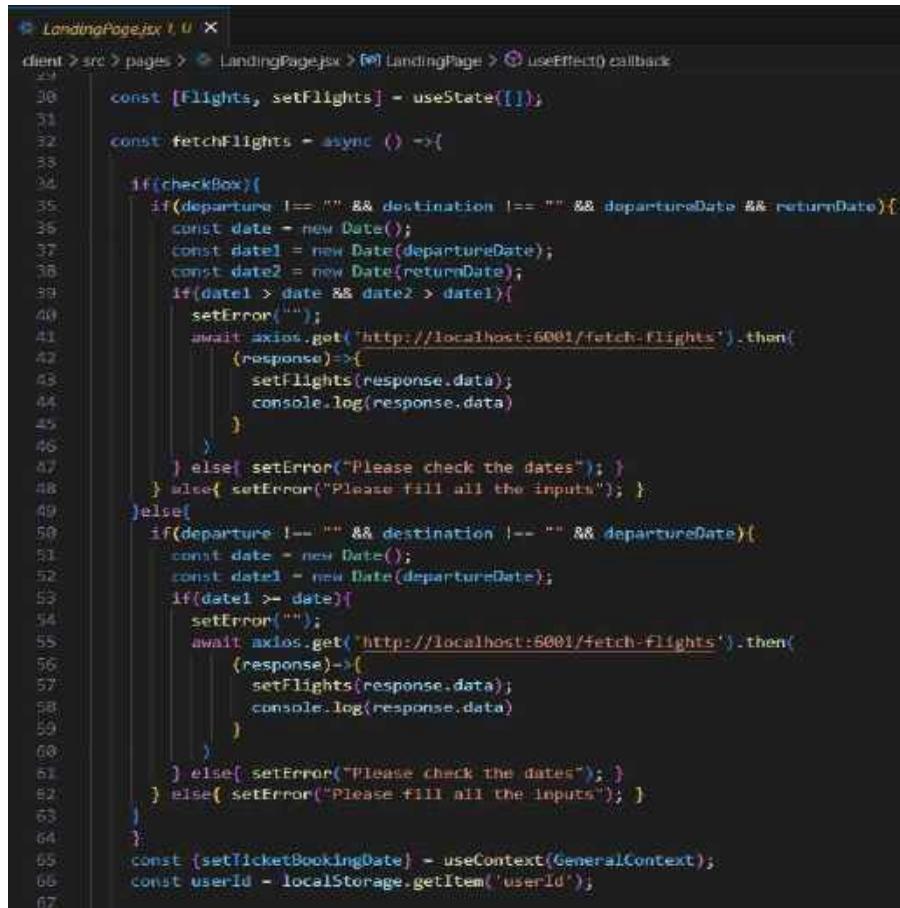
Flight Booking (User):

- 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.,

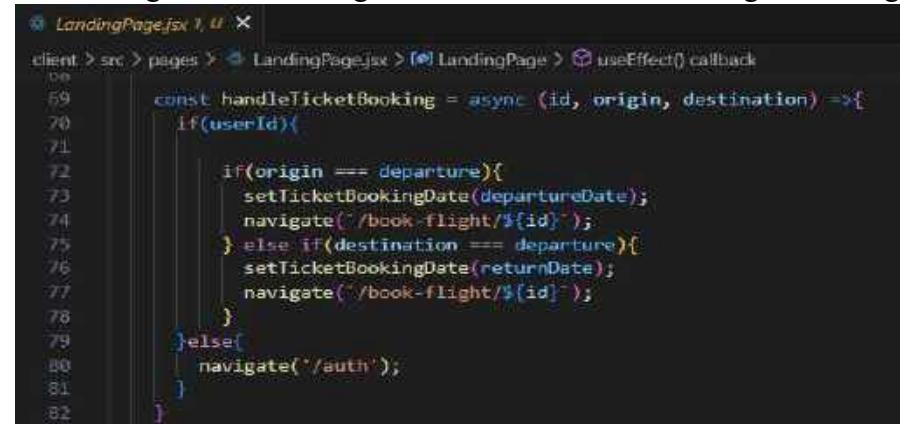
Flight Searching code:

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.



```
client > src > pages > ⚡ LandingPage.jsx > 🐛 LandingPage > 🎨 useEffect() callback
  30  const [flights, setFlights] = useState([]);
  31
  32  const fetchFlights = async () => {
  33
  34    if(checkBox){
  35      if(departure === "" && destination === "" && departureDate && returnDate){
  36        const date = new Date();
  37        const date1 = new Date(departureDate);
  38        const date2 = new Date(returnDate);
  39        if(date1 > date && date2 > date1){
  40          setError("");
  41          await axios.get('http://localhost:6001/fetch-flights').then(
  42            (response)=>{
  43              setFlights(response.data);
  44              console.log(response.data)
  45            }
  46          )
  47        } else{ setError("Please check the dates"); }
  48      } else{ setError("Please fill all the inputs"); }
  49    }else{
  50      if(departure === "" && destination === "" && departureDate){
  51        const date = new Date();
  52        const date1 = new Date(departureDate);
  53        if(date1 >= date){
  54          setError("");
  55          await axios.get('http://localhost:6001/fetch-flights').then(
  56            (response)=>{
  57              setFlights(response.data);
  58              console.log(response.data)
  59            }
  60          )
  61        } else{ setError("Please check the dates"); }
  62      } else{ setError("Please fill all the inputs"); }
  63    }
  64  }
  65  const {setTicketBookingDate} = useContext(GeneralContext);
  66  const userId = localStorage.getItem('userId');
  67
```

