**Full Stack Development with MERN**

**Project Documentation format**

1. Introduction:

• Project Title: Flight Finder: Navigating Your Air Travel Options

• Team Members: Roles:

- Bethampudi Lakshmi Priyanka (Leader) -Designed and developed the user interface.

- Sameer Sagar Thiragati (Member) - Created API endpoints and connected to MongoDB.

- Molla Afrid (Member) - Coordinated tasks and ensured timely progress.

- Panneru Lovaveerraju (Member) - Performed testing and reported bugs.

**2. Project Overview:**

**• Purpose:**

The primary purpose of this project is to design and develop a full-stack flight booking web application using the MERN (MongoDB, Express.js, React.js, Node.js) stack. This platform aims to streamline the flight booking process for different types of users, including customers, flight operators, and administrators.

- For **customers**, the system provides an easy-to-use interface to search for available flights, compare options, book tickets, and manage their bookings. They can enter their journey details, view available routes, and receive confirmation instantly. This saves time and reduces the need to visit a travel agent or ticket counter physically.

- For **flight operators**, the platform enables them to register and manage flights by adding new routes, setting flight details such as departure and arrival times, seat availability, and pricing. Once approved by the admin, flight operators can track bookings related to their flights and make necessary changes when required.

- For **admins**, the system includes functionality to monitor and control user activity, especially to verify and approve or reject flight operator applications. This helps ensure only valid operators are allowed to list flights on the platform, maintaining the integrity of the system.

Overall, the goal is to build a secure, scalable, and user-friendly application that brings convenience to users and digitizes the traditional flight booking process. The project demonstrates the use of modern web technologies and backend logic to create a functional and real-world solution.

**• Features:**

**The Flight Booking MERN Stack application includes a wide range of features, catering to three different types of users: Customers, Flight Operators, and Admins. Each role is equipped with specific functionalities to ensure a smooth and intuitive experience across the platform.**

**🔹 Customer Features**

* **User Registration & Login: Customers can securely register and log in using their email and password.**
* **Flight Search: Search for flights by specifying origin and destination cities.**
* **One-Way Booking: Book one-way tickets with dynamic price calculation based on seat class and number of passengers.**
* **Passenger Details Entry: Enter names, ages, and genders for all passengers while booking.**
* **Booking History: View all previous and upcoming bookings with journey details and status.**
* **Cancel Booking: Cancel confirmed tickets before the journey date.**

**🔹 Flight Operator Features**

* **Role-based Registration: Flight operators can register and await admin approval before accessing the dashboard.**
* **Add New Flights: Add and manage flights by providing details like flight ID, name, origin, destination, departure/arrival time, price, and seat capacity.**
* **View Bookings: See all bookings made for their flights with passenger and journey details.**
* **Approval System: Operators can only operate once approved by an admin.**

**🔹 Admin Features**

* **Admin Login: Secure login with elevated privileges.**
* **Flight Operator Management: Approve or reject flight operator requests.**
* **User Overview: View all users registered on the platform.**
* **Monitoring: Keep track of the number of flights and bookings in the system.**

**🔹 Common Platform Features**

* **Authentication: Secure login and registration with hashed passwords using bcrypt.**
* **Responsive UI: Clean and responsive interface built with React.**
* **RESTful API Integration: Backend handled with Express.js and MongoDB, using well-structured API endpoints.**
* **Data Validation & Error Handling: Frontend and backend validation for user inputs and proper error messaging.**
* **Real-time Feedback: Success/error alerts for every operation such as login, registration, booking, and flight addition.**

**3. Architecture:**

**• Frontend:**

**The frontend of the Flight Booking System is built using React.js, a popular JavaScript library for building user interfaces. It follows a component-based architecture, promoting modularity, reusability, and separation of concerns.**

**🔹 Key Components**

* **LandingPage.jsx: Allows users to search for available flights by entering origin, destination, and journey date.**
* **Register.jsx & Login.jsx: Handle user registration and authentication based on role (Customer, Admin, Flight Operator).**
* **BookFlight.jsx: Collects passenger details, seat class, and booking preferences, and submits bookings.**
* **FlightAdmin.jsx: Dashboard for flight operators, showing total flights, total bookings, and options to add/manage flights.**
* **FlightBookings.jsx: Lets operators view all bookings for their flights and cancel tickets if needed.**
* **NewFlight.jsx: A form interface for operators to add new flights, including validation and submission.**
* **AdminDashboard.jsx: Allows the admin to approve or reject flight operator registrations.**

