# BATCH

**1.Introduction**

**• Project Title:**

FlightFinder: Navigating your air travel options

**• Team Members:**

* Chadalawada Mahalakshmi – Team Leader,
* Bandla Nikhitha – Team Member,
* Balireddy Raghava Priya – Team Member,
* Battula Akhila– Team Member.

**2. Project Overview**

**• Purpose:**

The **FlightFinder App** is a full-stack web platform designed to simplify the process of searching, booking, and managing flights. Users can easily find available flights by entering their departure and destination cities along with the travel date. The app displays flight details such as price, timing, and duration, and allows users to book directly through a user-friendly modal interface.

In addition to passengers, the app provides **flight operators** with a dedicated dashboard to add, update, or remove flights and manage bookings efficiently. Role-based access ensures security and organized workflows.

**Administrators** have complete control over the system, allowing them to manage users, monitor platform activity, and analyze usage statistics. The app ensures data accuracy, system integrity, and a responsive interface across devices.

Overall, FlightFinder aims to offer a smooth, secure, and scalable solution for travelers, flight operators, and administrators, making it a **comprehensive tool for modern flight management**.

**• Features:**

**User Features (Passengers)**

* **Flight Search**Users can search flights by entering the departure city, destination city, and date of travel.
* **Flight Details View**  
  Displays key flight information such as airline name, departure/arrival time, duration, and price.
* **Easy Booking System**  
  Allows users to book flights using a clean and simple modal-based interface.
* **Booking History**  
  Users can view their past and upcoming flight bookings in one place.
* **Login/Register**  
  Secure login and registration system with form validation and role-based redirection.

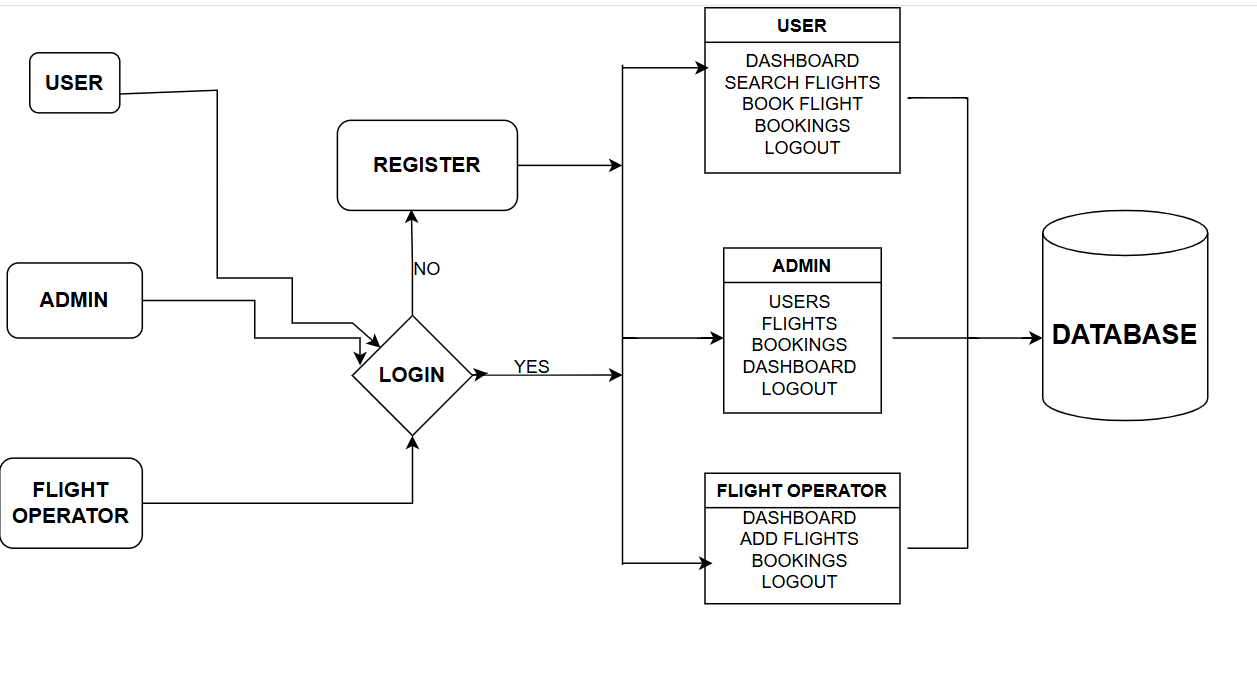
**Operator Features**

* **Add/Edit/Delete Flights**  
  Operators can manage their flight listings by adding new flights or modifying existing ones.
* **View Bookings**  
  Operators can see booking details for their flights, including passenger information.
* **Real-Time Flight Management**  
  Allows operators to track capacity and update flight statuses if needed.
* **Role-Based Access**  
  Operators only access functionalities relevant to their role, enhancing security.

**Admin Features**

* **User and Operator Management**  
  Admins can add or remove users and flight operators as needed.
* **System Monitoring**  
  View total number of users, bookings, and flights to oversee platform activity.
* **Statistics and Reports**  
  Access analytical data on usage, peak times, and user behavior.
* **Platform Integrity Control**  
  Ensure the app functions securely and efficiently across all roles**.**

**3. Architecture**



**Fig: - Architecture**

**Frontend (React.js)**

The frontend is responsible for the visual representation and interaction of the system. It would include:

1. **Login/Register**

* Create a Component which contains a form for taking the username and password.
* If the given inputs matches the data of user or admin or flight operator then navigate it to their respective home page

1. **Flight Booking (User):**

* 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 redirects to the flight-Booking page.

1. **Fetching user bookings:**

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

1. **Add new flight(Admin):**

* Now, in the admin dashboard, we provide functionality to add new flights.
* We create a html form with required inputs for the new flight and then send an httprequest to the server to add it to the database.

1. **Update Flight:**

* Here, in the admin dashboard, we will update the flight details in case if we want to make any edits to it
* Along with this, implement additional features to view all flights, bookings, and users in the admin dashboard.

**Backend (Express.js & Node.js)**

The backend handles the server-side operations, including:

1. **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.

1. **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.

1. **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.

1. **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.

1. **User Authentication:**

* Create routes and middleware for user registration, login, and logout.
* Set up authentication middleware to protect routes that require user authentication.

1. **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.

1. **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.

1. **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.

**Database (MongoDB)**

The database stores the persistent data of the system. It would typically include:

* **Configure schema**

Firstly, configure the Schemas for MongoDB database, to store the data in such a pattern. Use the data from the ER diagrams to create the schemas. The Schemas for this application look alike to the one provided below.

* **Connect database to backend**

Now, make sure the database is connected before performing any of the actions through the backend. The connection code looks similar to the one provided below.

**4. Setup Instructions**

**• Prerequisites:**

* **Node.js and npm:**Node.js is a powerful JavaScript runtime environment that allows you to run JavaScript code on the server side. It provides a scalable and efficient platform for building network applications.
* **Express.js:**  Express.js is a fast and minimalist web application framework for Node.js. It simplifies the process of creating robust APIs and web applications, offering features like routing, middleware support, and modular architecture.
* **MongoDB:**MongoDB is a flexible and scalable NoSQL database that stores data in a JSON-like format. It provides high performance, horizontal scalability, and seamless integration with Node.js, making it ideal for handling large amounts of structured and unstructured data.
* **Moment.js:**Momentjs is a JavaScript package that makes it simple to parse, validate, manipulate, and display date/time in JavaScript. Moment.js allows you to display dates in a human-readable format based on your location. Install React.js, a JavaScript library for building user interfaces
* **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.

**• Installation:**

* React
* Bootstrap
* Material UI
* Axios
* react-bootstrap
* bcryptjs
* express
* dotenv
* mongoose
* Nodemon
* Jsonwebtoken

**5. Folder Structure**

**•Client:**

* **public**: Contains static files like index.html and favicon.ico.
* **src**: Main source folder.
  + **assets**: Stores images, styles, and other static assets.
  + **components**: Reusable UI components.
  + **pages**: Different pages of the application.
  + **services**: API calls and other services.
  + **App.js**: Main application component.
  + **index.js**: Entry point of the React application.
* **package.json**: Lists dependencies and scripts.
* **.env**: Environment variables.
* **README.md**: Project documentation.