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



```
client > src > pages > ⚡ LandingPage.jsx > 🐛 LandingPage > 🎨 useEffect() callback
  68
  69  const handleTicketBooking = async (id, origin, destination) =>{
  70    if(userId){
  71
  72      if(origin === departure){
  73        setTicketBookingDate(departureDate);
  74        navigate('/book-flight/${id}');
  75      } else if(destination === departure){
  76        setTicketBookingDate(returnDate);
  77        navigate('/book-flight/${id}');
  78      }
  79    }else{
  80      navigate('/auth');
  81    }
  82  }
```

- **Backend**

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

```

JS index.js X
server > JS index.js> ⚡ then() callback
129   // fetch trains
130
131   app.get('/fetch-trains', async (req, res) =>{
132
133     try{
134       const trains = await Train.find();
135       res.json(trains);
136
137     }catch(err){
138       console.log(err);
139     }
140   })
141

```

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

```

# index.js X
server > # index.js> ⚡ then() callback > ⚡ app.put('/cancel-ticket/:id') callback
141
142   // Book ticket
143
144   app.put('/book-ticket', async (req, res) =>{
145     const user, flight, flightName, flightId, departure, destination,
146           email, mobile, passengers, totalPrice, journeyDate, journeyTime, seatClass = req.body;
147
148     try{
149       const bookings = await Booking.find({flight: flight, journeyDate: journeyDate, seatClass: seatClass});
150       const numBookedSeats = bookings.reduce((acc, booking) => acc + booking.passengers.length, 0);
151
152       let seats = '';
153       const seatCode = ('economy': 'E', 'premium-economy': 'P', 'business': 'B', 'first-class': 'A');
154       let coach = seatCode[seatClass];
155       for(let i = numBookedSeats + 1; i < numBookedSeats + passengers.length; i++){
156         if(seats === ''){
157           seats = seats.concat(coach, '-1', 1);
158         }else{
159           seats = seats.concat(',', coach, '-1', i);
160         }
161       }
162
163       const booking = new Booking(user, flight, flightName, flightId, departure, destination,
164                                 email, mobile, passengers, totalPrice, journeyDate, journeyTime, seatClass, seats);
165       await booking.save();
166       res.json({message: 'Booking successful!'});
167     }catch(err){
168       console.log(err);
169     }
170   })
171

```

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
```

```
client > src > pages > Bookings.jsx > Bookings
  8  const [bookings, setBookings] = useState([]);
  9
 10  const userId = localStorage.getItem('userId');
 11
 12  useEffect(()=>{
 13    | fetchBookings();
 14  }, [])
 15
 16  const fetchBookings = async ()=>{
 17    await axios.get('http://localhost:6001/Fetch-bookings').then(
 18      (response)=>{
 19        setBookings(response.data);
 20      }
 21    )
 22  }
 23  const cancelTicket = async (id) =>{
 24    await axios.put(`http://localhost:6001/cancel-ticket/${id}`).then(
 25      (response)=>{
 26        alert("Ticket cancelled!!!");
 27        fetchBookings();
 28      }
 29    )
 30  }
 31
```

- **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
  160  app.get('/fetch-bookings', async (req, res)=>[
 161
 162    try{
 163      const bookings = await Booking.find();
 164      res.json(bookings);
 165
 166    }catch(err){
 167      console.log(err);
 168    }
 169  })
 170
```