**🔹 State Management**

* **React Hooks (useState, useEffect, useContext) are used to manage component state and lifecycle.**
* **Context API is used (via GeneralContext) to share global state like booking date across multiple components.**

**🔹 Routing**

* **React Router DOM handles navigation between pages:**
  + **/ → Landing Page**
  + **/login and /register → Auth Pages**
  + **/flights, /add-flight, /flight-bookings → Operator Pages**
  + **/admin → Admin Dashboard**

**🔹 Styling**

* **Each component has an associated CSS file (e.g., FlightAdmin.css, BookFlight.css) for modular styling.**
* **The UI is responsive and styled using a mix of custom CSS and Bootstrap classes.**

**🔹 Data Flow**

* **Components communicate with the backend via Axios to make HTTP requests (GET, POST, PUT).**
* **Data such as user session info (userId, username, etc.) is stored in localStorage to persist across sessions.**

**🔹 User Experience**

* **The app provides real-time feedback via alert boxes and conditional UI rendering (e.g., approval messages for operators).**
* **Validation messages guide the user through required fields and error cases.**

**• Backend:**

**The backend of the Flight Booking System is developed using Node.js and the Express.js framework. It follows a RESTful API architecture, handling core functionalities like user authentication, flight management, booking operations, and admin controls.**

**🔹 Technology Stack**

* **Node.js: Provides the runtime environment.**
* **Express.js: Framework used to build the HTTP server and define routes.**
* **MongoDB (with Mongoose): NoSQL database used to store user, flight, and booking data.**
* **bcrypt: For secure password hashing.**
* **dotenv: To manage environment variables (like MongoDB URI).**
* **CORS & body-parser: For handling cross-origin requests and parsing incoming request bodies.**

**🔹 Key Modules & Functionality**

1. **Authentication Routes**
   * **POST /register: Registers a new user (Customer, Admin, Flight Operator). Operators are marked as "not-approved" by default.**
   * **POST /login: Validates credentials and returns user data upon successful login.**
2. **Flight Management**
   * **POST /add-flight: Adds a new flight to the database (restricted to approved flight operators).**
   * **GET /fetch-flights: Retrieves all flights.**
   * **GET /fetch-flight/:id: Fetches details of a single flight.**
   * **POST /search-flights: Returns flights matching origin and destination criteria.**
3. **Booking System**
   * **POST /book-ticket: Books a ticket by storing passenger and flight details.**
   * **GET /bookings/:userId: Gets all bookings made by a specific user.**
   * **GET /fetch-bookings: Fetches all bookings (for flight operators/admins).**
   * **PUT /cancel-ticket/:id: Cancels a specific ticket.**
4. **Admin Controls**
   * **GET /fetch-users: Retrieves all registered users.**
   * **POST /approve-operator: Admin approves a pending flight operator.**
   * **POST /reject-operator: Admin rejects (and deletes) a flight operator’s registration.**
5. **User Management**
   * **GET /fetch-user/:id: Fetches details of a specific user.**

**🔹 Database Models (Mongoose)**

* **User Schema: Stores user credentials, type, and approval status.**
* **Flight Schema: Contains flight name, ID, route, times, seats, and price.**
* **Booking Schema: Links user and flight, stores passenger info, ticket status, and journey metadata.**

**🔹 Security & Validations**

* **All passwords are hashed before storage using bcrypt.**
* **Each route performs validation to ensure required fields are provided.**
* **Operators can only manage their own flights and bookings, controlled via username matching logic.**

**🔹 API Design**

* **RESTful principles: each entity (user, flight, booking) has clear, semantically named endpoints.**
* **Returns structured JSON responses for easy integration with the React frontend.**

**• Database:**

**The Flight Booking System uses MongoDB, a NoSQL document-oriented database, to store and manage structured data for users, flights, and bookings. The Mongoose library is used to define schemas and interact with the database through a clear and consistent object data modeling (ODM) approach.**

**🔹 Key Collections & Schemas**

**1. User Schema**

**Stores information for all types of users (customers, admins, and flight operators).**

**{**

**username: String,**

**email: String,**

**usertype: 'customer' | 'admin' | 'flight-operator',**

**password: String (hashed),**

**approval: 'approved' | 'not-approved' | 'rejected',**

**timestamps: true**

**}**

* + **usertype defines the role and permissions.**
  + **approval is used to control access for flight operators.**

