Flight Booking APP

INTRODUCTION

Introducing **AirVoyager**, a comprehensive and intelligent flight booking platform built using the powerful **MERN (MongoDB, Express.js, React.js, Node.js)** stack. AirVoyager is designed to modernize the flight booking experience by offering a seamless, intuitive, and role-based system for users, flight operators, and administrators. 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.

AirVoyager empowers travelers to easily search, explore, and book flights based on their preferences. Whether you're a regular flyer or planning a one-time trip, the platform ensures that discovering the right flight is quick and user-friendly. Users can browse flight details such as departure and arrival cities, timings, airline, and seat availability in real-time before making a reservation. 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.

Booking a flight with AirVoyager is simple. Users just need to enter the necessary passenger information along with the desired travel route and dates. Once booked, the system instantly confirms the ticket and provides a dedicated **"My Bookings"** section where users can view all current and past bookings, as well as cancel any upcoming flights if needed.

What sets AirVoyager apart is its **role-based access system**:

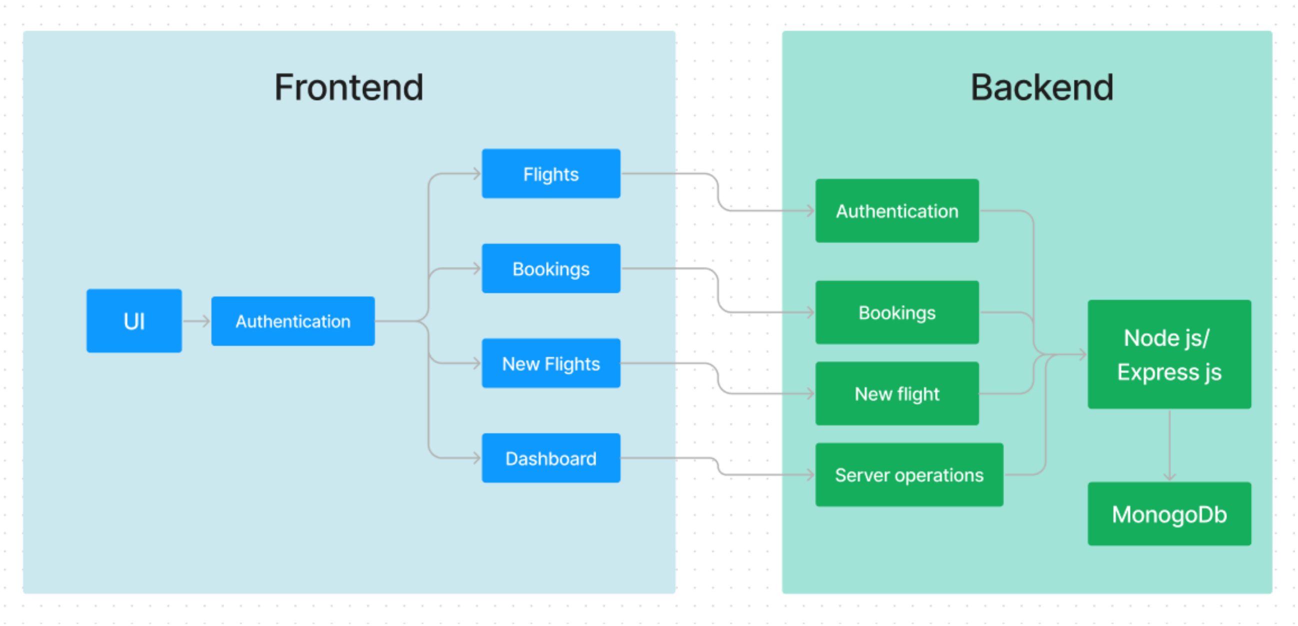
- **Users** can search and book flights, and manage bookings.
- **Flight Operators** have access to a specialized dashboard to add, update, and view the flights they manage. They can also track bookings made by users on their flights and view passenger details.
- **Administrators** hold the responsibility of approving newly registered flight operators and monitoring all registered users and flight records in the system.

  The system includes secure login and registration flows, admin approval for flight operators, and dynamic booking updates, all managed in a highly responsive UI built with React.

  AirVoyager is not just a booking app — it is a fully scalable flight management solution that enhances convenience, improves efficiency, and ensures a smooth travel planning experience for every type of user.

  Get ready to explore the skies with **AirVoyager** — where booking flights becomes smart, fast, and effortless.

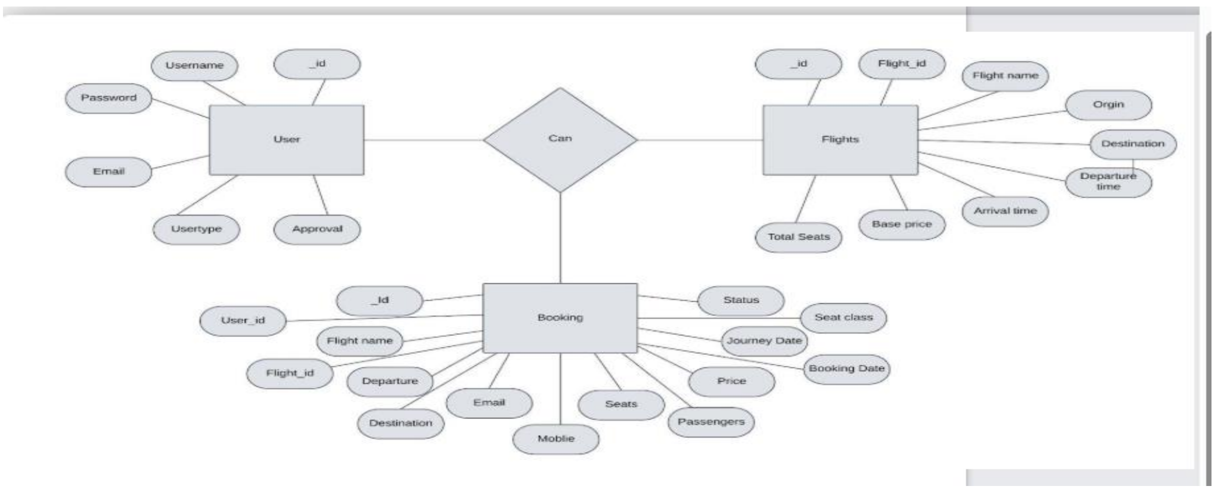
TECHINICAL ARCHITECTURE:



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 of AirVoyager

1. **Extensive Flight Listings**
   AirVoyager presents a dynamic list of available flights added by registered flight operators. Users can browse flights by filtering based on origin, destination, and travel date. Each listing provides important details including airline name, flight number, departure and arrival times, price, and seat availability.
2. **User-Friendly Flight Booking**
   Once the user selects a flight, clicking the **"Book Now"** button takes them to a dedicated booking page. Here, users can enter passenger information including name and email for each traveler, making it simple and convenient to finalize the reservation.
3. **My Bookings Dashboard**
   After booking, users can view all their upcoming and past reservations in the **"My Bookings"** section. From here, they can review booking details such as flight info, passenger list, and travel dates. Users are also empowered with a **"Cancel Booking"** option if plans change.

4. **Flight Operator Panel**
Flight operators have a dedicated dashboard where they can:

- Add new flights with complete information
- View and manage their existing flights
- Edit or update flight details
- Track bookings made on their flights
- View users who booked their flights

5. **Admin Dashboard**
Admins can:

- View all users and flight operators
- Approve or reject pending flight operator registrations
- Monitor bookings across the platform
- Ensure system integrity and user control

6. **Secure Authentication System**
AirVoyager provides role-based login with secure JWT-based authentication. There are separate logins for users, flight operators, and admins, with restricted access to specific dashboards based on roles.

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 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/Harshitha-2210/FlightFinder-Navigating-Your-Air-Travel-Options

Install Dependencies:

- Navigate into the cloned repository directory:

  cd FlightFinder

- Install the required dependencies by running the following command:
  Install both **client** and **server** dependencies separately:

```
cd client
npm install

cd ../server
npm install
```

Environment Variables Setup:

In the /server directory, create a .env file with the following content:

```
MONGO_URI=your_mongodb_connection_string
JWT_SECRET=your_jwt_secret
```

Start the Development Servers:

You can run both servers individually:

```
cd server
node index.js

cd client
npm start
```

Access the Application

Once both servers are running:

- Visit the frontend at: http://localhost:3000
- Backend API runs on: http://localhost:5000

You should see the AirVoyager flight booking homepage load in the browser.

✅ You’re All Set!

You have successfully set up and launched the AirVoyager MERN stack flight booking application on your local machine. You can now continue with customization, feature enhancement, or testing!