**• Server:**

* **config**: Configuration files, such as database and application settings.
* **controllers**: Logic for handling requests and responses.
* **models**: Database schemas and models.
* **routes**: Route definitions for the API endpoints.
* **middlewares**: Custom middleware functions.
* **utils**: Utility functions and helpers.
* **server.js**: Entry point of the Node.js application.
* **package.json**: Lists dependencies and scripts.
* **.env**: Environment variables.
* **README.md**: Project documentation.

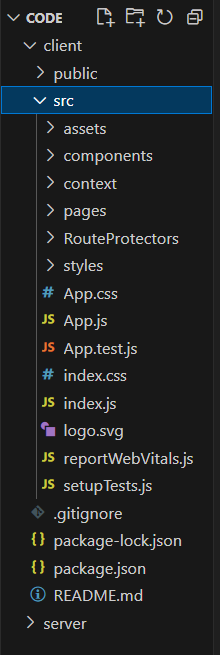
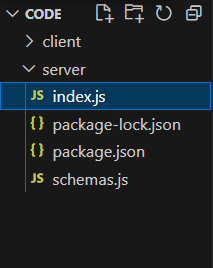
 

Fig: - Frontend Fig: - Backend

**6. Running the Application**

**Frontend:** npm start -in the client directory  **Backend:** npm run debug- in the server directory.

**7. API Documentation**

**Authentication: -**

**POST /api/auth/login** — Authenticate User

**Request Body:**

{

"email": "user@example.com",

"password": "test123"

}

**Response:**

{

"success": true,

"data": {

"token": "JWT\_TOKEN\_HERE"

},

"message": "Successfully Logged In"

}

**POST /api/auth/register —** Register New User

**Request Body:**

{

"firstname": "Priya",

"lastname": "Balireddy",

"email": "priya@example.com",

"password": "test123",

"phonenumber": "1234567890",

"role": "user"

}

**Response:**

{

"success": true,

"data": {

"token": "JWT\_TOKEN\_HERE"

},

"message": "User Successfully Registered"

}

**Flight Management: -**

**GET /api/flights/ —** Get All Flights

**Response:**

{

"success": true,

"data": [ { flight details... } ],

"message": "Flights fetched successfully"

}

**GET /api/flights/:id —** Get Flight by ID

**Response:**

{

"success": true,

"data": { flight details },

"message": "Flight fetched successfully"

}

**POST /api/flights/add —** Add New Flight *(Operator Only)*

**Request:**

{

"flightNumber": "AI202",

"source": "Delhi",

"destination": "Mumbai",

"departureTime": "2025-06-30T10:00:00",

"arrivalTime": "2025-06-30T12:00:00",

"price": 4000

}

**Response:**

{

"success": true,

"message": "Flight added successfully"

}

**PUT /api/flights/:id —** Update Flight

**Request Body:**

{

"flightNumber": "AI203",

"source": "Kolkata",

"destination": "Mumbai",

"departureTime": "2025-07-01T09:00:00",

"arrivalTime": "2025-07-01T11:30:00",

"price": 3800

}

**Response:**

{

"success": true,

"data": {

"id": "6648e212e2f93d16b58f1c35",

"flightNumber": "AI203",

"source": "Kolkata",

"destination": "Mumbai",

"departureTime": "2025-07-01T09:00:00.000Z",

"arrivalTime": "2025-07-01T11:30:00.000Z",

"price": 3800

},

"message": "Flight updated successfully"

}

**DELETE /api/flights/:id —** Delete Flight

DELETE /api/flights/6648e212e2f93d16b58f1c35

**Response:**

{

"success": true,

"message": "Flight deleted successfully"

}

**Booking Management:**

**POST /api/bookings/book —** Book a Flight

**Request:**

{

"userId": "USER\_ID",

"flightId": "FLIGHT\_ID"

}

**Response:**

{

"success": true,

"data": {

"bookingId": "BOOKING\_ID",

"flightId": "FLIGHT\_ID",

"userId": "USER\_ID",

"status": "booked"

},

"message": "Flight booked successfully"

}

**GET /api/bookings/:id —** Get Booking by ID

**Request:**

GET /api/users/6657d5c13c3b7c001fb6d9e1

**Response:**

{

"success": true,

"data": {

"id": "6657d5c13c3b7c001fb6d9e1",

"firstname": "Priya",

"lastname": "Balireddy",

"email": "priya@example.com",

"phonenumber": "1234567890",

"role": "user",

"active": true,

"createdAt": "2025-06-20T10:30:00.000Z",

"updatedAt": "2025-06-23T12:00:00.000Z"

},

"message": "User fetched successfully"

}

**GET /api/bookings/user/:userId —** Get Bookings of a User

**Response:**

{

"success": true,

"data": [ { booking details... } ],

"message": "Bookings fetched successfully"

}

**DELETE /api/bookings/:id — Cancel Booking**

**Response:**

{

"success": true,

"data": {},

"message": "Booking cancelled successfully"

}

**User Management**

**GET /api/users/ —** *Get All Registered Users*

**Request:**

GET /api/users/

**Response:**

{

"success": true,

"data": [

{

"id": "685a8ab8f81c27b0ca2a0364",

"username": "preethi",

"email": "preethi@gmail.com",

"usertype": "admin",

"approval": "approved"

},

{

"id": "685aa6b6fb037ea64428de74",

"username": "sruthi",

"email": "sruthi@gmail.com",

"usertype": "flight-operator",

"approval": "approved"

},

{

"id": "685aa695fb037ea64428de71",

"username": "keerthi",

"email": "keerthi@gmail.com",

"usertype": "customer",

"approval": "approved"

}

],

"message": "Fetched all users successfully"

}

**GET /api/users/:id —** *Get Single User by ID*

**Request:**

GET /api/users/6657d5c13c3b7c001fb6d9e1

**Response:**

{

"success": true,

"data": {

"id": "6657d5c13c3b7c001fb6d9e1",

"firstname": "Priya",

"lastname": "Balireddy",

"email": "priya@example.com",

"phonenumber": "1234567890",

"role": "user",

"active": true,

"createdAt": "2025-06-20T10:30:00.000Z",

"updatedAt": "2025-06-23T12:00:00.000Z"

},

"message": "User fetched successfully"

}

**API Request/Response Formats**

- Request: JSON

- Response: JSON

**API Error Handling**

- Error Codes: 400, 404, 500

- Error Messages: Detailed error messages

**8. Authentication**

Explain how authentication and authorization are handled in the project.

Include details about tokens, sessions, or any other methods used.

• **Authentication:**

* User Registration: Users register with email and password.
* Password Hashing: Passwords are hashed using BCrypt.
* Login: Users login with email and password.
* JSON Web Tokens (JWT): Upon successful login, a JWT is generated and sent to the client.

• **Authorization:**

* Role-Based Access Control (RBAC): Users are assigned roles (admin, doctor, patient).
* Token Verification: JWT is verified on each request to ensure authenticity.
* Route Protection: Routes are protected based on user roles.

• **Token Management:**

* Token Generation: JWT is generated using user ID, role, and expiration time.
* Token Storage: JWT is stored in local storage on the client-side.
* Token Validation: JWT is validated on each request using middleware.

**9. User Interface**

• Provide screenshots or GIFs showcasing different UI features.