1. **Flight Schema**

**Represents individual flights added by operators.**

**{**

**flightName: String,**

**airlineName: String,**

**flightId: String (unique),**

**origin: String,**

**destination: String,**

**departureTime: String,**

**arrivalTime: String,**

**basePrice: Number,**

**totalSeats: Number,**

**timestamps: true**

**}**

* + **Stores essential route and scheduling details.**
  + **Indexed by flightId for easy lookups.**

1. **Booking Schema**

**Manages all user bookings, including seat and passenger data.**

**{**

**user: ObjectId (ref: 'User'),**

**flight: ObjectId (ref: 'Flight'),**

**flightName: String,**

**flightId: String,**

**departure: String,**

**destination: String,**

**email: String,**

**mobile: String,**

**seats: [String],**

**passengers: [**

**{ name: String, age: Number, gender: String }**

**],**

**totalPrice: Number,**

**bookingDate: Date,**

**journeyDate: String,**

**journeyTime: String,**

**seatClass: String,**

**bookingStatus: 'confirmed' | 'cancelled',**

**timestamps: true**

**}**

* + **Links to both user and flight for relational access.**
  + **Stores multiple passengers per booking.**
  + **Tracks status changes like cancellation.**

**🔹 Database Relationships**

* **Users ↔ Bookings: One-to-many — a user can have multiple bookings.**
* **Flights ↔ Bookings: One-to-many — each flight can have multiple related bookings.**
* **Managed via ObjectId references in MongoDB (Mongoose handles population and joins when needed).**

**🔹 Data Validation & Integrity**

* **Mongoose schemas enforce field types and required fields.**
* **Email and flightId are unique.**
* **Approval logic ensures only verified flight operators can insert flight records.**
* **On booking, seat availability and flight association are checked via backend logic.**

**🔹 Sample Query Interactions**

* **Register user: User.create(data)**
* **Add flight: Flight.create(data)**
* **Book ticket: Booking.create(data)**
* **Fetch all flights: Flight.find()**
* **Fetch bookings by user: Booking.find({ user: userId })**
* **Cancel ticket: Booking.findByIdAndUpdate(id, { bookingStatus: 'cancelled' })**

**4. Setup Instructions:**

**• Prerequisites:**

**Before setting up and running the Flight Booking System, ensure the following software dependencies and tools are installed and properly configured on your system:**

**✅ 1. Node.js & npm**

* **Purpose: Runs the backend (Express server) and manages frontend/backend packages.**
* **Required Version: Node.js ≥ 16.x**
* **Install From:** [**https://nodejs.org**](https://nodejs.org)

**To verify installation:**

**node -v**

**npm -v**

**✅ 2. MongoDB**

* **Purpose: NoSQL database to store users, flights, and bookings.**
* **Options:**
  + **Use MongoDB Atlas (cloud-hosted)**
  + **Or install MongoDB locally (**[**https://www.mongodb.com/try/download/community**](https://www.mongodb.com/try/download/community)**)**

**For local MongoDB, make sure the service is running:**

**mongod**

**✅ 3. Git**

* **Purpose: Clone the project repository and manage version control.**
* **Install From:** [**https://git-scm.com/**](https://git-scm.com/)

**To check:**

**git --version**

**✅ 4. Code Editor (Recommended: VS Code)**

* **Purpose: Write and manage source code.**
* **Install From:** [**https://code.visualstudio.com/**](https://code.visualstudio.com/)

**✅ 5. Modern Web Browser**

* **Purpose: Access the React frontend (e.g., Chrome, Firefox, Edge)**

**✅ 6. Environment File (.env)**

* **Purpose: Store sensitive keys like MongoDB connection string.**
* **Must create a .env file in the backend directory:**