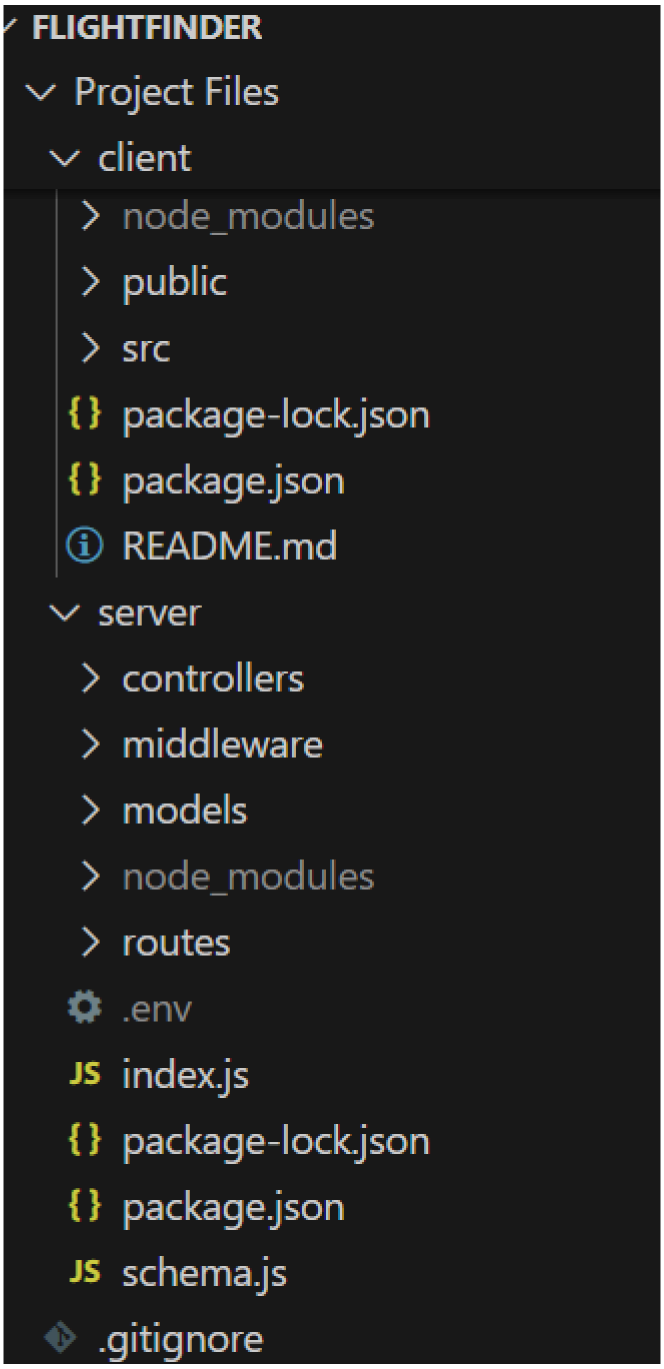
USER & ADMIN FLOW:

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:



```
FLIGHTFINDER
  Project Files
    client
      node_modules
      public
      src
      package-lock.json
      package.json
      README.md
    server
      controllers
      middleware
      models
      node_modules
      routes
      .env
      index.js
      package-lock.json
      package.json
      schema.js
    .gitignore
```

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 Use Case

1. User Schema

- **Schema Name**: userSchema
- **Model Name**: 'User'
- **Purpose**:
  The userSchema defines the structure for user data in the application and is used for both registration and authentication.
- **Fields Include**:
  - name: User's full name
  - email: Unique identifier for login (marked as unique)
  - password: Encrypted password
  - role: Defines the user type — "user", "admin", or "operator"
  - isApproved: Used specifically for flight operators to indicate admin approval

🛂 This schema is critical for **access control**, allowing different features for Admin, User, and Operator roles.

Flight Schema

- **Schema Name:** flightSchema
- **Model Name:** 'Flight'
- **Purpose:**
  The flightSchema manages all flight-related data and is used by flight operators to **create, view, edit, or delete flights**.
- **Fields Include:**
  - airline: Airline name (e.g., Indigo, Air India)
  - flightNumber: Unique flight identifier
  - departure: Departure city
  - destination: Arrival city
  - departureTime & arrivalTime: Date-time of travel
  - price: Price per ticket
  - availableSeats: Track remaining seats
  - createdBy: Reference to the operator (User ID)

🛫 Enables the full CRUD functionality for operators managing their flights.

---

3. Booking Schema

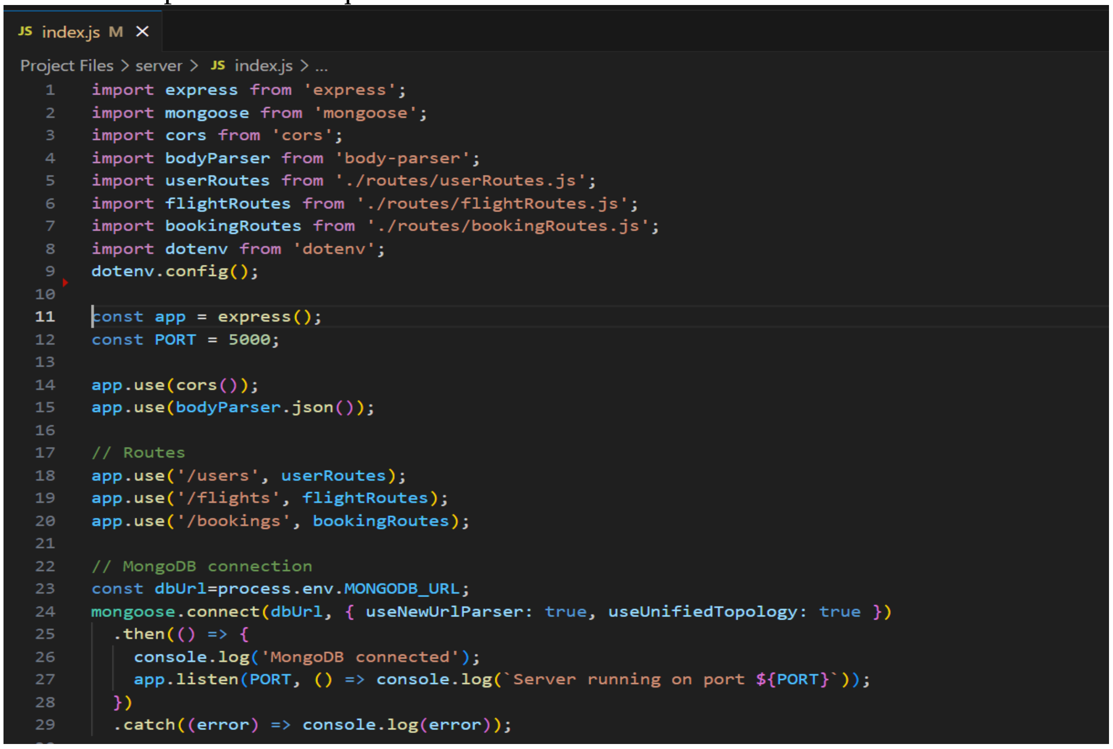
- **Schema Name:** bookingSchema
- **Model Name:** 'Booking'
- **Purpose:**
  The bookingSchema stores **booking records** made by users and links them to the respective flight and user.
- **Fields Include:**
  - userId: Reference to the user who booked
  - flightId: Reference to the booked flight
  - passengers: Number of seats booked
  - passengerDetails: Array of objects (each with name and email)
  - bookingDate: Timestamp when booking was made

▯ This schema ensures that **each user's booking history** is stored and retrievable.

Code Explanation:

Server setup:

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



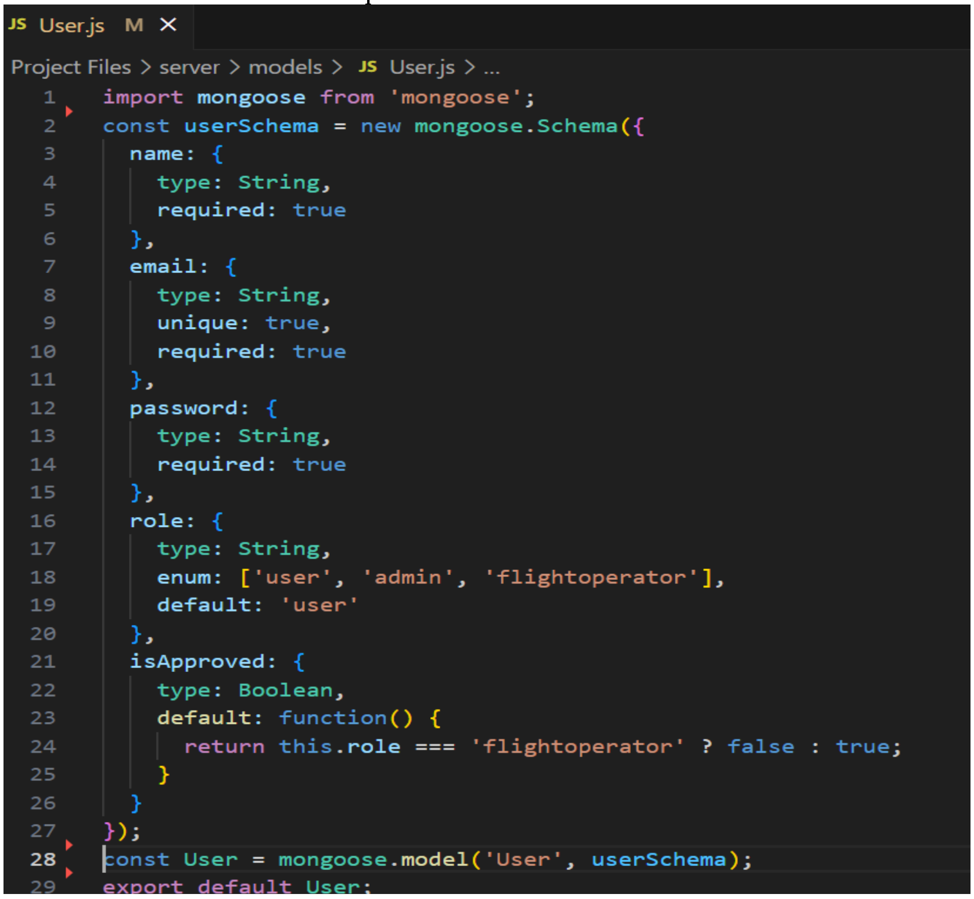
JS index.js M ×

Project Files > server > JS index.js > ...

```
import express from 'express';
import mongoose from 'mongoose';
import cors from 'cors';
import bodyParser from 'body-parser';
import userRoutes from './routes/userRoutes.js';
import flightRoutes from './routes/flightRoutes.js';
import bookingRoutes from './routes/bookingRoutes.js';
import dotenv from 'dotenv';
dotenv.config();

const app = express();
const PORT = 5000;

app.use(cors());
app.use(bodyParser.json());

// Routes
app.use('/users', userRoutes);
app.use('/flights', flightRoutes);
app.use('/bookings', bookingRoutes);

// MongoDB connection
const dbUrl=process.env.MONGODB_URL;
mongoose.connect(dbUrl, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => {
    console.log('MongoDB connected');
    app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
  })
  .catch((error) => console.log(error));
```

Schemas:

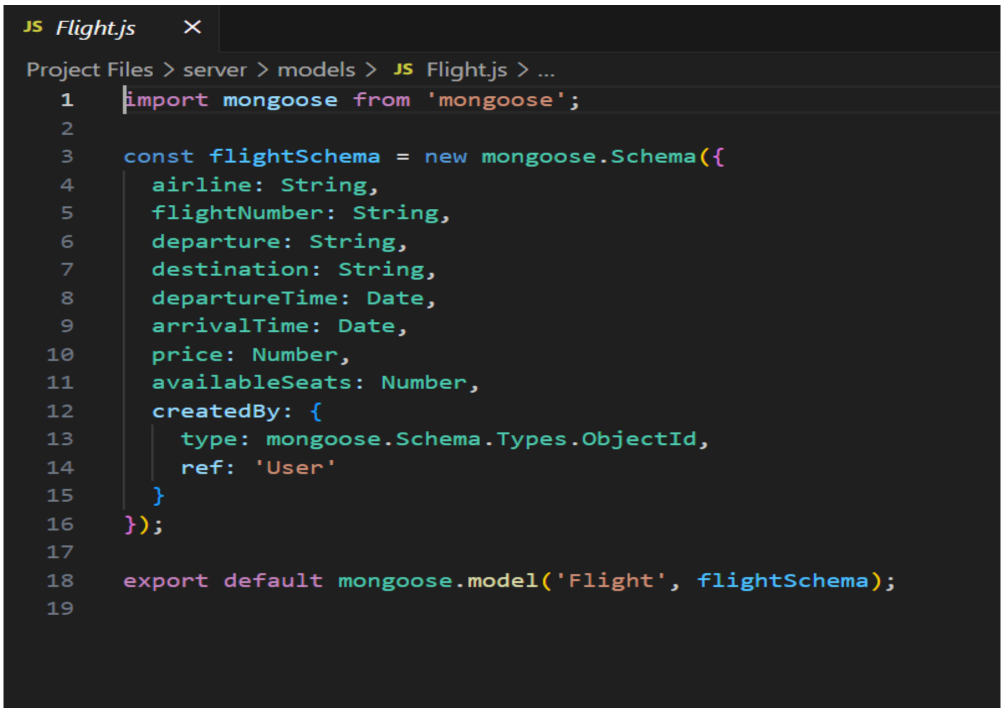
Now let us define the required schemas



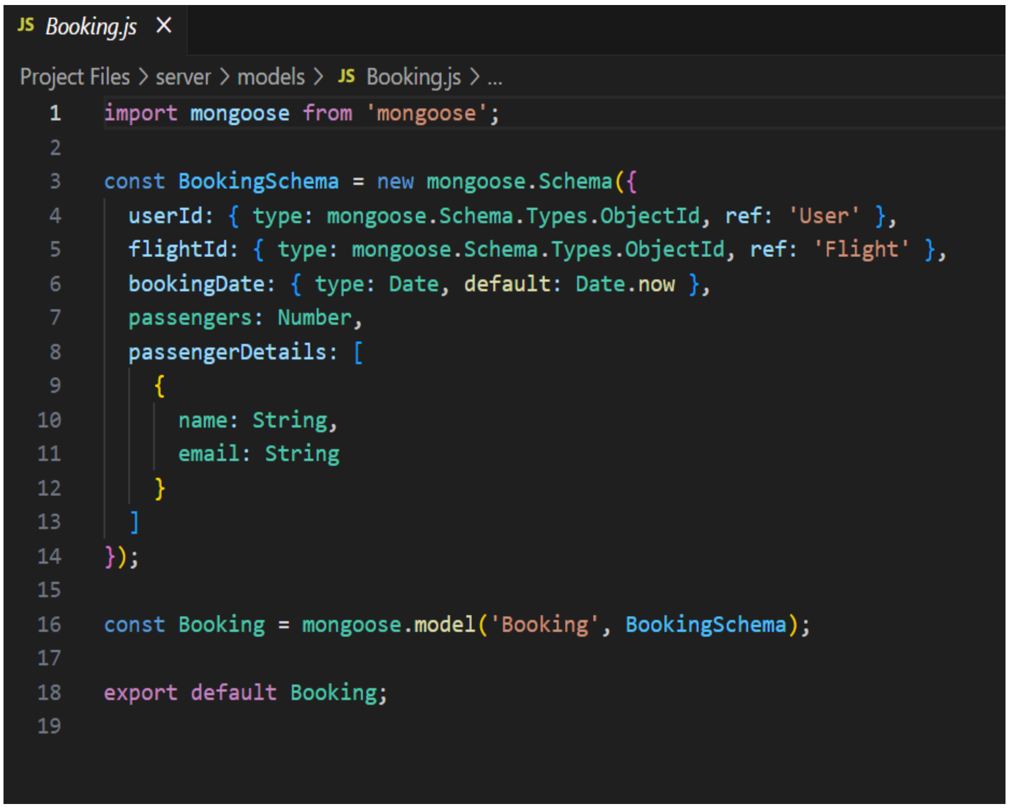
JS User.js M ×

Project Files > server > models > JS User.js > ...

```
import mongoose from 'mongoose';
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  email: {
    type: String,
    unique: true,
    required: true
  },
  password: {
    type: String,
    required: true
  },
  role: {
    type: String,
    enum: ['user', 'admin', 'flightoperator'],
    default: 'user'
  },
  isApproved: {
    type: Boolean,
    default: function() {
      return this.role === 'flightoperator' ? false : true;
    }
  }
});
const User = mongoose.model('User', userSchema);
export default User;
```



```javascript
import mongoose from 'mongoose';

const flightSchema = new mongoose.Schema({
  airline: String,
  flightNumber: String,
  departure: String,
  destination: String,
  departureTime: Date,
  arrivalTime: Date,
  price: Number,
  availableSeats: Number,
  createdBy: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  }
});

export default mongoose.model('Flight', flightSchema);
```