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

```
index.js
```

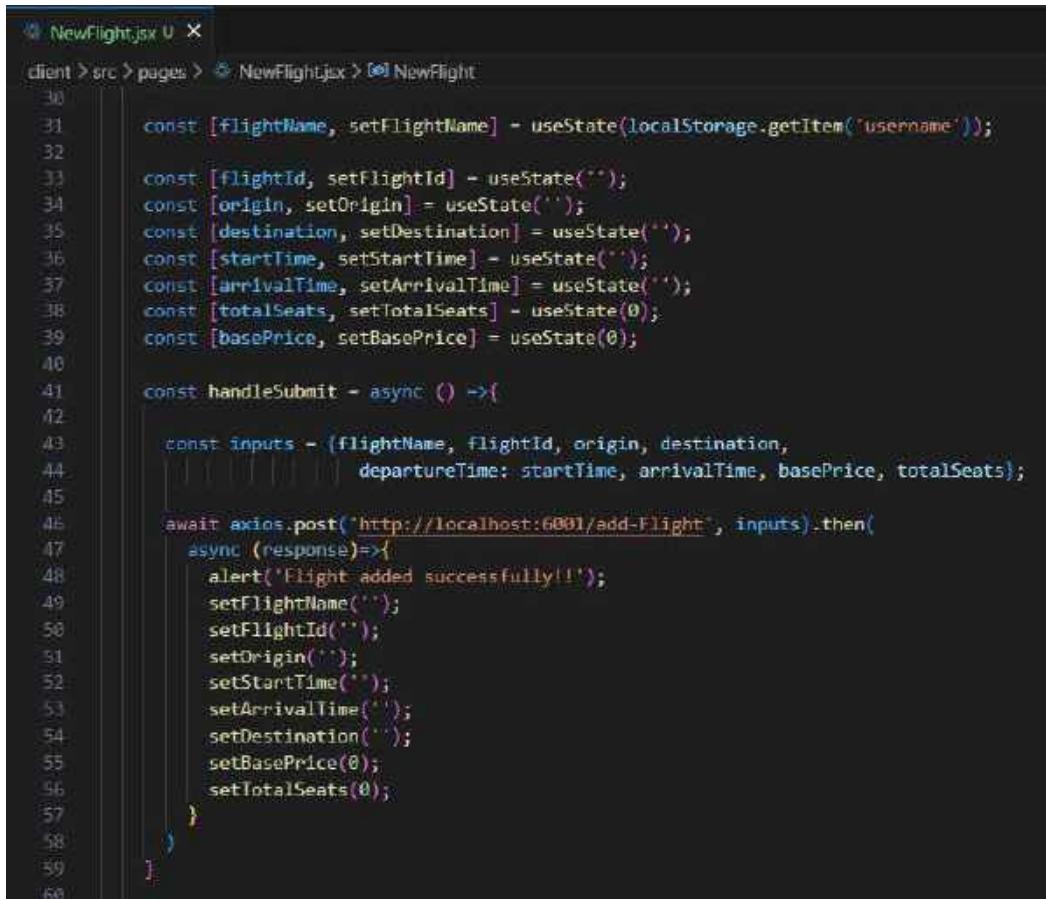
```
server > index.js > ...
  205  app.put('/cancel-ticket/:id', async (req, res)=>{
 206    const id = await req.params.id;
 207    try{
 208      const booking = await Booking.findById(req.params.id);
 209      booking.bookingStatus = 'cancelled';
 210      await booking.save();
 211      res.json({message: "booking cancelled"});
 212
 213    }catch(err){
 214      console.log(err);
 215    }
 216  })
```

Add new flight:

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

- **Frontend**

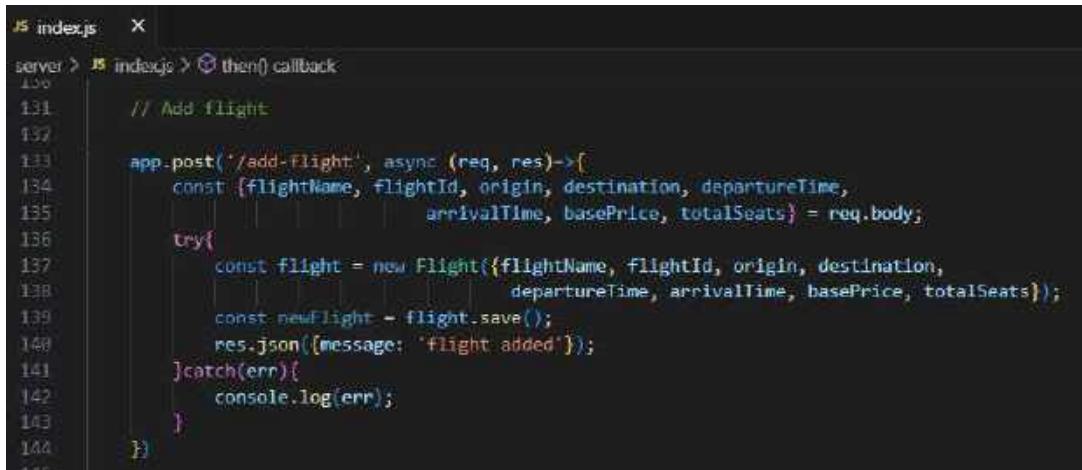
We create a html form with required inputs for the new flight and then send an http request to the server to add it to database.



```
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.