**env**

**MONGO\_URI=your\_mongodb\_connection\_string**

**• Installation:**

**Step-by-Step Setup Guide**

**Follow the steps below to install and set up the Flight Booking System on your local machine:**

**✅ 1. Clone the Repository**

**Use Git to clone the project from GitHub:**

**bash**

**git clone https://github.com/your-username/your-repo-name.git**

**cd your-repo-name**

**This project typically has two main folders:**

**client/ – React frontend**

**server/ – Express backend**

**✅ 2. Install Backend Dependencies**

**Navigate to the server directory and install required packages:**

**bash**

**cd server**

**npm install**

**✅ 3. Set Up Environment Variables**

**Create a .env file inside the server folder:**

**bash**

**touch .env**

**Then, add your MongoDB URI:**

**env**

**Copy**

**Edit**

**MONGO\_URI=mongodb+srv://<username>:<password>@your-cluster.mongodb.net/your-db-name**

**Replace <username>, <password>, and your-db-name with actual credentials from your MongoDB Atlas or local MongoDB.**

**✅ 4. Install Frontend Dependencies**

**Now set up the React frontend:**

**bash**

**cd ../client**

**npm install**

**✅ 5. Start the Development Servers**

**Start Backend:**

**bash**

**cd ../server**

**npm start**

**Start Frontend:**

**bash**

**cd ../client**

**npm start**

**The React app should run at http://localhost:3000**

**The Express backend runs at** [**http://localhost:6001**](http://localhost:6001)

**5. Folder Structure:**

**● Client:**

**React Frontend Structure**

**The client/ folder contains the complete frontend logic built with React.js. It handles user interfaces, routing, and API communication with the backend.**

**Below is the typical structure of the client directory:**

**php**

**client/**

**├── public/**

**│ └── index.html # Root HTML file**

**├── src/**

**│ ├── assets/ # Images, icons, and static assets**

**│ ├── components/ # Reusable UI components (e.g., Navbar, Footer)**

**│ ├── context/ # React Context for global state (e.g., auth, booking)**

**│ ├── pages/ # Main pages (e.g., Login, Register, BookFlight)**

**│ ├── styles/ # CSS files for styling components/pages**

**│ ├── App.js # Main component with routing logic**

**│ ├── index.js # Entry point that renders the app**

**│ └── api.js # Axios configuration for backend API calls**

**Key Highlights:**

**Routing:**

**React Router DOM is used for navigating between pages like /login, /register, /book-flight, etc.**

**State Management:**

**Context API manages global values such as selected flight, user details, and ticket booking info.**

**Styling:**

**CSS is modularized under the styles/ folder for pages and components.**

**API Integration:Axios is used in pages to communicate with backend endpoints (e.g., fetch flights, register, login).**

**• Server:**

**Server: Node.js Backend Structure**

**The server/ folder contains the backend logic built with Node.js, Express.js, and MongoDB. It handles routing, authentication, flight and booking operations, and database communication.**

**Here’s the typical structure of the server directory:**

**bash**

**CopyEdit**

**server/**

**├── index.js # Main entry point that sets up Express app and routes**

**├── schemas.js # Mongoose schemas for User, Flight, Booking**

**├── .env # Environment variables (MongoDB URI, etc.)**

**├── package.json # Lists dependencies and scripts**

**Key Components:**

* **index.js  
  Initializes the Express app, connects to MongoDB, and defines all API endpoints:**
  + **/register, /login**
  + **/add-flight, /fetch-flights**
  + **/book-ticket, /fetch-bookings**
  + **/approve-operator, /fetch-users, etc.**
* **schemas.js  
  Defines Mongoose models:**
  + **User: Includes fields like username, email, usertype, and approval**
  + **Flight: Contains flightId, origin, destination, times, etc.**
  + **Booking: Stores booking details, passenger list, and user/flight references**
* **.env  
  Contains environment-sensitive values such as MONGO\_URI used during database connection.**
* **Middleware  
  Express uses middleware for:**
  + **cors: Enables cross-origin requests**
  + **express.json() and bodyParser: Parses JSON payloads from the client**

**6. Running the Application:**

**🔹 Frontend**

**The React-based frontend is located in the client/ directory. To start the frontend development server:**

**bash**

**cd client**

**npm install # Install all required dependencies**

**npm start # Start the React app**

**This will launch the app in your default browser at:**

[**http://localhost:3000**](http://localhost:3000)

**🔹 Backend**

**The backend server is located in the server/ directory. To start the Express server:**

**Bash**

**cd server**

**npm install # Install backend dependencies**

**npm start # Start the Node.js/Express server**

**The backend will run on:**

[**http://localhost:6001**](http://localhost:6001)

**Make sure your .env file is configured with the correct MongoDB URI before running the backend.**

**7. API Documentation:**

This section outlines all RESTful API endpoints exposed by the backend, including request methods, parameters, and sample responses.