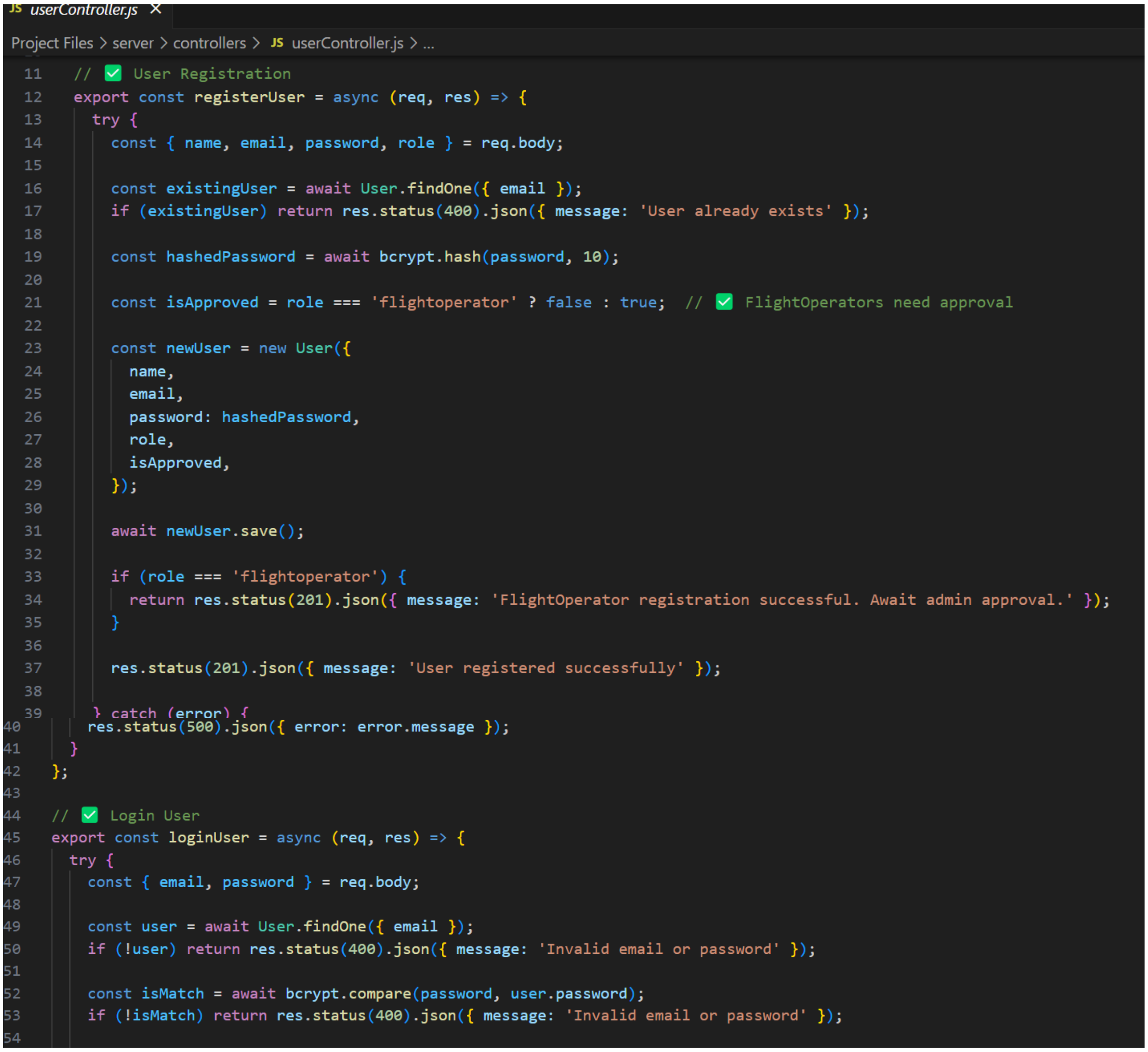


```javascript
import mongoose from 'mongoose';

const BookingSchema = new mongoose.Schema({
  userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  flightId: { type: mongoose.Schema.Types.ObjectId, ref: 'Flight' },
  bookingDate: { type: Date, default: Date.now },
  passengers: Number,
  passengerDetails: [
    {
      name: String,
      email: String
    }
  ]
});

const Booking = mongoose.model('Booking', BookingSchema);

export default Booking;
```

User Authentication:

- **Backend**

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

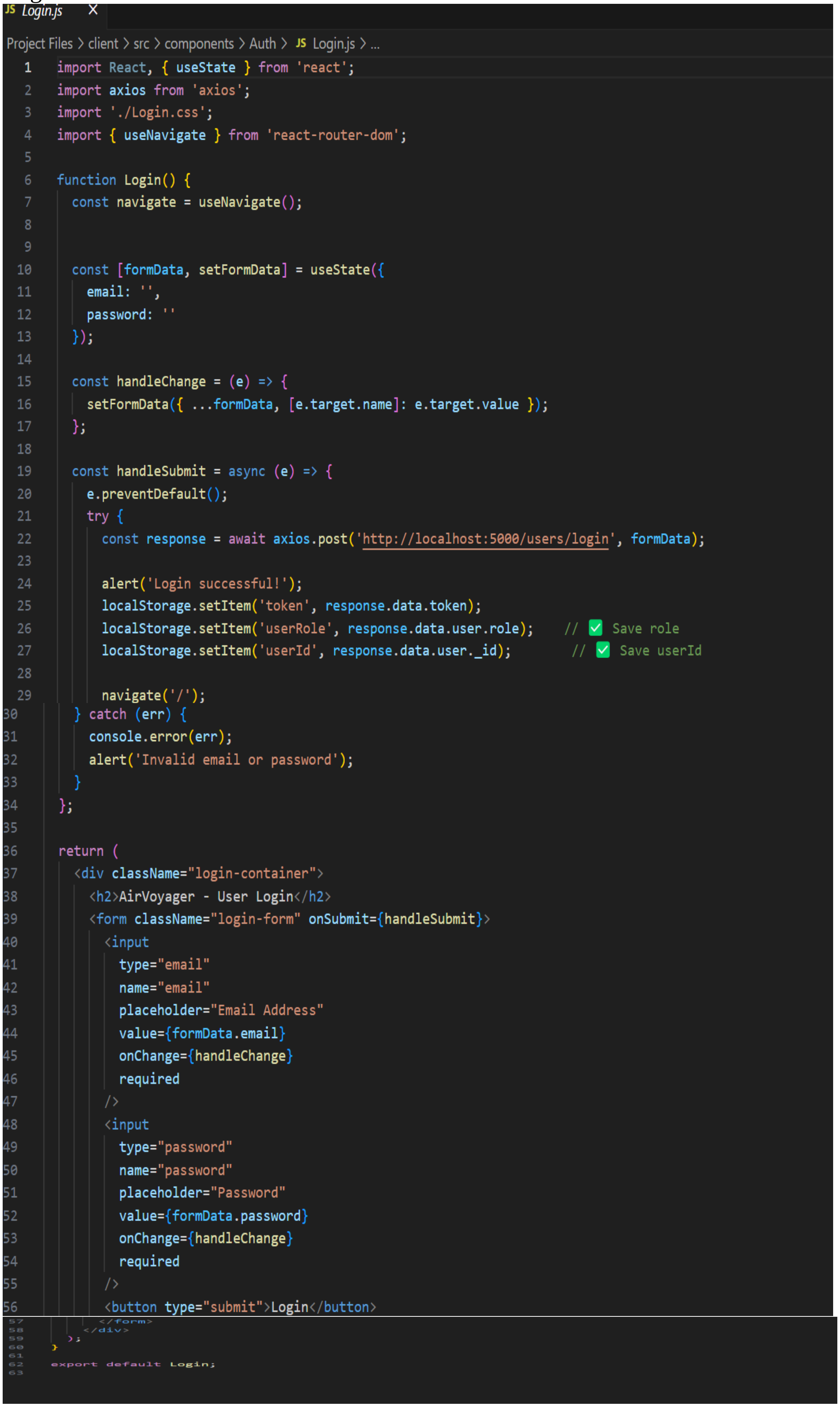


userController.js ×
Project Files > server > controllers > userController.js > ...

```javascript
// ✅ User Registration
export const registerUser = async (req, res) => {
  try {
    const { name, email, password, role } = req.body;

    const existingUser = await User.findOne({ email });
    if (existingUser) return res.status(400).json({ message: 'User already exists' });

    const hashedPassword = await bcrypt.hash(password, 10);

    const isApproved = role === 'flightoperator' ? false : true;  // ✅ FlightOperators need approval

    const newUser = new User({
      name,
      email,
      password: hashedPassword,
      role,
      isApproved,
    });

    await newUser.save();

    if (role === 'flightoperator') {
      return res.status(201).json({ message: 'FlightOperator registration successful. Await admin approval.' });
    }

    res.status(201).json({ message: 'User registered successfully' });

  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};

// ✅ Login User
export const loginUser = async (req, res) => {
  try {
    const { email, password } = req.body;

    const user = await User.findOne({ email });
    if (!user) return res.status(400).json({ message: 'Invalid email or password' });

    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.status(400).json({ message: 'Invalid email or password' });

```

- **Frontend**

Login:

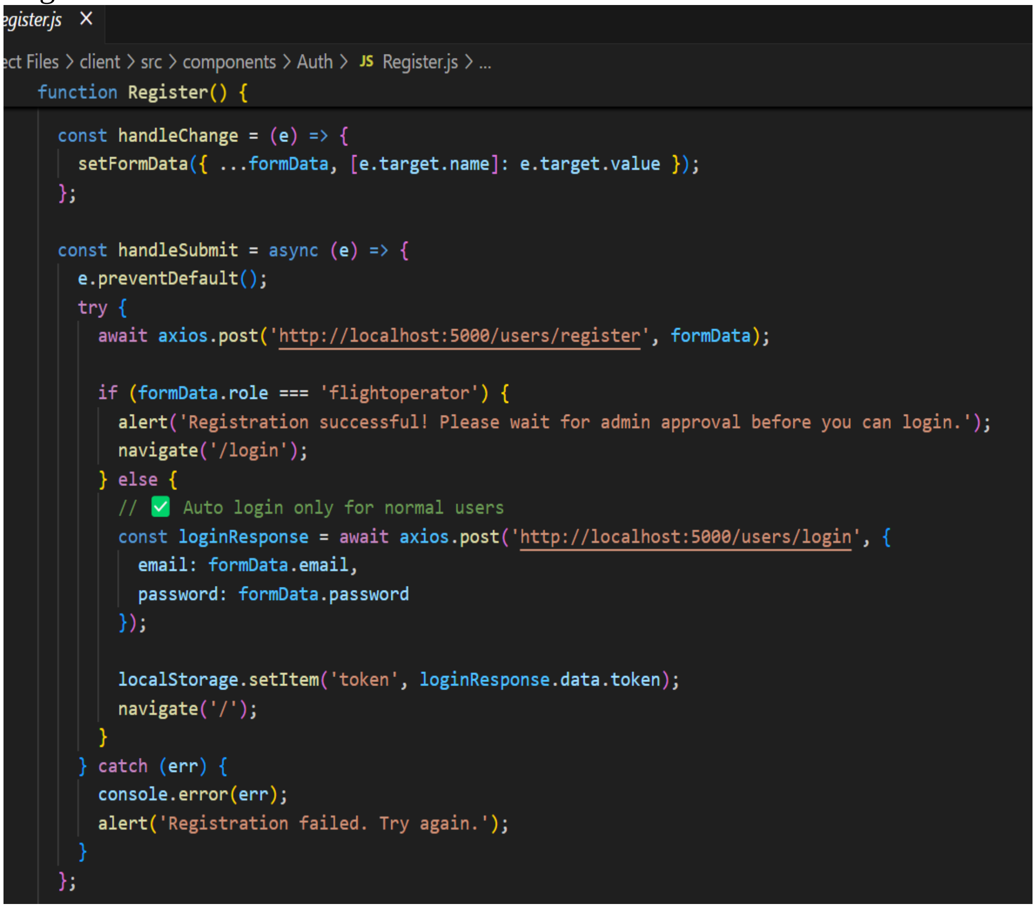


JS Login.js X

Project Files > client > src > components > Auth > JS Login.js > ...

```jsx
import React, { useState } from 'react';
import axios from 'axios';
import './Login.css';
import { useNavigate } from 'react-router-dom';

function Login() {
  const navigate = useNavigate();


  const [formData, setFormData] = useState({
    email: '',
    password: ''
  });

  const handleChange = (e) => {
    setFormData({ ...formData, [e.target.name]: e.target.value });
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      const response = await axios.post('http://localhost:5000/users/login', formData);

      alert('Login successful!');
      localStorage.setItem('token', response.data.token);
      localStorage.setItem('userRole', response.data.user.role);    // ✅ Save role
      localStorage.setItem('userId', response.data.user._id);        // ✅ Save userId

      navigate('/');
    } catch (err) {
      console.error(err);
      alert('Invalid email or password');
    }
  };

  return (
    <div className="login-container">
      <h2>AirVoyager - User Login</h2>
      <form className="login-form" onSubmit={handleSubmit}>
        <input
          type="email"
          name="email"
          placeholder="Email Address"
          value={formData.email}
          onChange={handleChange}
          required
        />
        <input
          type="password"
          name="password"
          placeholder="Password"
          value={formData.password}
          onChange={handleChange}
          required
        />
        <button type="submit">Login</button>
      </form>
    </div>
  );
}

export default Login;
```

Register:



egister.js ×

ect Files > client > src > components > Auth > JS Register.js > ...

```
function Register() {

  const handleChange = (e) => {
    setFormData({ ...formData, [e.target.name]: e.target.value });
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      await axios.post('http://localhost:5000/users/register', formData);

      if (formData.role === 'flightoperator') {
        alert('Registration successful! Please wait for admin approval before you can login.');
        navigate('/login');
      } else {
        // ✅ Auto login only for normal users
        const loginResponse = await axios.post('http://localhost:5000/users/login', {
          email: formData.email,
          password: formData.password
        });

        localStorage.setItem('token', loginResponse.data.token);
        navigate('/');
      }
    } catch (err) {
      console.error(err);
      alert('Registration failed. Try again.');
    }
  };
```

Flight Booking (User):

🖥 Frontend

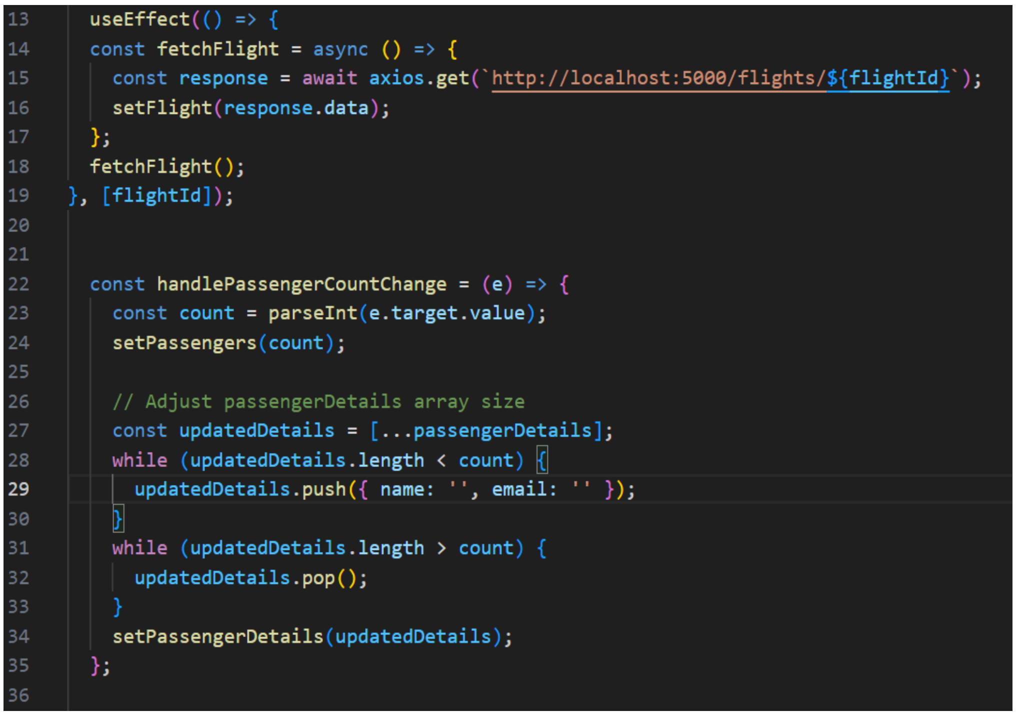
In the **frontend**, built with **React.js**, we designed a user-friendly interface for all key functionalities of the flight booking platform. The entire booking and flight browsing experience is seamless and intuitive.

🔍 Flight Search Functionality

On the **Home Page**, users are presented with a **flight search form**, which includes the following inputs:

- **Departure City**
- **Destination City**
- **Departure Date**

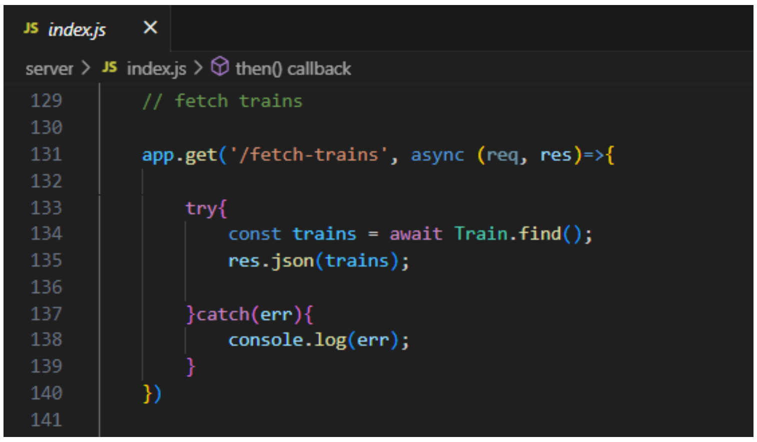
Once the user fills out the search form and clicks **"Search Flights"**, the app fetches all available flights matching the selected criteria from the backend.



```
  useEffect(() => {
  const fetchFlight = async () => {
    const response = await axios.get(`http://localhost:5000/flights/${flightId}`);
    setFlight(response.data);
  };
  fetchFlight();
}, [flightId]);


  const handlePassengerCountChange = (e) => {
    const count = parseInt(e.target.value);
    setPassengers(count);

    // Adjust passengerDetails array size
    const updatedDetails = [...passengerDetails];
    while (updatedDetails.length < count) {
      updatedDetails.push({ name: '', email: '' });
    }
    while (updatedDetails.length > count) {
      updatedDetails.pop();
    }
    setPassengerDetails(updatedDetails);
  };