```
server > index.js > ⚡ then() callback
  100
  101   // Add flight
  102
  103   app.post('/add-flight', async (req, res)=>[
  104     const {flightName, flightId, origin, destination, departureTime,
  105           arrivalTime, basePrice, totalSeats} = req.body;
  106     try{
  107       const flight = new Flight({flightName, flightId, origin, destination,
  108                               departureTime, arrivalTime, basePrice, totalSeats});
  109       const newFlight = flight.save();
  110       res.json({message: 'flight added'});
  111     }catch(err){
  112       console.log(err);
  113     }
  114   ])
  115 }
```

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:**

```
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  }
```

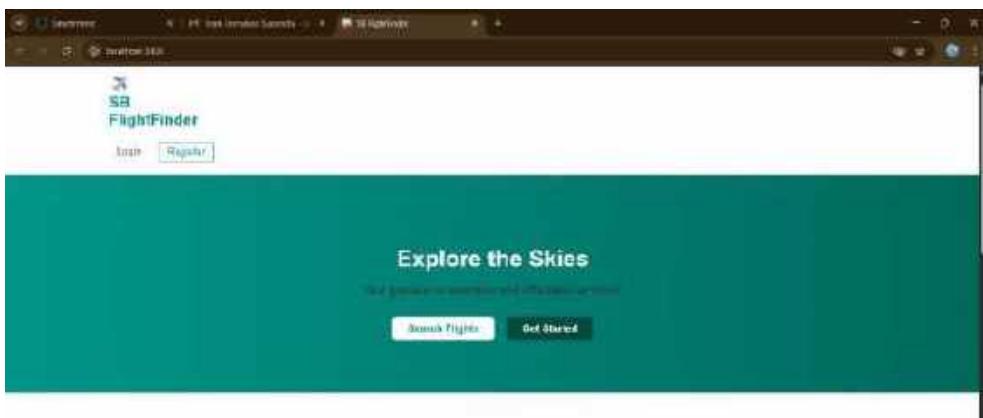
o Backend:

```
server > index.js > then0 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.