**● Authentication:-**

**POST /register**

* **Description**: Register a new user.
* **Request Body**:

json

CopyEdit

{

"username": "john\_doe",

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

"usertype": "customer",

"password": "mypassword"

}

* **Response**:

{

"\_id": "64b09eabc123",

"username": "john\_doe",

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

"usertype": "customer",

"approval": "approved"

}

**POST /login**

* **Description**: Log in an existing user.
* **Request Body**:

{

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

"password": "mypassword"

}

* **Response**:

json

CopyEdit

{

"\_id": "64b09eabc123",

"username": "john\_doe",

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

"usertype": "customer",

"approval": "approved"

}

**●Flight Management:-**

**POST /add-flight**

* **Description**: Add a new flight.
* **Request Body**:

{

"flightName": "IndiGo",

"airlineName": "IndiGo Airlines",

"flightId": "6E123",

"origin": "Chennai",

"destination": "Delhi",

"departureTime": "08:00",

"arrivalTime": "10:30",

"basePrice": 2500,

"totalSeats": 180

}

* **Response**:

{

"message": "Flight added successfully"

}

**GET /fetch-flights**

* **Description**: Get all flights.
* **Response**:

json

CopyEdit

[

{

"\_id": "64b0abc123",

"flightName": "IndiGo",

"origin": "Chennai",

"destination": "Delhi",

...

}

]

**POST /search-flights**

* **Description**: Search flights by origin and destination.
* **Request Body**:

{

"origin": "Chennai",

"destination": "Delhi"

}

* **Response**: Same as above, filtered by criteria.

**GET /fetch-flight/:id**

* **Description**: Get flight details by ID.
* **URL Params**: :id – flight ID
* **Response**:

{

"\_id": "64b0abc123",

"flightName": "IndiGo",

"origin": "Chennai",

...

}

**●Booking Management:-**

**POST /book-ticket**

* **Description**: Book a ticket.
* **Request Body**:

{

"user": "64afxyz",

"flight": "64b0abc123",

"flightName": "IndiGo",

"flightId": "6E123",

"departure": "Chennai",

"destination": "Delhi",

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

"mobile": "9876543210",

"seats": ["E-1", "E-2"],

"passengers": [

{ "name": "John", "age": 30, "gender": "Male" },

{ "name": "Jane", "age": 28, "gender": "Female" }

],

"totalPrice": 5000,

"journeyDate": "2025-07-10",

"journeyTime": "08:00",

"seatClass": "Economy"

}

* **Response**:

{

"message": "Booking successful",

"bookingId": "64b1xyz123"

}

**GET /fetch-bookings**

* **Description**: Get all bookings (admin/operator use).
* **Response**:

**[**

{

"\_id": "64b1xyz123",

"flightName": "IndiGo",

"passengers": [...],

...

}

]

**GET /bookings/:userId**

* **Description**: Get bookings by user ID.
* **URL Params**: :userId
* **Response**: Same as above, filtered.

**PUT /cancel-ticket/:id**

* **Description**: Cancel a booking.
* **URL Params**: :id – booking ID
* **Response**:

{

"message": "Booking cancelled"

}

**●Admin Routes:-**

**GET /fetch-users**

* **Description**: Get all registered users.
* **Response**: List of users.

**POST /approve-operator**

* **Description**: Approve a flight operator.
* **Request Body**:

{

"id": "64afoperator"

}

* **Response**:

{

"message": "Operator approved"

}

**POST /reject-operator**

* **Description**: Reject/delete flight operator.
* **Request Body**:

{

"id": "64afoperator"

}

* **Response**:

{

"message": "Operator rejected and deleted"

}

**8. Authentication:-**

**Authentication in this project is handled through a simple email and password mechanism. When users (customers, admins, or flight operators) register or log in, their credentials are securely verified on the backend.**

**● User Registration and Login**

* **Registration (/register)  
  Users must provide:**
  + **username**
  + **email**
  + **password**
  + **usertype (customer, admin, or flight-operator)**
  + **Passwords are hashed using bcrypt before being stored in MongoDB for security.**
  + **Flight operators are marked as "not-approved" by default until approved by the admin.**
* **Login (/login)**
  + **The user provides their email and password.**
  + **The backend:**
    - **Checks if the user exists.**
    - **Compares the submitted password with the hashed one in the database using bcrypt.compare().**
  + **If successful, the user object is returned to the frontend.**

**● Authorization**

* **Role-based access is determined by the usertype field:**
  + **Customer: Can book and cancel flights.**
  + **Flight Operator: Can add flights, view bookings for their flights.**
  + **Admin: Can approve/reject flight operators and view all data.**
* **Frontend access is restricted using conditional rendering and route protection based on user type and approval status.**