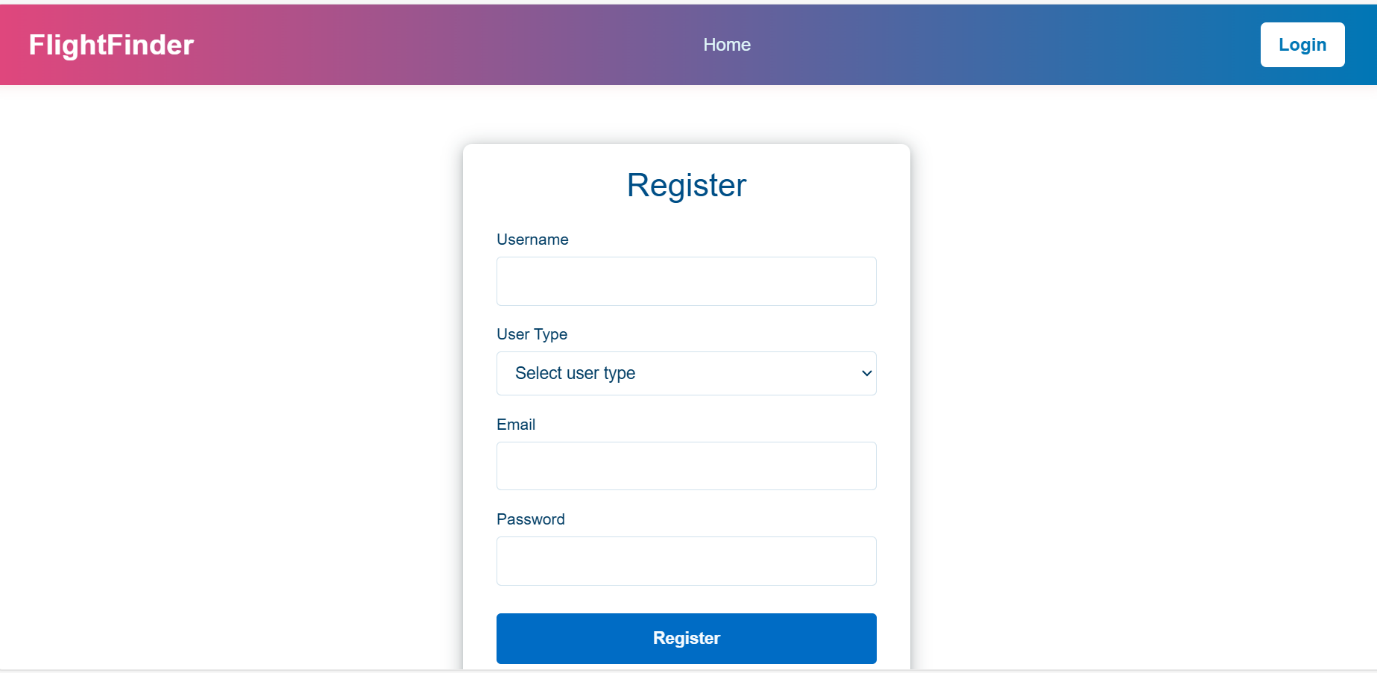


Fig: - Register Form

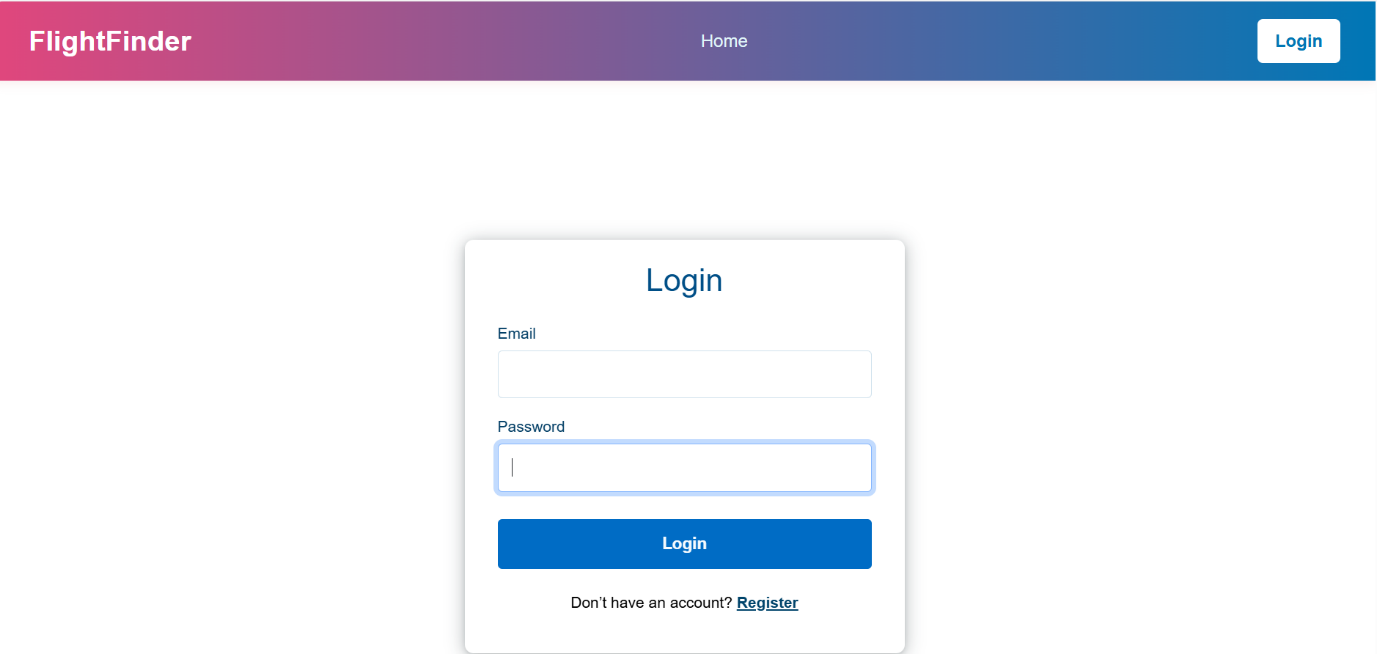


Fig: - Login Page

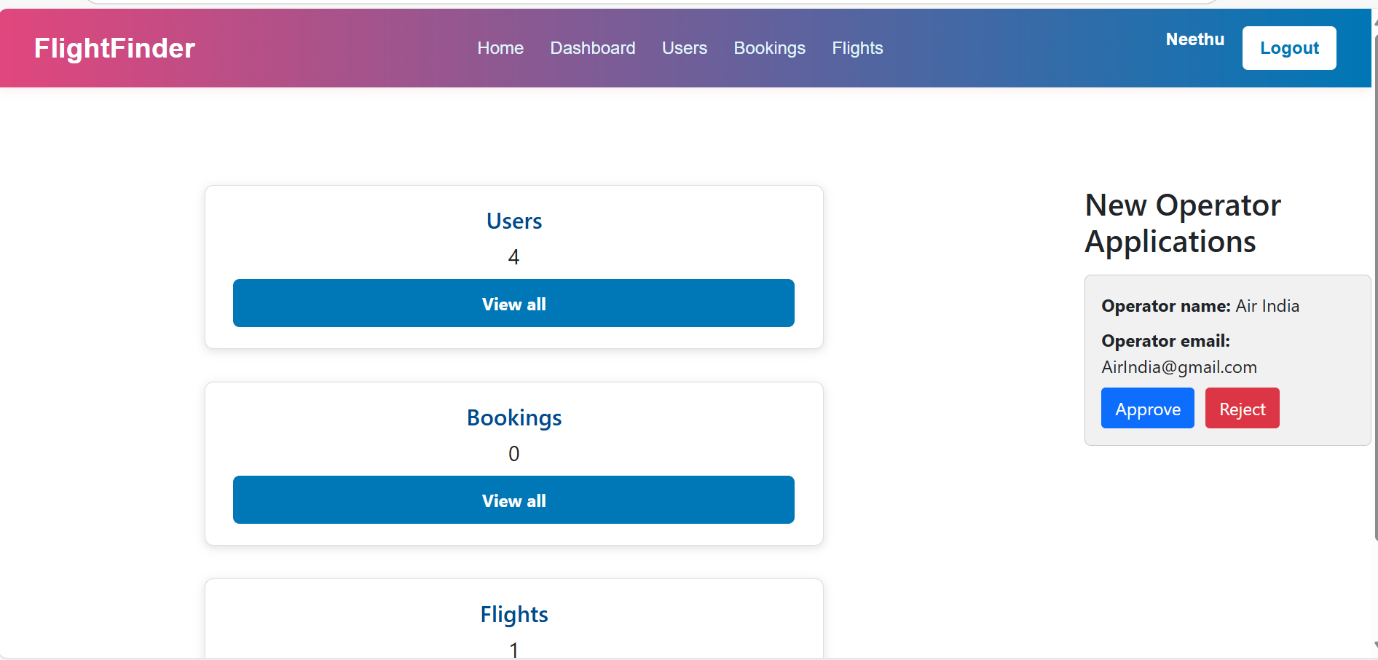
****

Fig: - Admin Interface

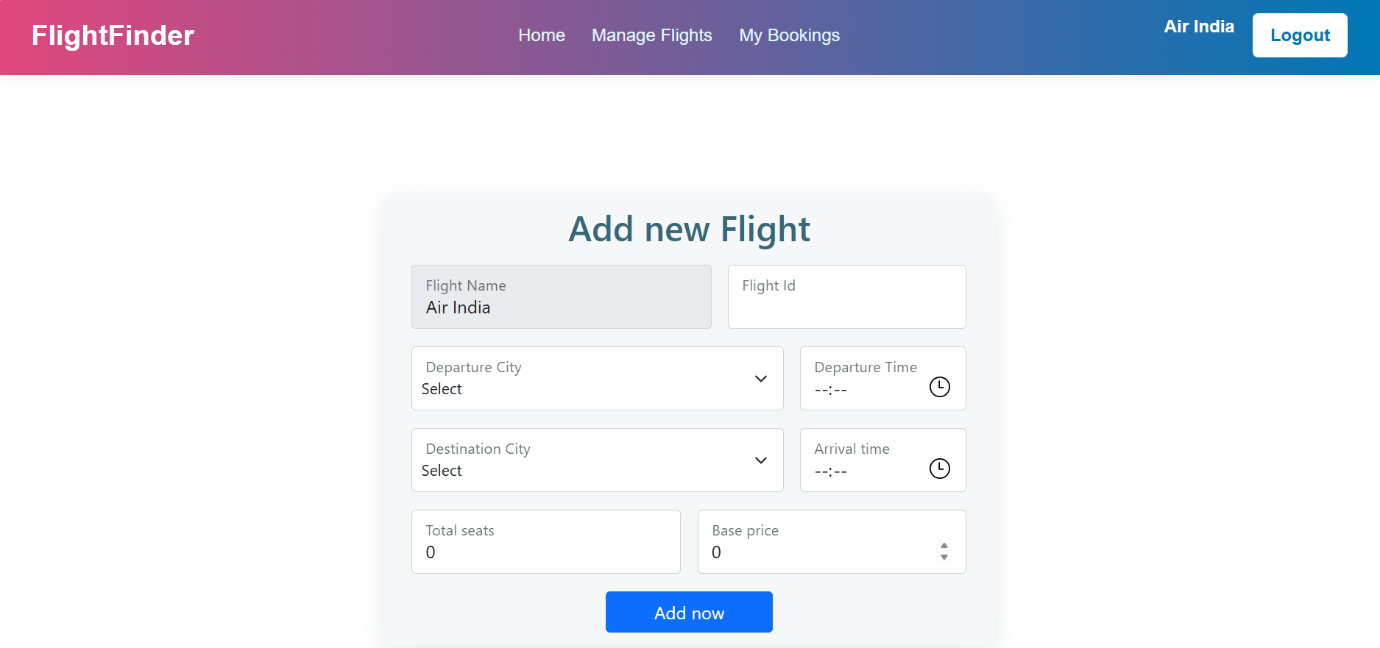
****

Fig: - FlightOperator Interface

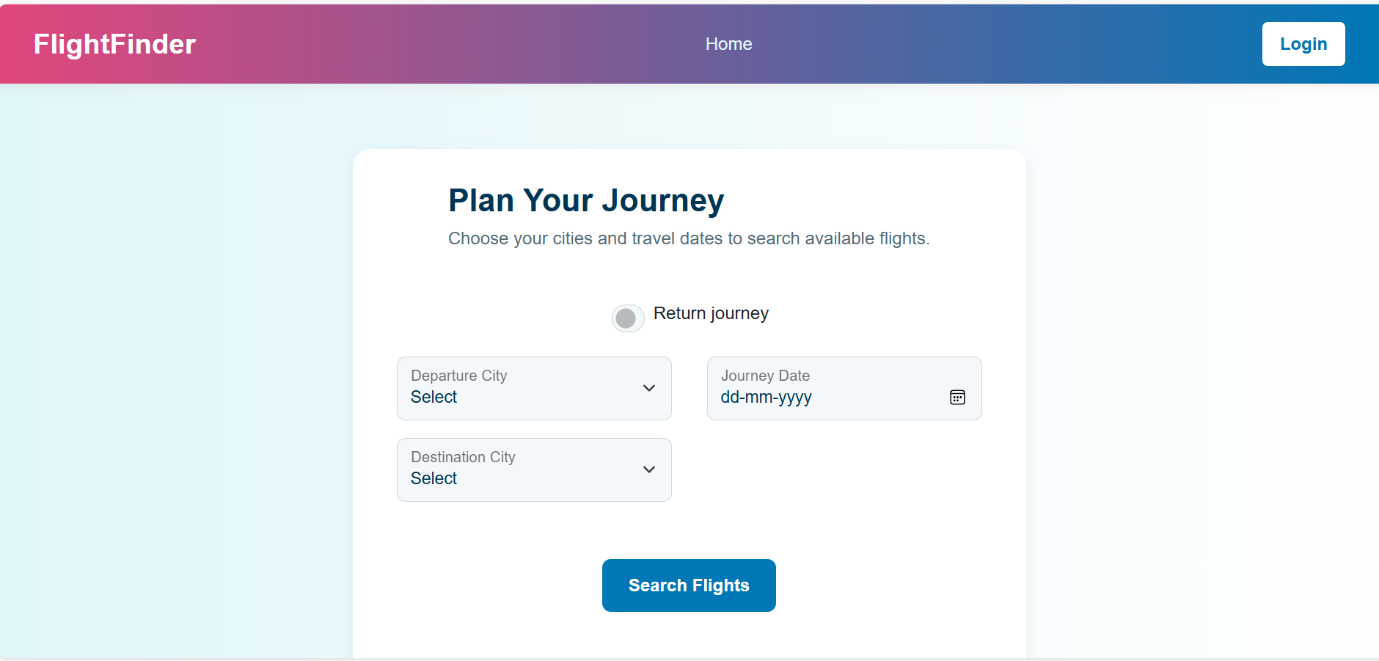
****

Fig: - User Interface

**10. Testing:**  Describe the testing strategy and tools used.

* **Unit Testing:** Test individual components (functions, classes) to ensure correct behaviour.
* **Integration Testing:** Verify interactions between components.
* **End-to-End (E2E) Testing:** Simulate user interactions to test entire workflows.
* **Acceptance Testing:** Validate app meets requirements and user expectations.