Screenshots

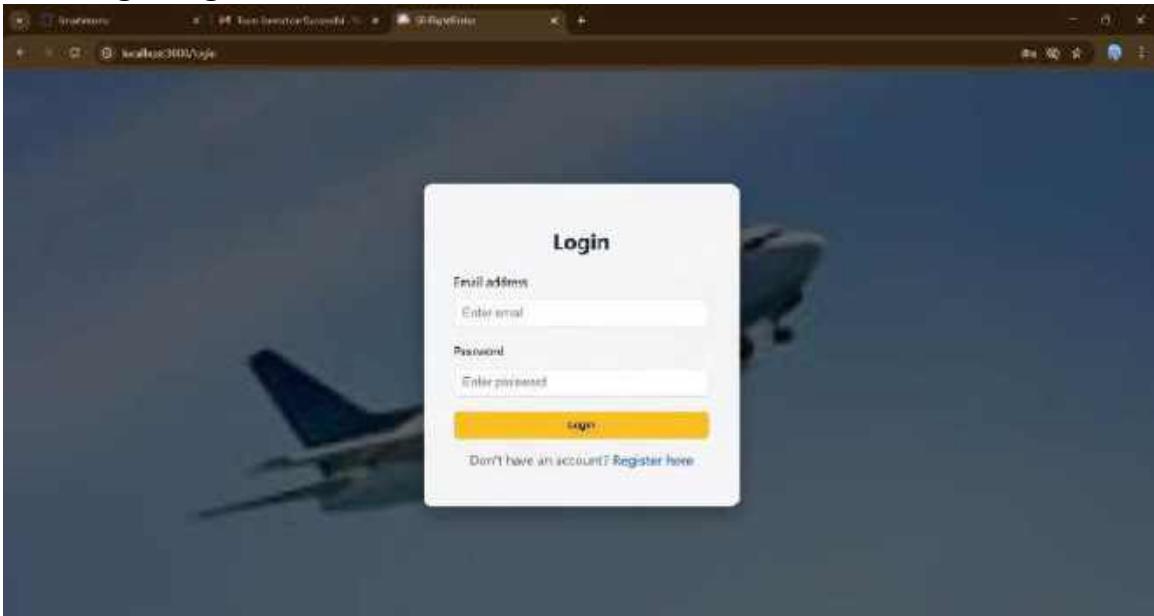
1. Home Page



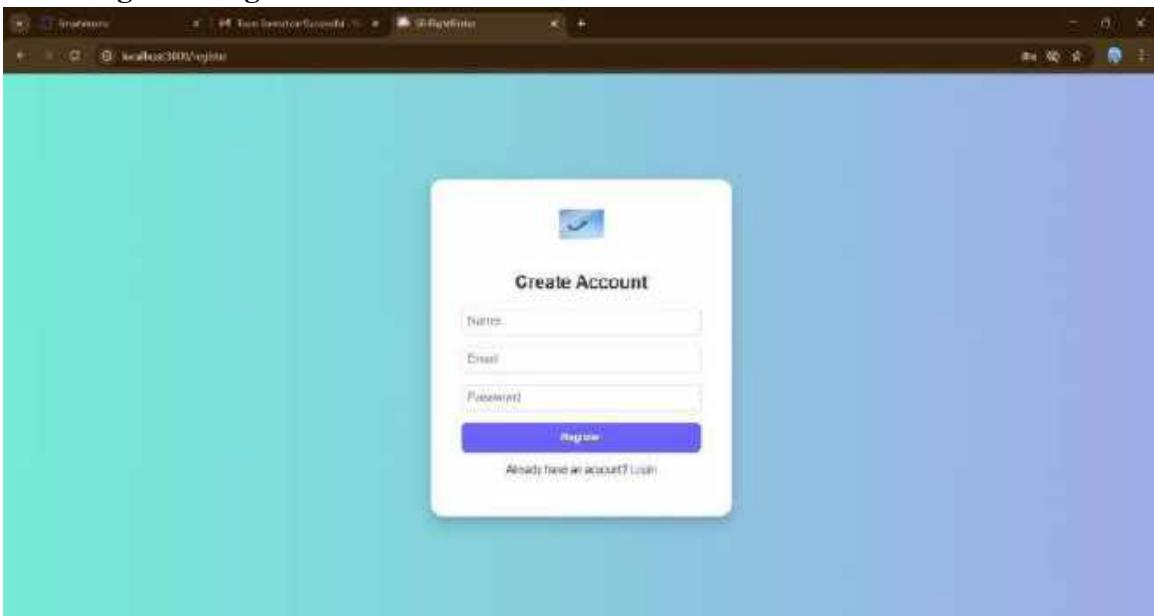
Why Choose FlightFinder?

- Real-time flight search and availability
- Enhanced flight booking with flight search options
- Loyalty rewards and frequent flier tools

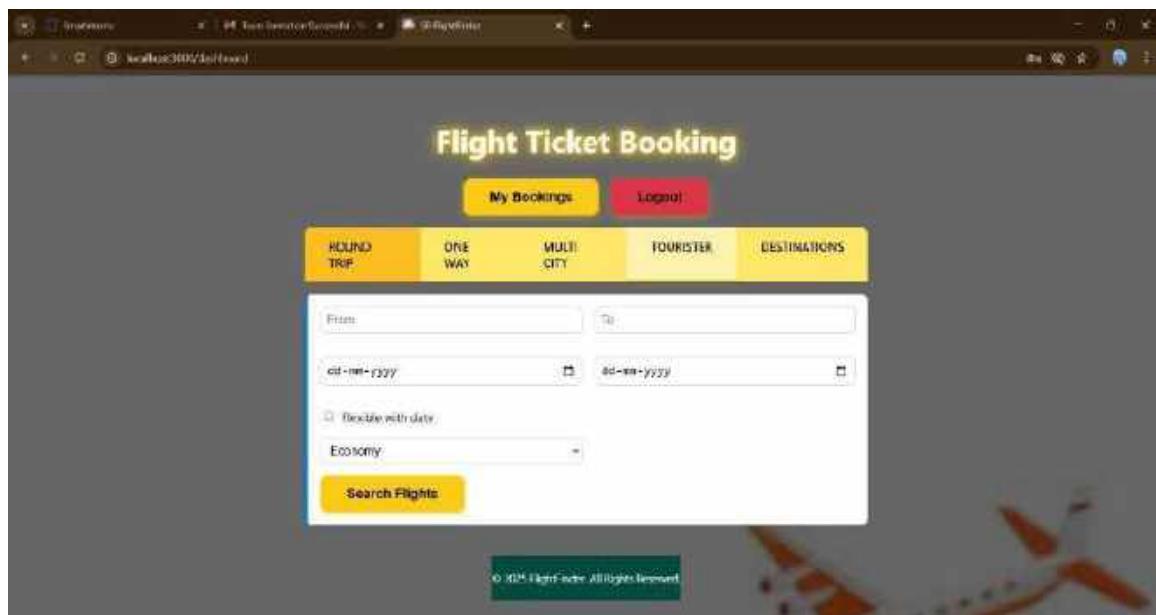
2. Login Page



3. Register Page



4. Dashboard



5.Flight search

A screenshot of the flight search results page. The top header shows the search criteria: "From: New York To: Paris Start: 2025-06-15 Return: 2025-06-15". Below this, there are two tables: "Departure Flights" and "Return Flights".

Departure Flights

| Flight | From | To | Departure | Arrival | Date | Price | Action |
|---------|----------|-------|-----------|---------|------------|--------|---------------------------------|
| Eirates | New York | Paris | 12:20 | 01:30 | 2025-06-10 | \$1000 | View All Result |