**● Sessions or Tokens?**

**Currently:**

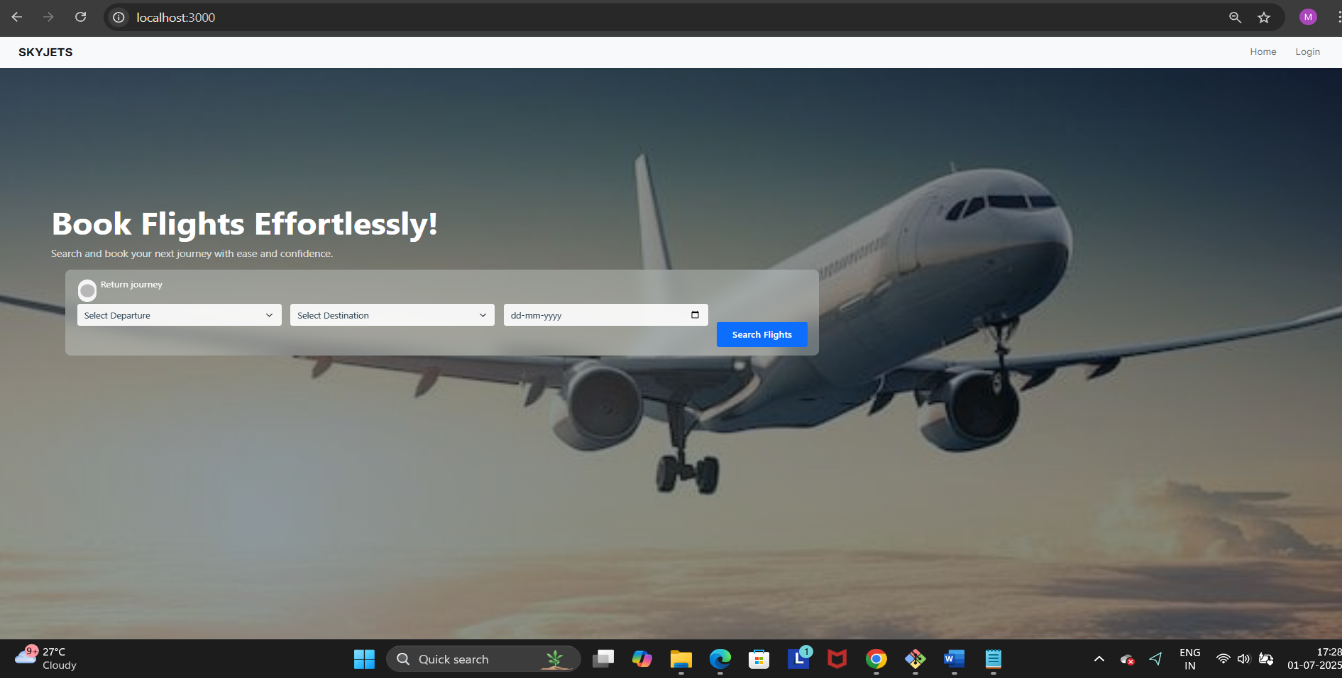
* **This project does not use JWT tokens or session storage on the backend.**
* **User identity is tracked on the frontend using localStorage, where the user’s:**
  + **userId**
  + **username**
  + **usertype**
  + **approval status**

**are stored after login and used across components.**

**●Future Enhancement: For better security and scalability, implementing JWT-based authentication or using session cookies is recommended in production.**

**9. User Interface:-**

**- Here few screenshots of our project to show the UI features:**

****

**A screenshot of a computer

AI-generated content may be incorrect.**

**A computer screen shot of a plane

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**10. Testing:-**

**This project was tested using a combination of manual testing and API testing tools to ensure reliability and correctness of both frontend and backend components.**

**🔹 Frontend Testing**

* **Manual testing was conducted across all user roles: Customer, Admin, and Flight Operator.**
* **Key user interactions tested include:**
  + **Registration and login**
  + **Flight search and booking**
  + **Viewing and cancelling bookings**
  + **Adding and managing flights (for flight operators)**
  + **Admin approval/rejection actions**
* **Browser console logs and React Developer Tools were used to trace UI behavior and state updates.**
* **Responsive behavior and basic styling were tested across different screen sizes.**