**11. Screenshots or Demo**

**ADMIN**

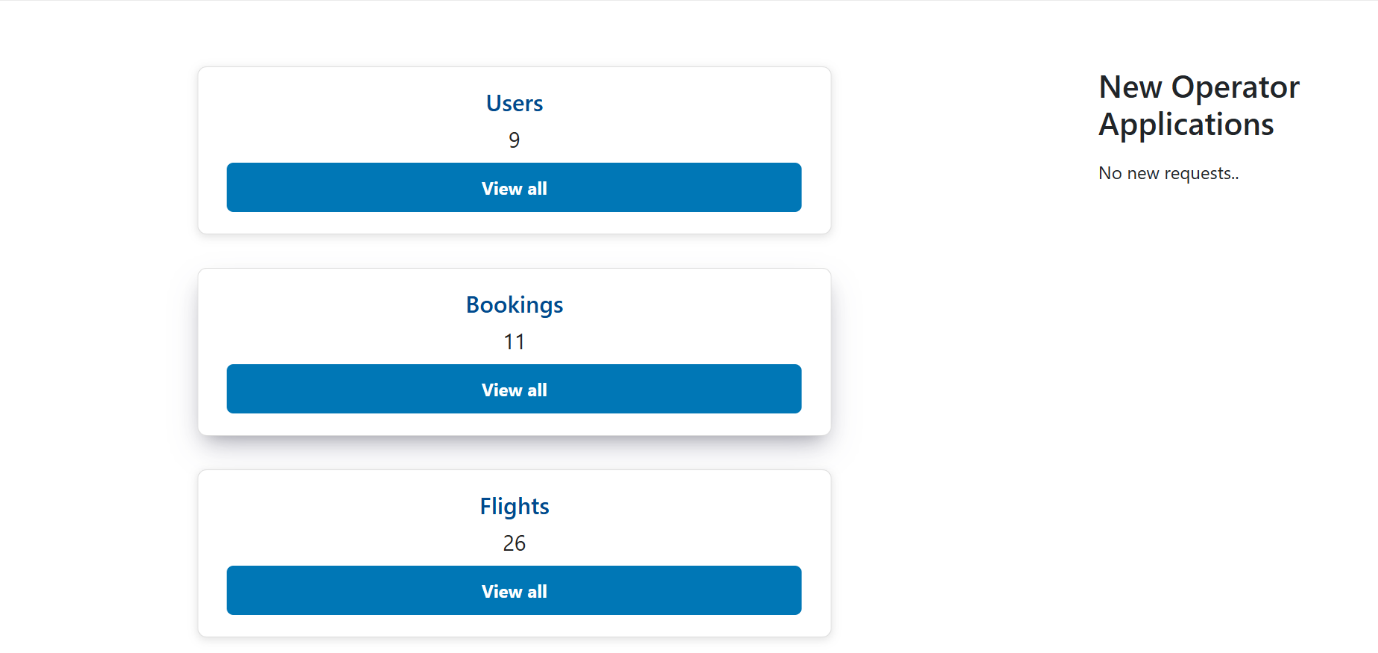


Fig: - Admin Dashboard

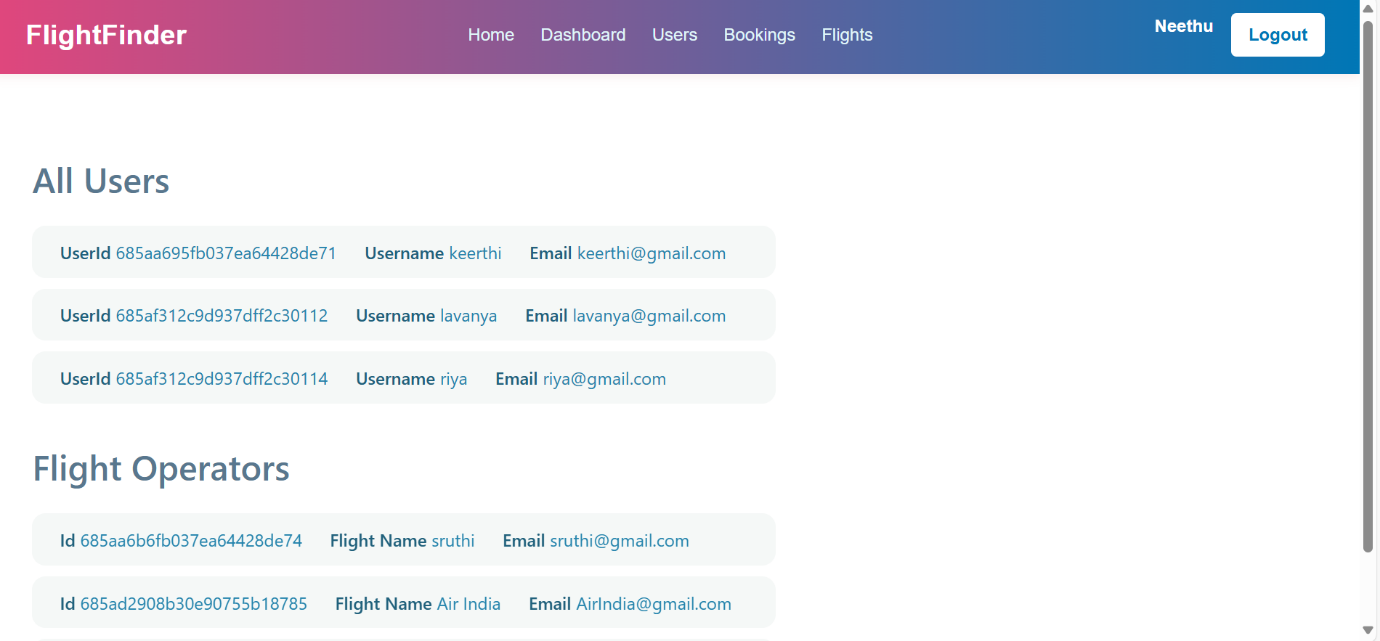


Fig: -Users