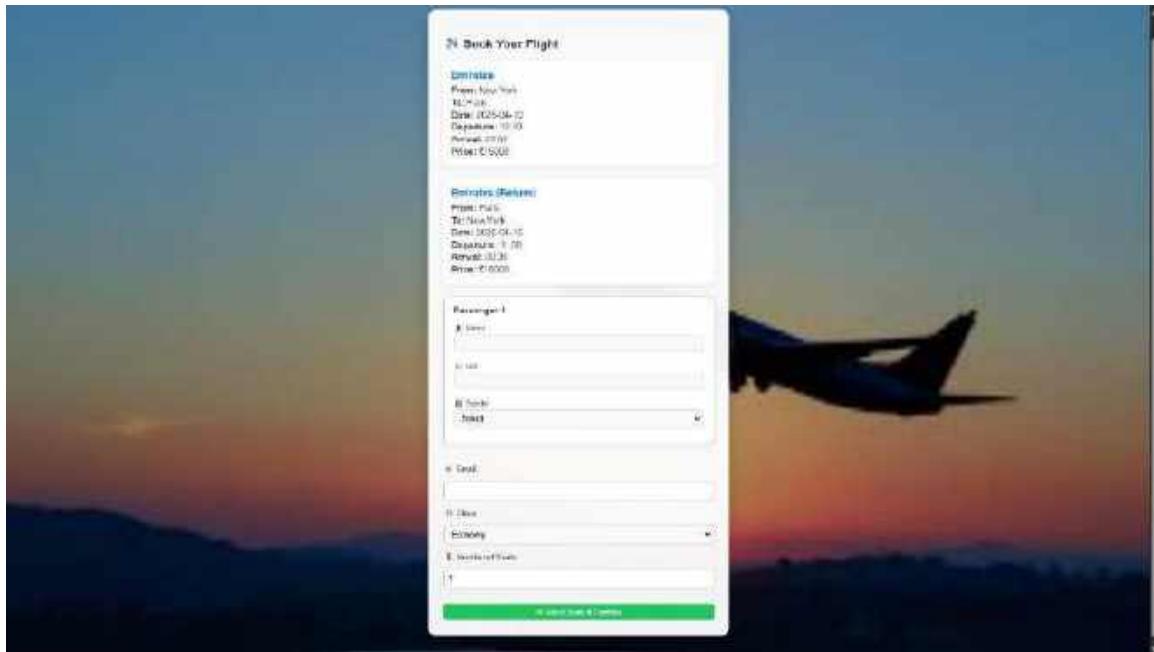
Return Flights

| Flight | From | To | Departure | Arrival | Date | Price |
|---------|-------|----------|-----------|---------|------------|--------|
| Eirates | Paris | New York | 11:30 | 00:30 | 2025-06-15 | \$1000 |

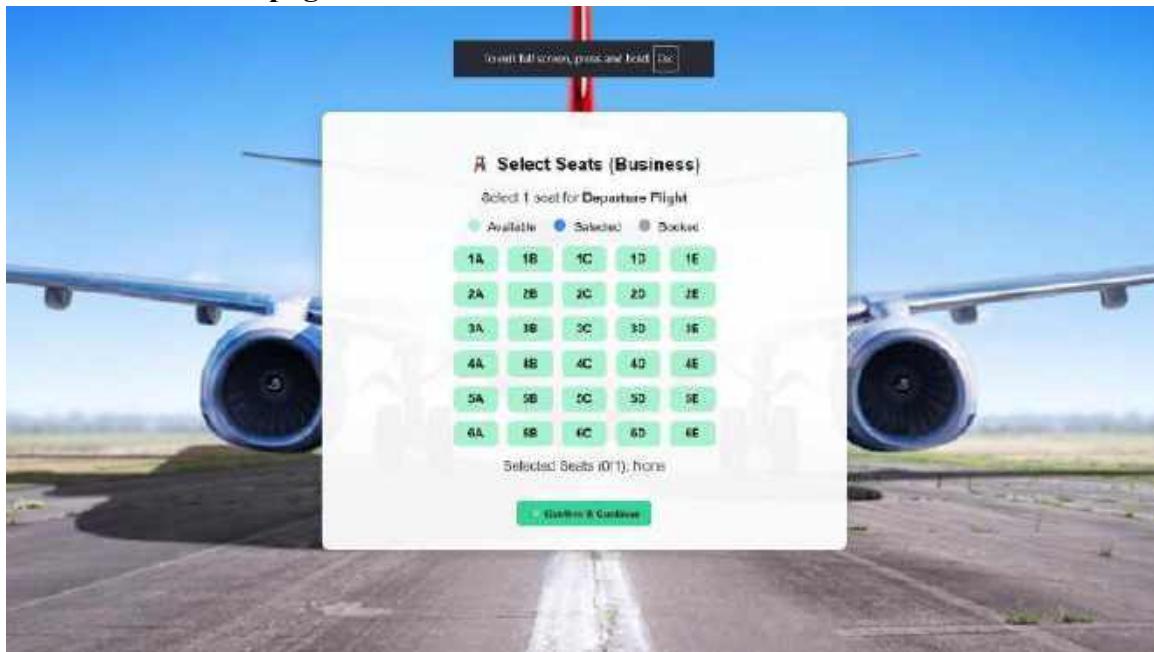
[← Return to Dashboard](#)

The background features a blurred image of a sunset over mountains.