```

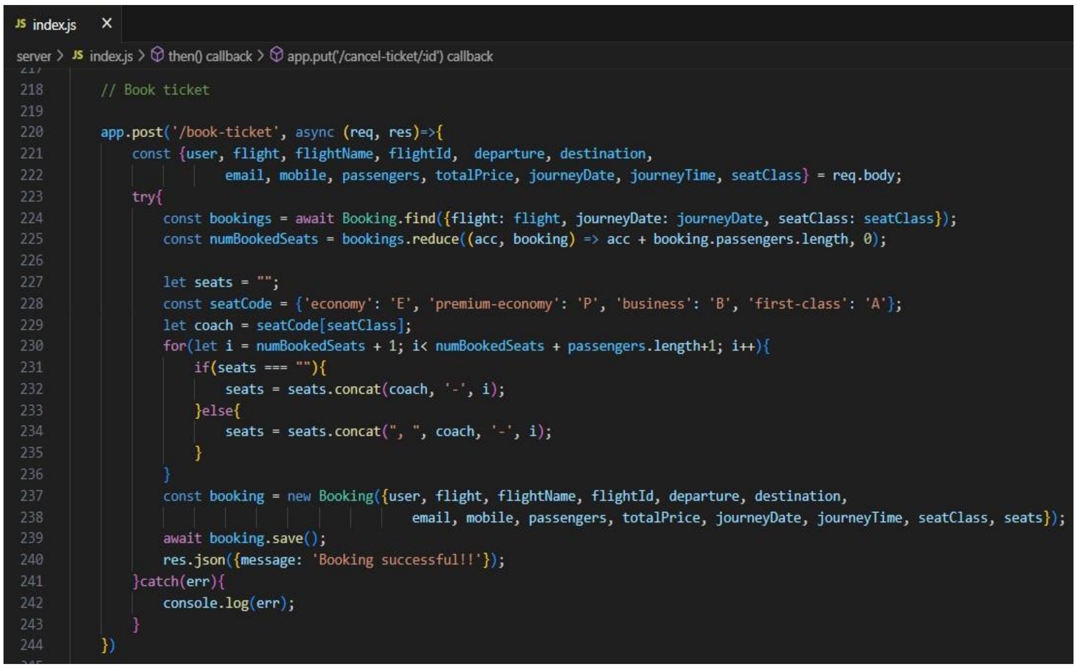
- **Backend**

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



```javascript
// 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.



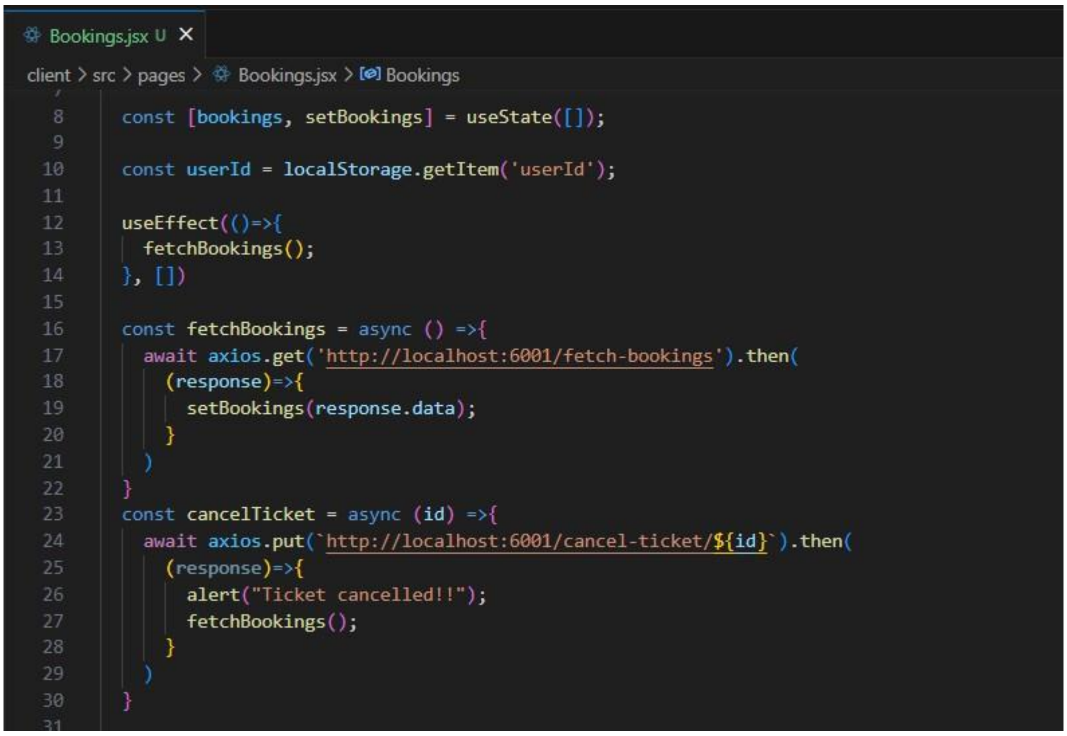
index.js
server > index.js > then() callback > app.put('/cancel-ticket/:id') callback

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

Fetching user bookings:

- **Frontend**

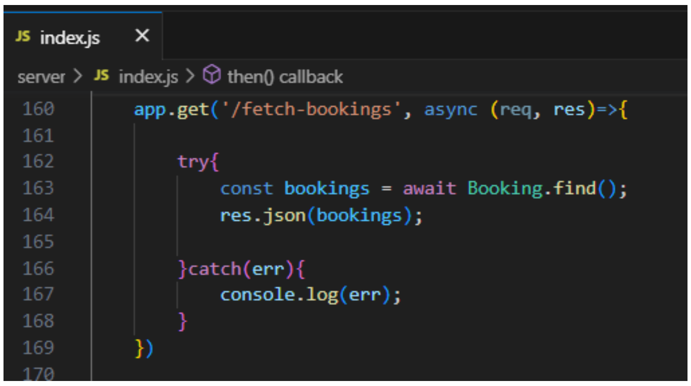
  In the bookings page, along with displaying the past bookings, we will also provide an option to cancel that booking.



```
    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();
        }
      )
    }
```

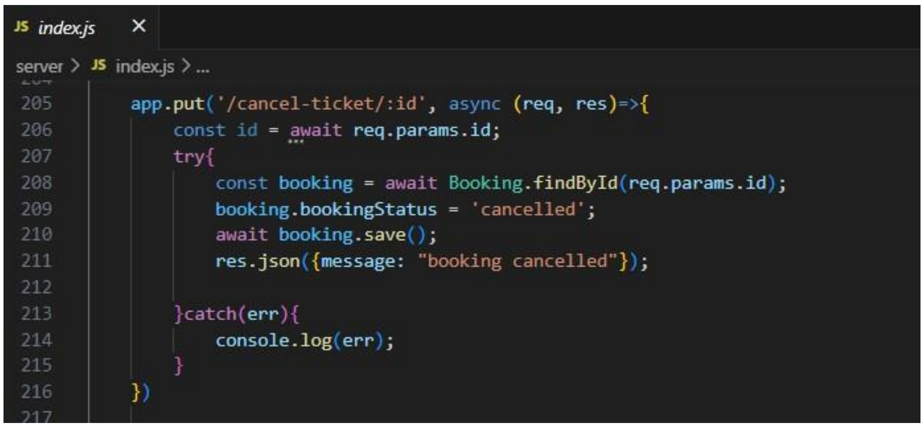
- **Backend**

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



```
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.