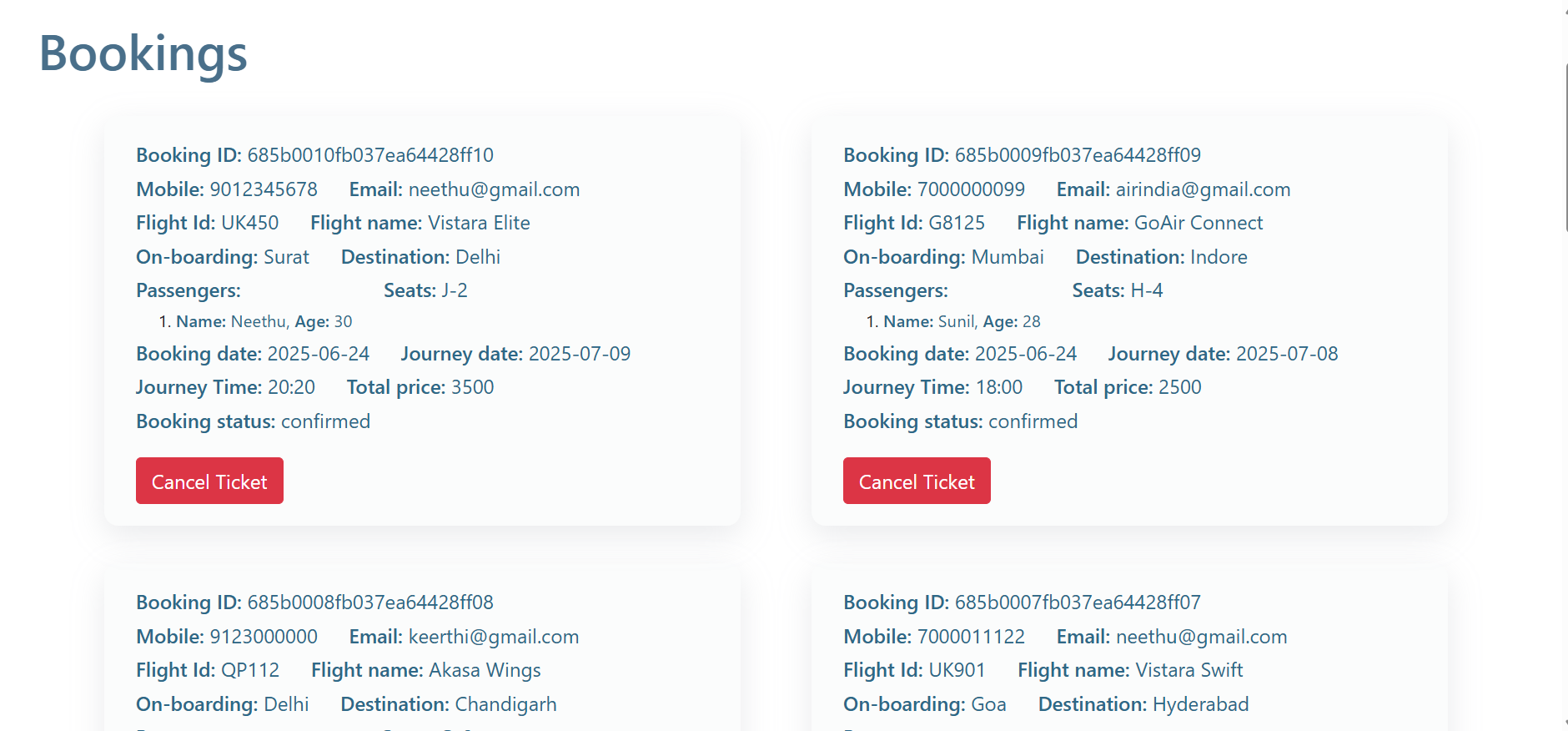


Fig: - Bookings

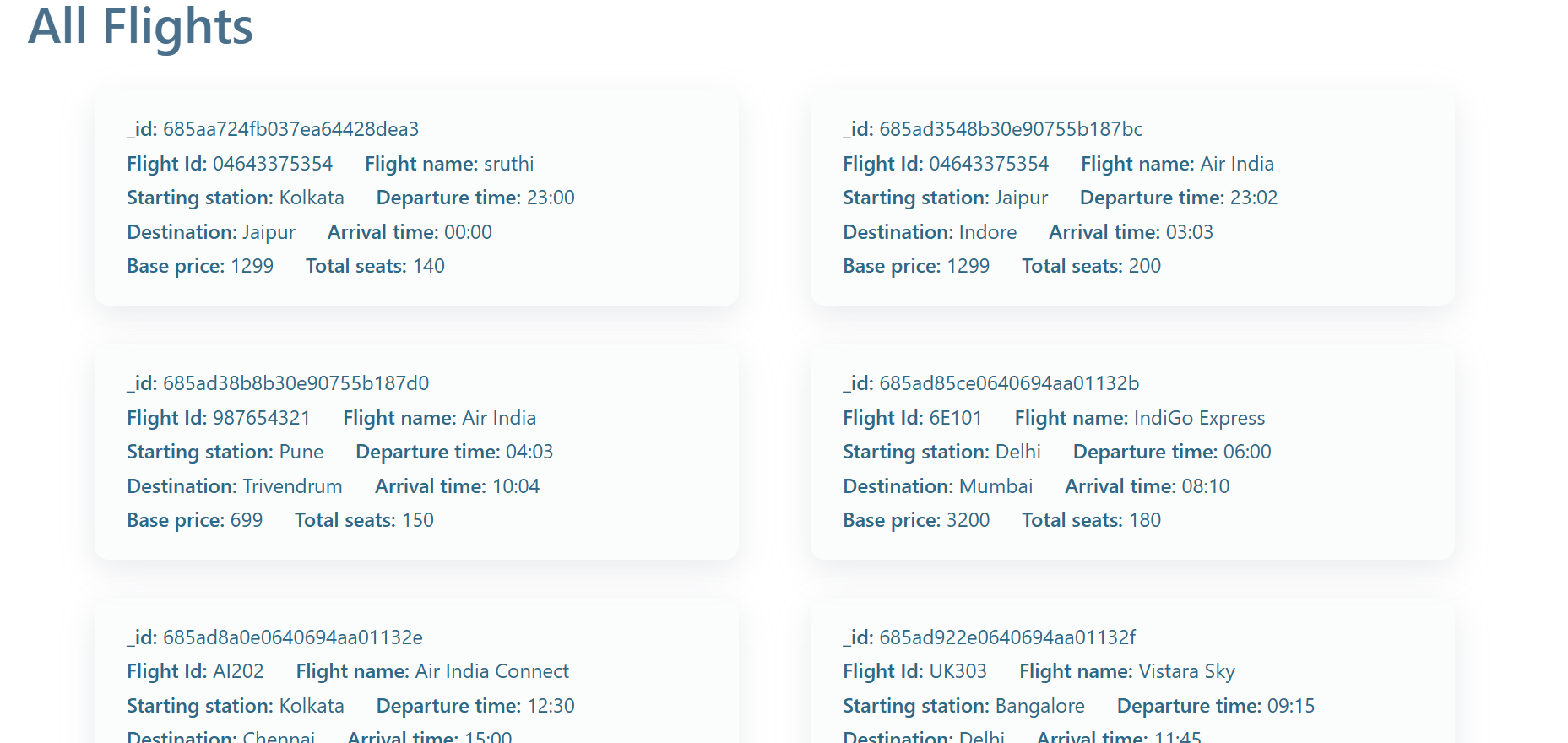


Fig: - Flights

**FLIGHT OPERATOR**

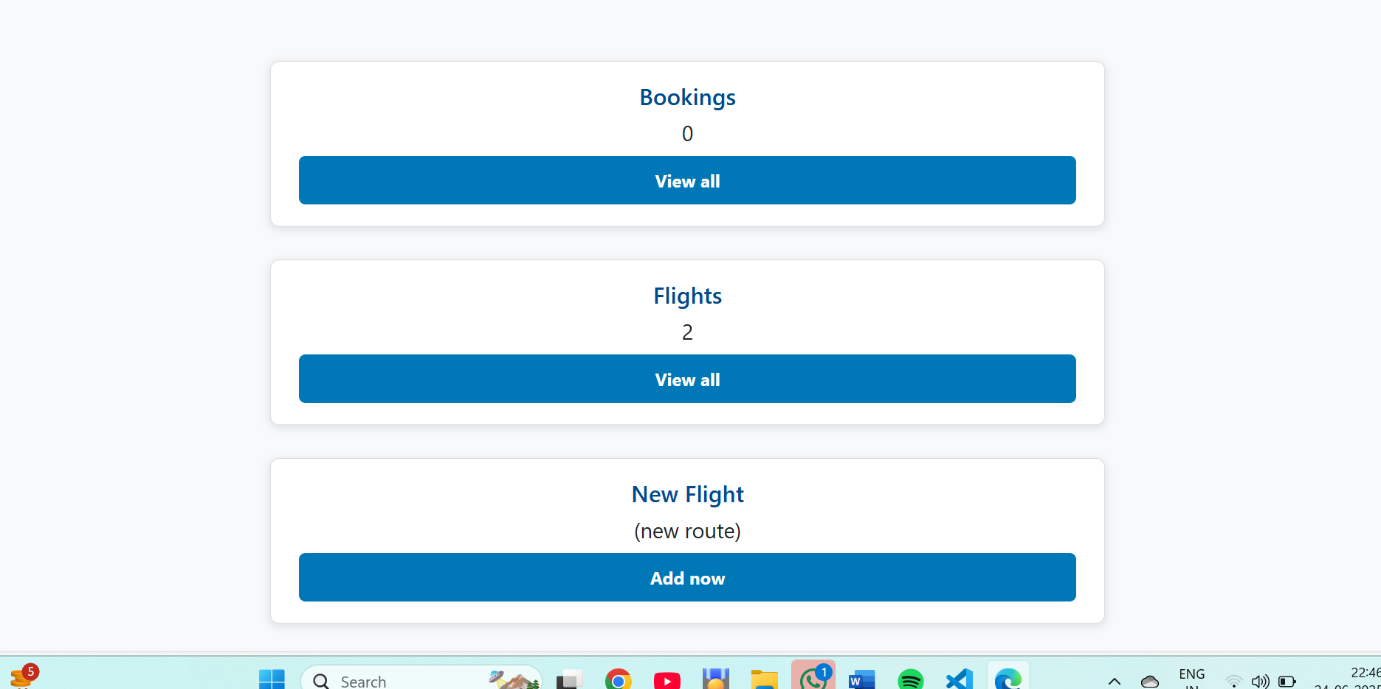