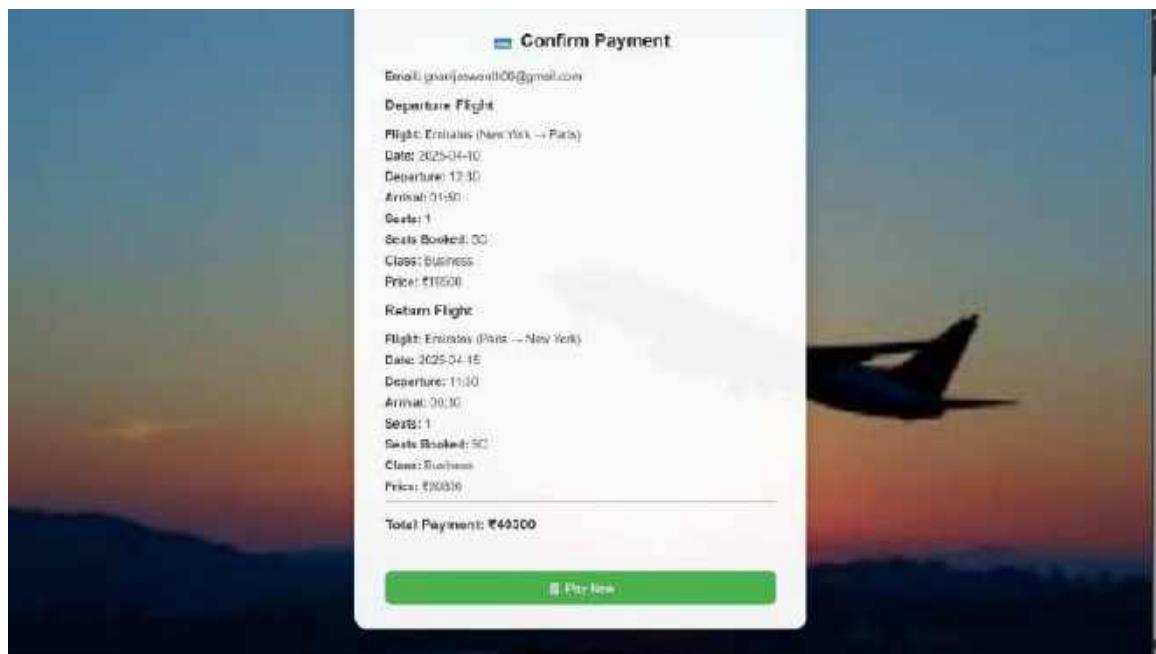
6.booking page



6.seat selection page



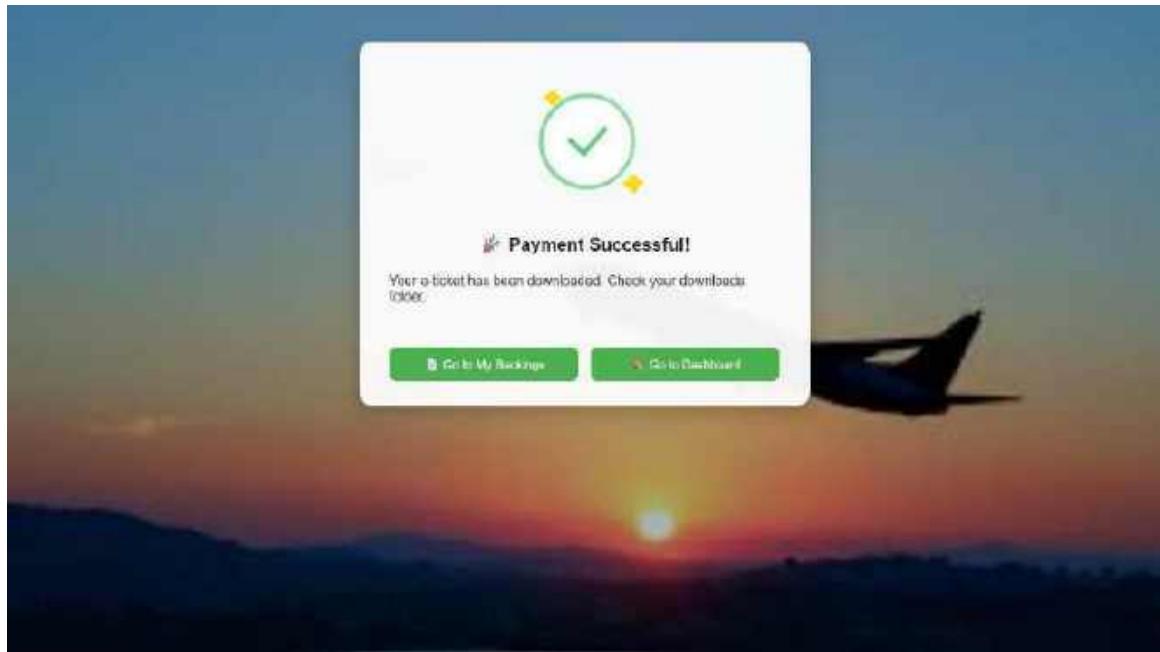
7.payment page



7.transaction page

This is a payment transaction page. At the top, it says "New business services" and "Flight: Emirates (New York - Paris) ₹40,300.00". Below this, there's a "Card" section with a placeholder "Card number" field containing "4444 3333 2222 1111". To the right, there are fields for "Email" (gauravjeetw00@gmail.com), "Last name" (John), and "Country or region" (India). There's also a "Save my information for faster checkout" checkbox. At the bottom, a large blue "Pay" button is prominent, with a note below it stating: "After clicking the Pay button, you will be redirected to a secure payment gateway where you can complete your payment." Below the button, there are links for "Cancel" and "Print".