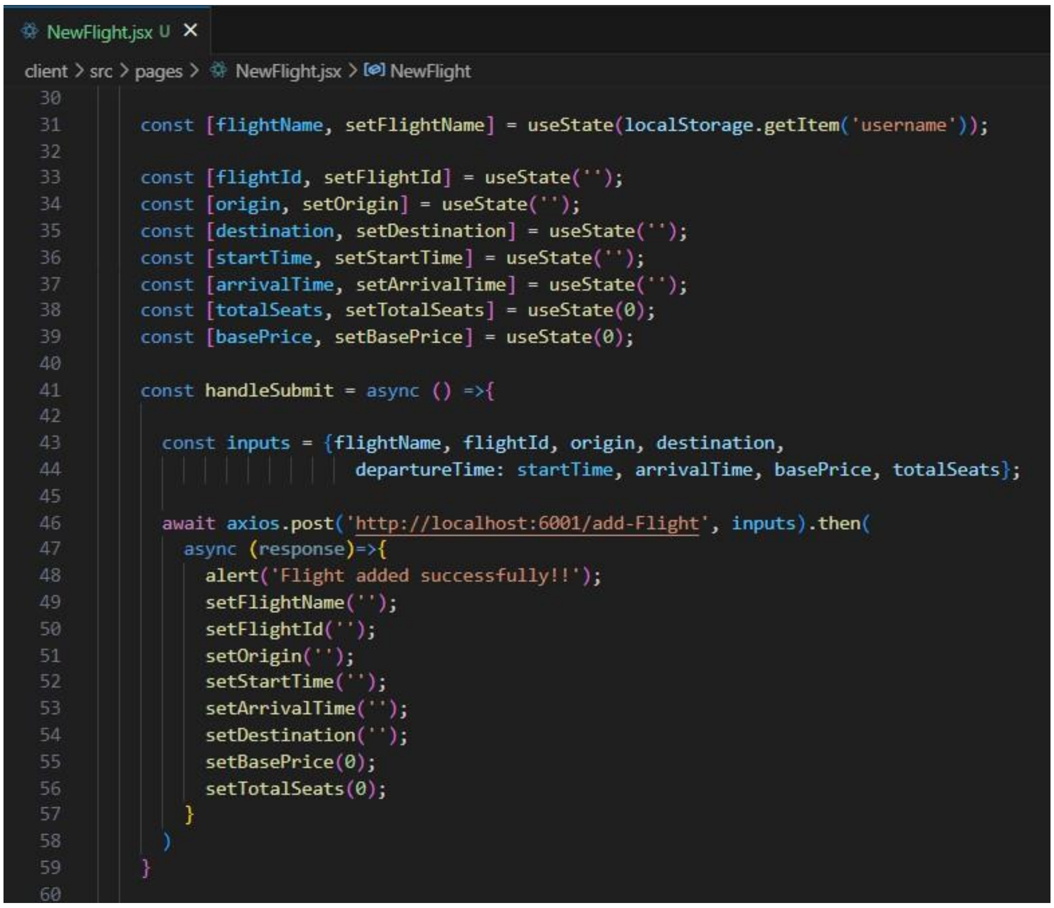
```
app.put('/cancel-ticket/:id', async (req, res)=>{
    const id = await req.params.id;
    try{
        const booking = await Booking.findById(req.params.id);
        booking.bookingStatus = 'cancelled';
        await booking.save();
        res.json({message: "booking cancelled"});

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

Add new flight:

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

- **Frontend**

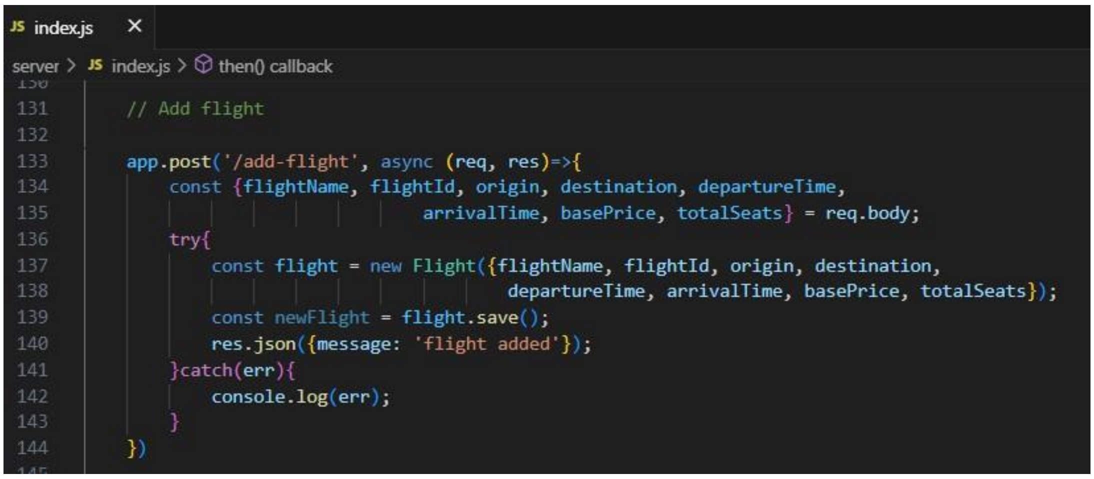
  We create a html form with required inputs for the new flight and then send an http request to the server to add it to database.



```jsx

    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.

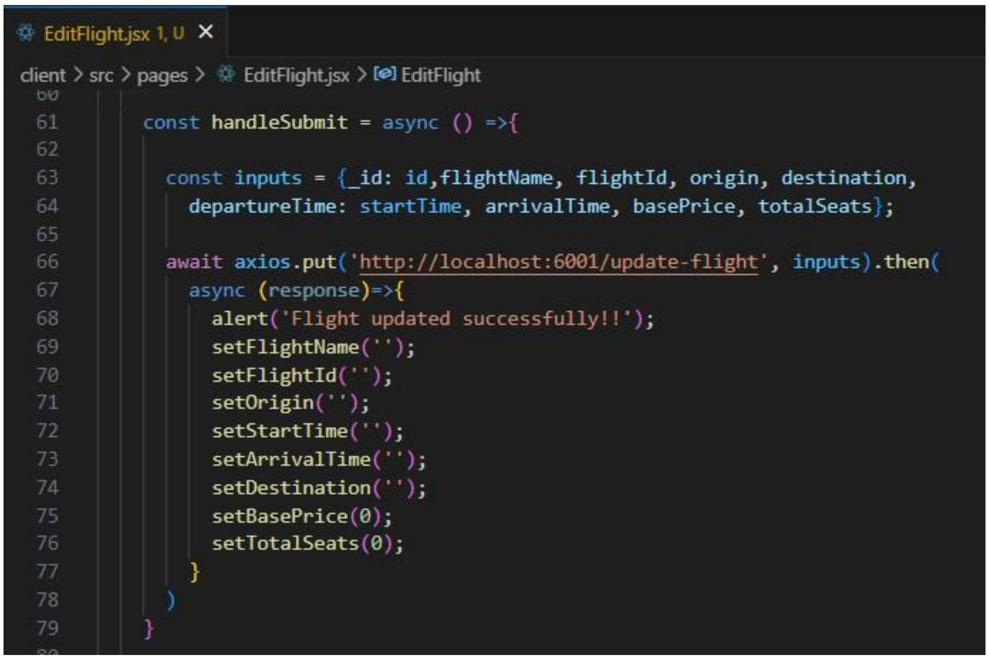


```js
    // Add flight

    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:

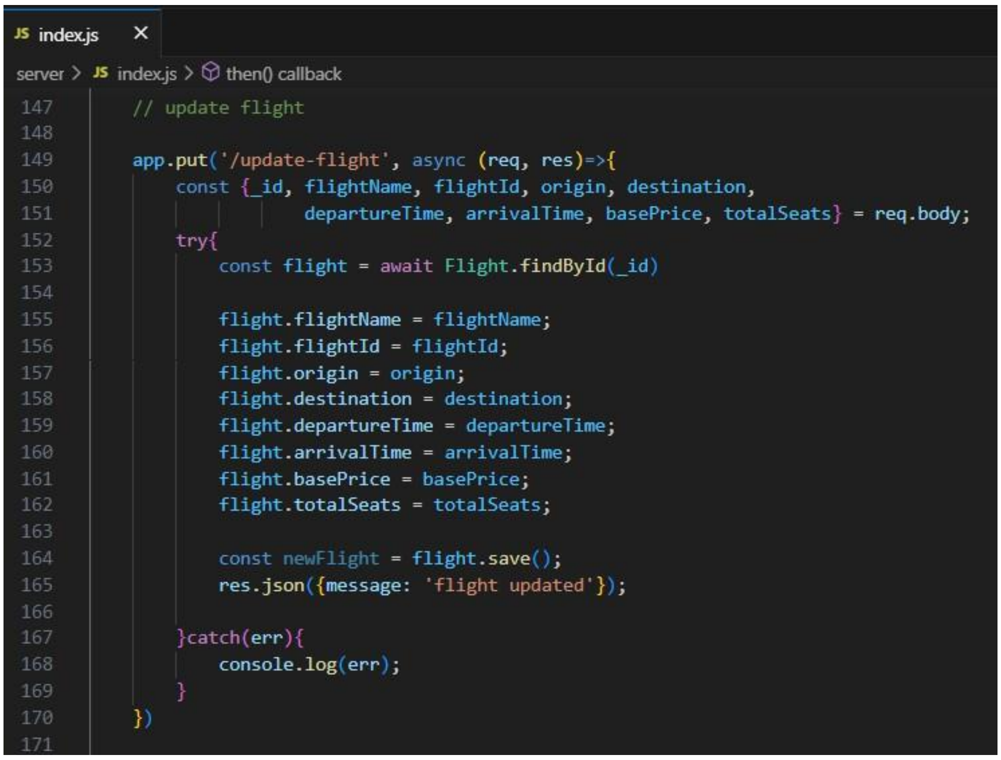
Here, in the admin dashboard, we will update the flight details in case if we want to make any edits to it

- **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:**

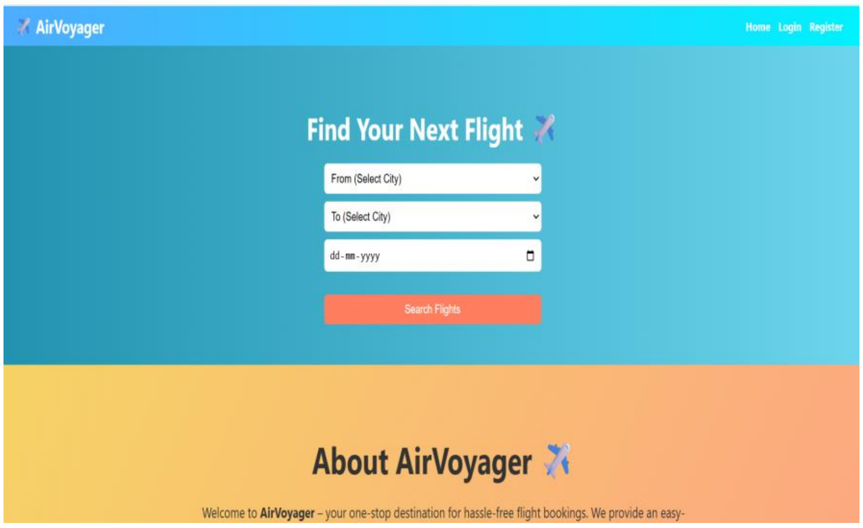