Fig: - Dashboard

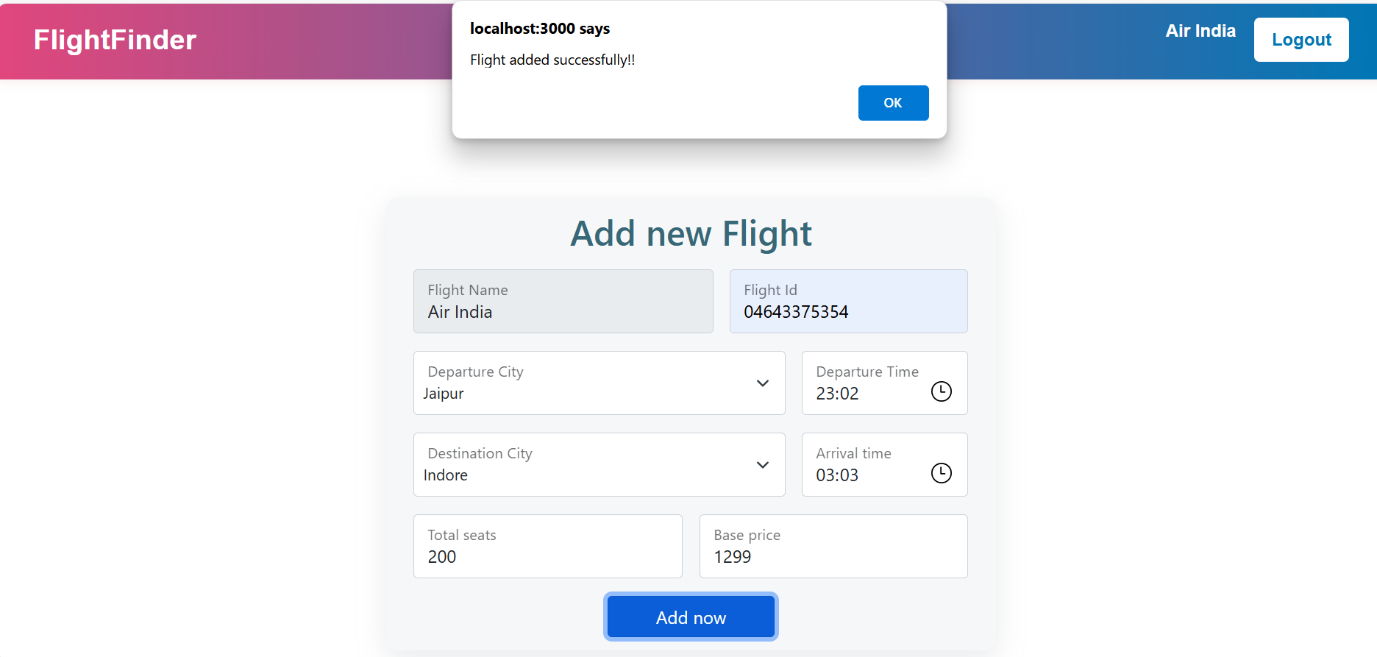
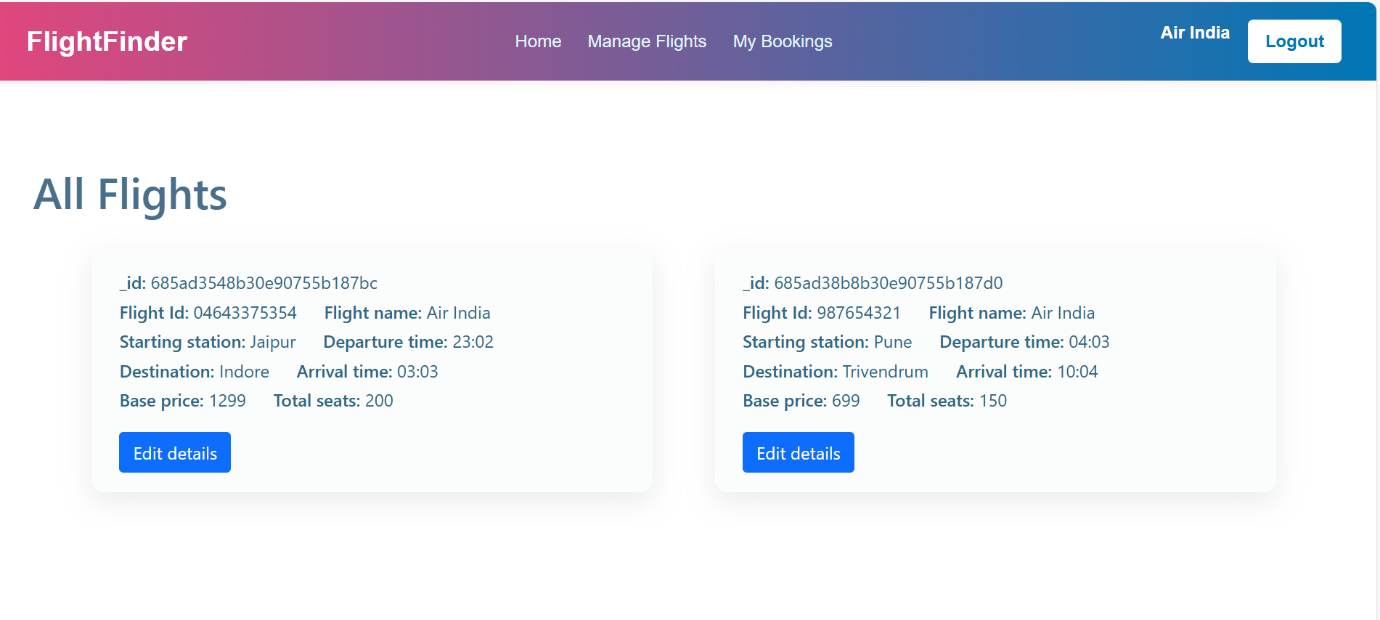
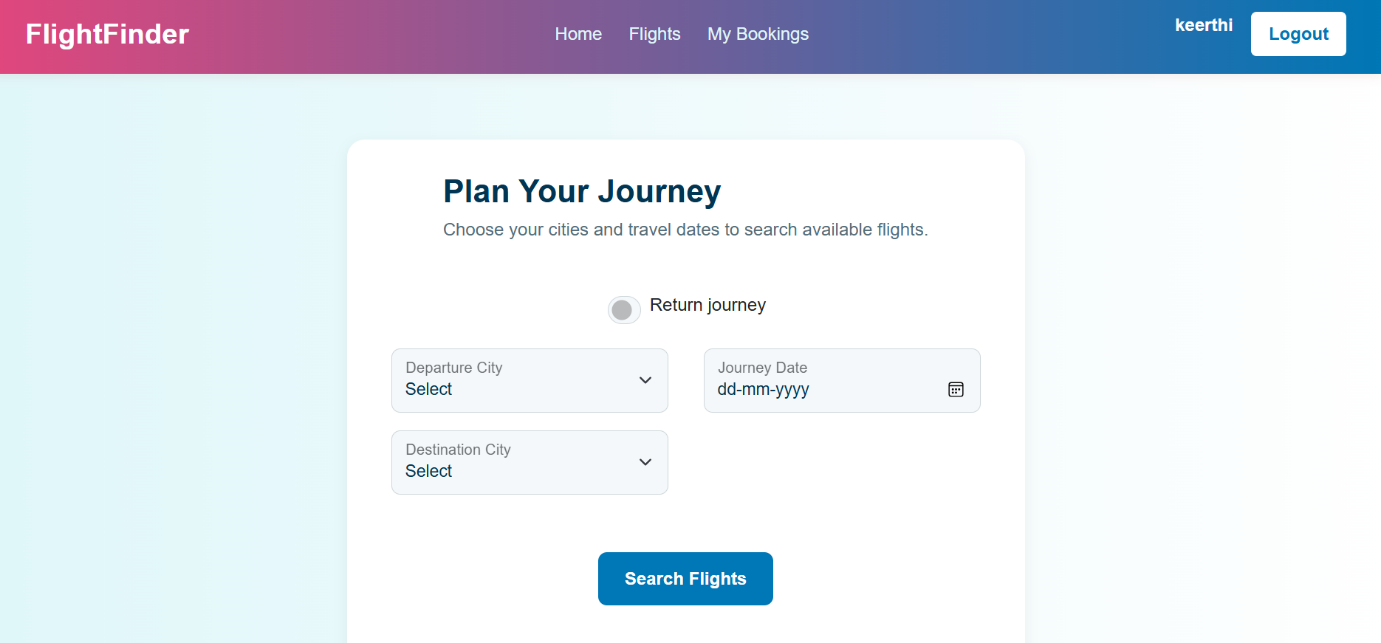