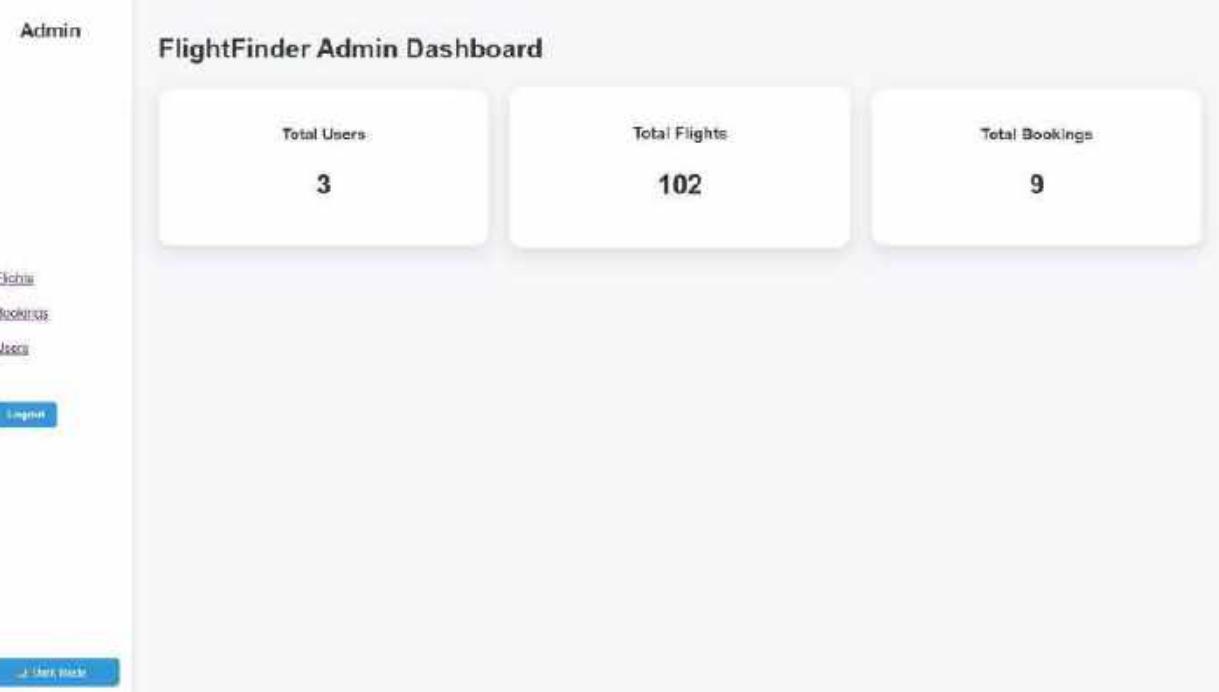
8.confirmation page



9.my bookings page

| My Flight Bookings | | | | | | |
|--------------------|-----------|-------|-----------|------|-----------|--|
| Flight | From | To | Date | Seat | Status | |
| Emirates | New York | Paris | 4/19/2025 | 1 | Confirmed | |
| Emirates | New York | Paris | 4/10/2025 | 1 | Confirmed | |
| Qatar | Hyderabad | Delhi | 1/1/2025 | 1 | Confirmed | |
| Emirates | New York | Paris | 4/16/2025 | 1 | Confirmed | |
| Emirates | New York | Paris | 4/10/2025 | 1 | Confirmed | |
| Qatar | Hyderabad | Delhi | 1/1/2025 | 1 | Confirmed | |
| Air India | Hyderabad | Delhi | 1/1/2025 | 1 | Confirmed | |
| Singapore Airlines | Hyderabad | Delhi | 6/29/2025 | 1 | Confirmed | |
| Emirates | New York | Paris | 4/10/2025 | 1 | Confirmed | |

10.admin dashboard page



Demo Instructions

1. Set up environment variables

- Create a `'.env'` file in `/server` and `/client` (if needed)

2. Install dependencies cd

```
server && npm install  
cd  
../client && npm  
install
```

3. Run locally cd server

```
&& npm run dev  
cd  
../client && npm start
```

4. Open <http://localhost:3000> in browser

Future Improvements

- Flight filters (duration, airline, stops)
- Email ticket confirmations
- Mobile responsive redesign
- Admin notifications
- Chat support

Contributors

- Koduru Balaji
- Arun Partha Kullakanda
- Sudheer Kumar Malakanti
- Mamanduru Eswar Reddy