```
    // update flight

    app.put('/update-flight', async (req, res)=>{
        const {_id, flightName, flightId, origin, destination,
                    departureTime, arrivalTime, basePrice, totalSeats} = req.body;
        try{
            const flight = await Flight.findById(_id)

            flight.flightName = flightName;
            flight.flightId = flightId;
            flight.origin = origin;
            flight.destination = destination;
            flight.departureTime = departureTime;
            flight.arrivalTime = arrivalTime;
            flight.basePrice = basePrice;
            flight.totalSeats = totalSeats;

            const newFlight = flight.save();
            res.json({message: 'flight updated'});

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

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

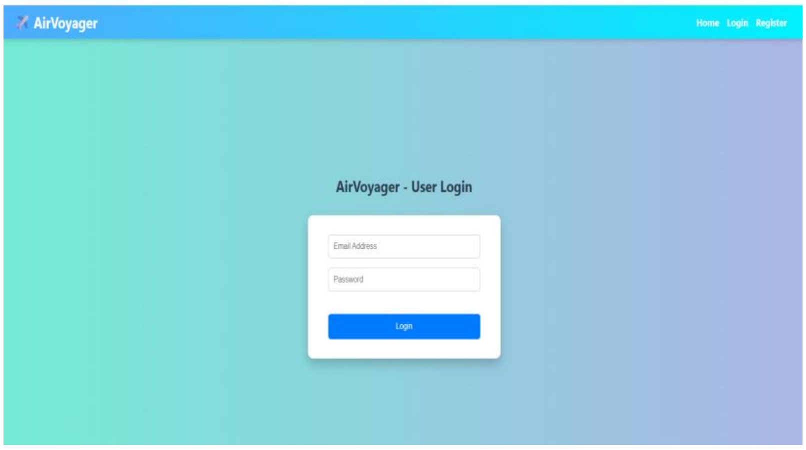
Demo UI images:

- **Landing page**

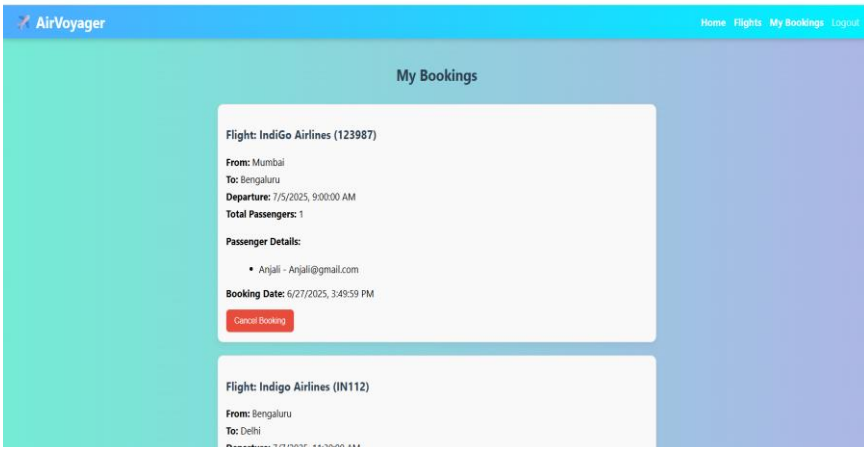


Welcome to **AirVoyager** – your one-stop destination for hassle-free flight bookings. We provide an easy-

- **Authentication**



- **User bookings**



My Bookings

Flight: IndiGo Airlines (123987)

From: Mumbai
To: Bengaluru
Departure: 7/5/2025, 9:00:00 AM
Total Passengers: 1

Passenger Details:

- Anjali - Anjali@gmail.com

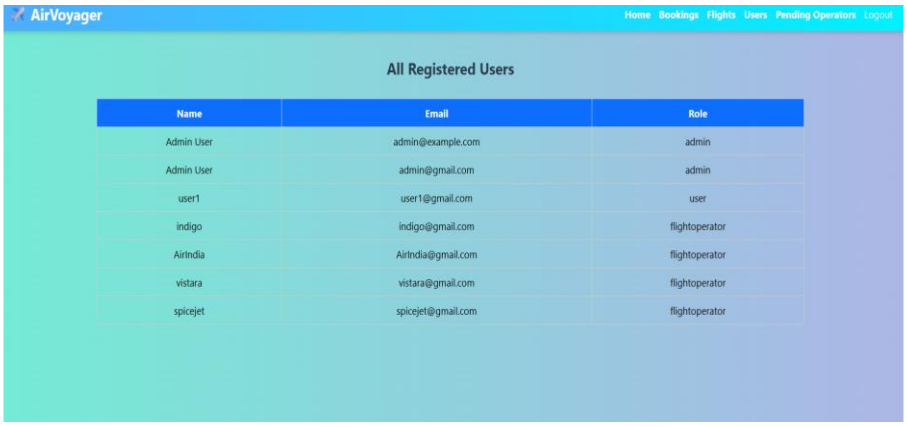
Booking Date: 6/27/2025, 3:49:59 PM

Flight: Indigo Airlines (IN112)

From: Bengaluru
To: Delhi

- **Admin Dashboard**

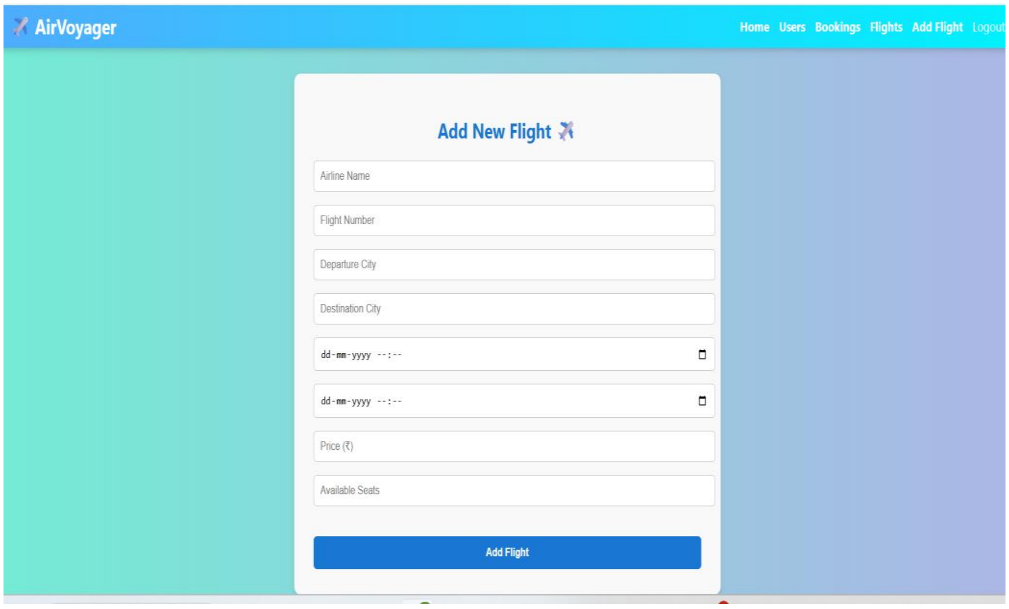
- **All users**



All Registered Users

Name	Email	Role
Admin User	admin@example.com	admin
Admin User	admin@gmail.com	admin
user1	user1@gmail.com	user
indigo	indigo@gmail.com	flightoperator
AirIndia	AirIndia@gmail.com	flightoperator
vistara	vistara@gmail.com	flightoperator
spicejet	spicejet@gmail.com	flightoperator

New Flight



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

https://github.com/Harshitha-2210/FlightFinder-Navigating-Your-Air-Travel-Options

The demo of the app is available at:

https://drive.google.com/file/d/1dZrSI2UkzNTE1j9HP9KjnNUYodqOOLKf/view?usp=sharing

** Happy Coding **