Fig: - Add Flights

 Fig: - Flights

**USER**

 Fig: - Dashboard

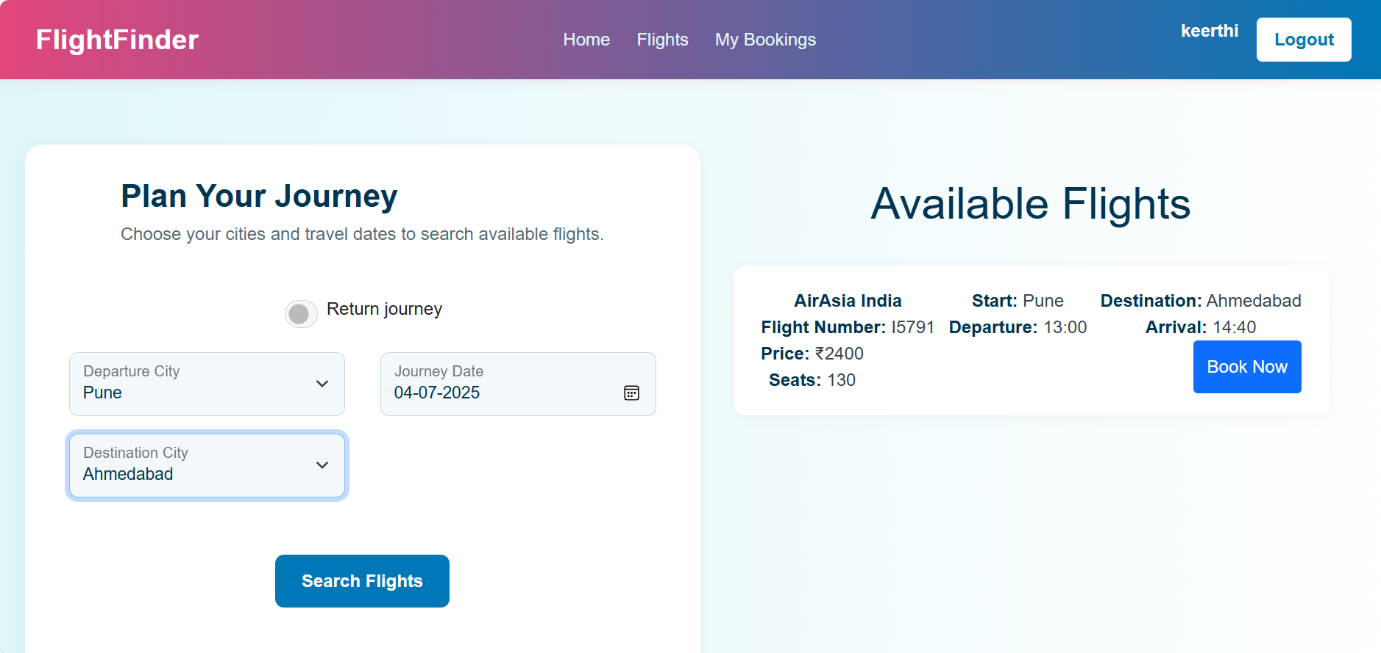


Fig: - Search Flights & Book

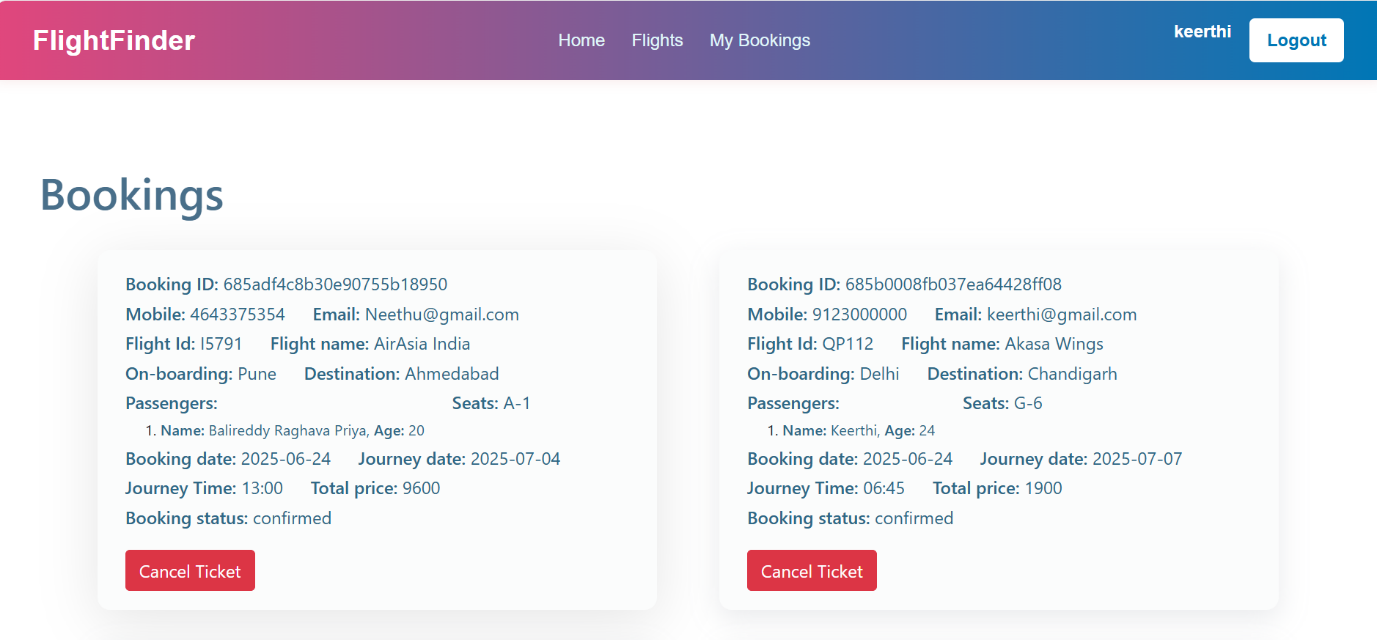


Fig: - Bookings

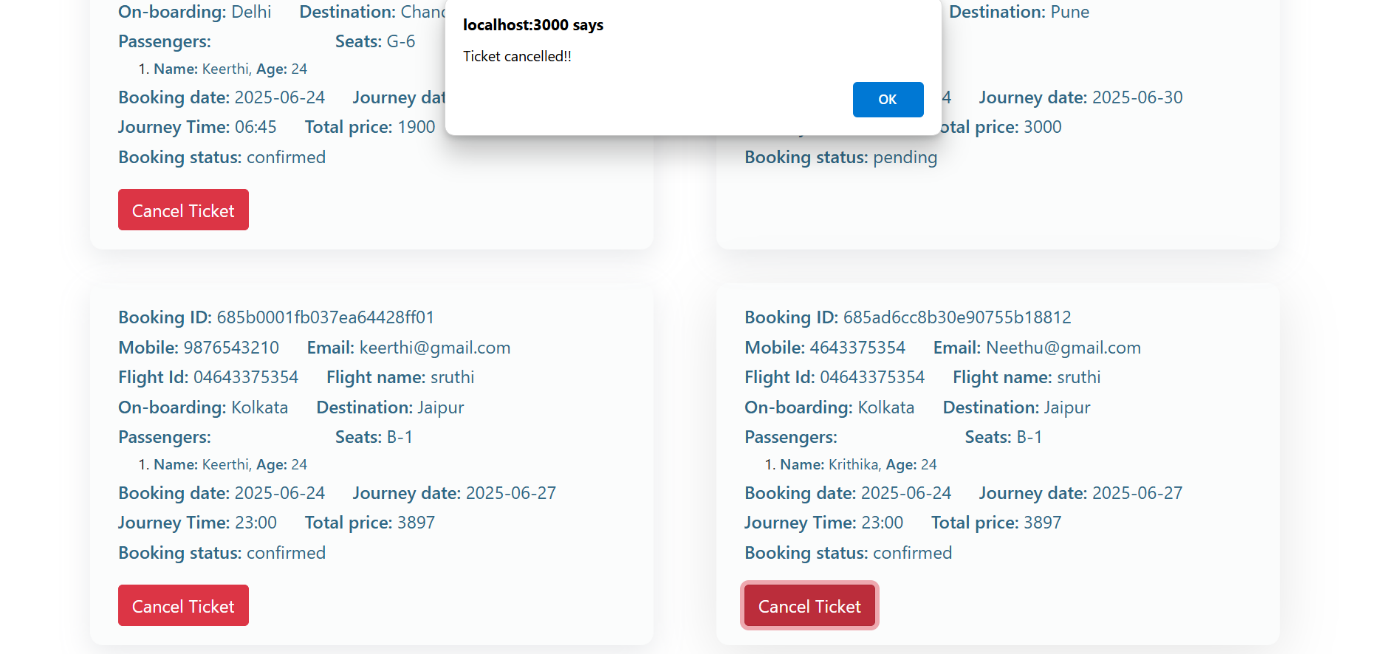


Fig: - Cancel Booking

**12. Known Issues**

Document any known bugs or issues that users or developers should be aware of.

**• Backend (Node.js):**

* Error handling: Uncaught exceptions may cause server crashes.
* MongoDB connection issues: Occasional disconnections require manual restart.
* API rate limiting: No rate limiting implemented, vulnerable to abuse.

**• Frontend (React):**

* Browser compatibility: Issues with older browser versions (IE, Safari).
* Mobile responsiveness: Layout issues on smaller screens.
* State management: Complex state updates may cause rendering errors.

**• Database (MongoDB):**

* Data consistency: Lack of transactions may lead to inconsistent data.
* Query optimization: Slow queries impact performance.
* Indexing: Missing indexes cause slow query execution.

**• Security:**

* Authentication: Weak password hashing (BCrypt).
* Authorization: Incomplete role-based access control.
* XSS protection: Insufficient input sanitization.

**13. Future Enhancements**

Outline potential future features or improvements that could be made to the project.

**Mobile Optimization**

* Create mobile apps for Android and iOS.
* Make the website more user-friendly on phones.
* Add mobile payment options like UPI, Google Pay, etc.

**Better Search and User Experience**

* Show flight suggestions based on user’s past searches.
* Add more filters (like airline, number of stops, etc.).
* Let users save searches and get alerts for price drops.