**🔹 Backend Testing**

* **All backend APIs were tested using Postman:**
  + **POST /register**
  + **POST /login**
  + **GET /fetch-flights**
  + **POST /add-flight**
  + **POST /book-ticket**
  + **PUT /cancel-ticket/:id**
  + **Admin routes for approving/rejecting flight operators**
* **Ensured proper request validation, status codes, and error handling.**
* **Verified data storage and updates in MongoDB Atlas through the dashboard.**

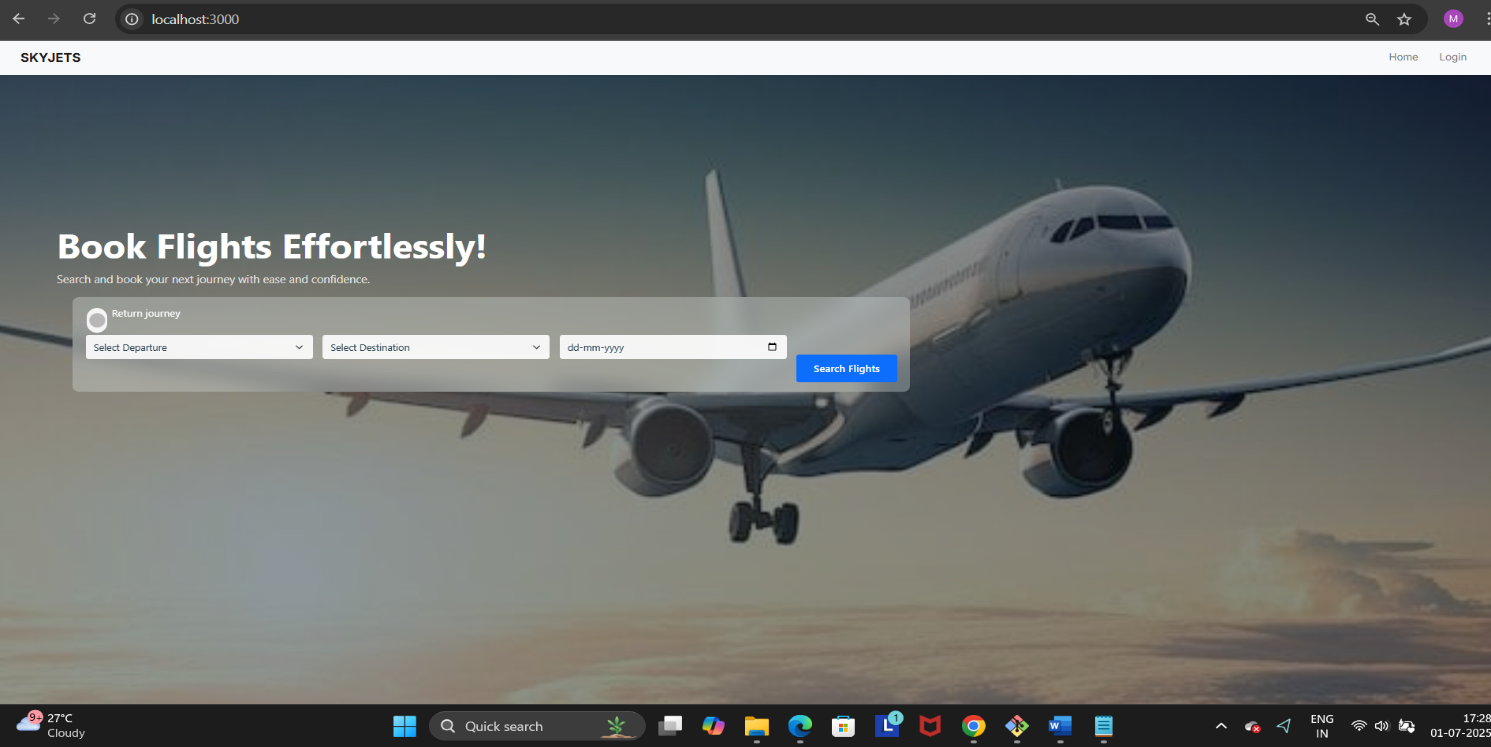
**🔹 Error Handling & Edge Cases**

* **Verified form validations for missing or invalid fields.**
* **Checked behavior for unauthorized access and invalid credentials.**
* **Ensured error messages display clearly on both client and server sides.**

**✅ The system was thoroughly tested for functional accuracy, usability, and stability to deliver a smooth flight booking experience.**

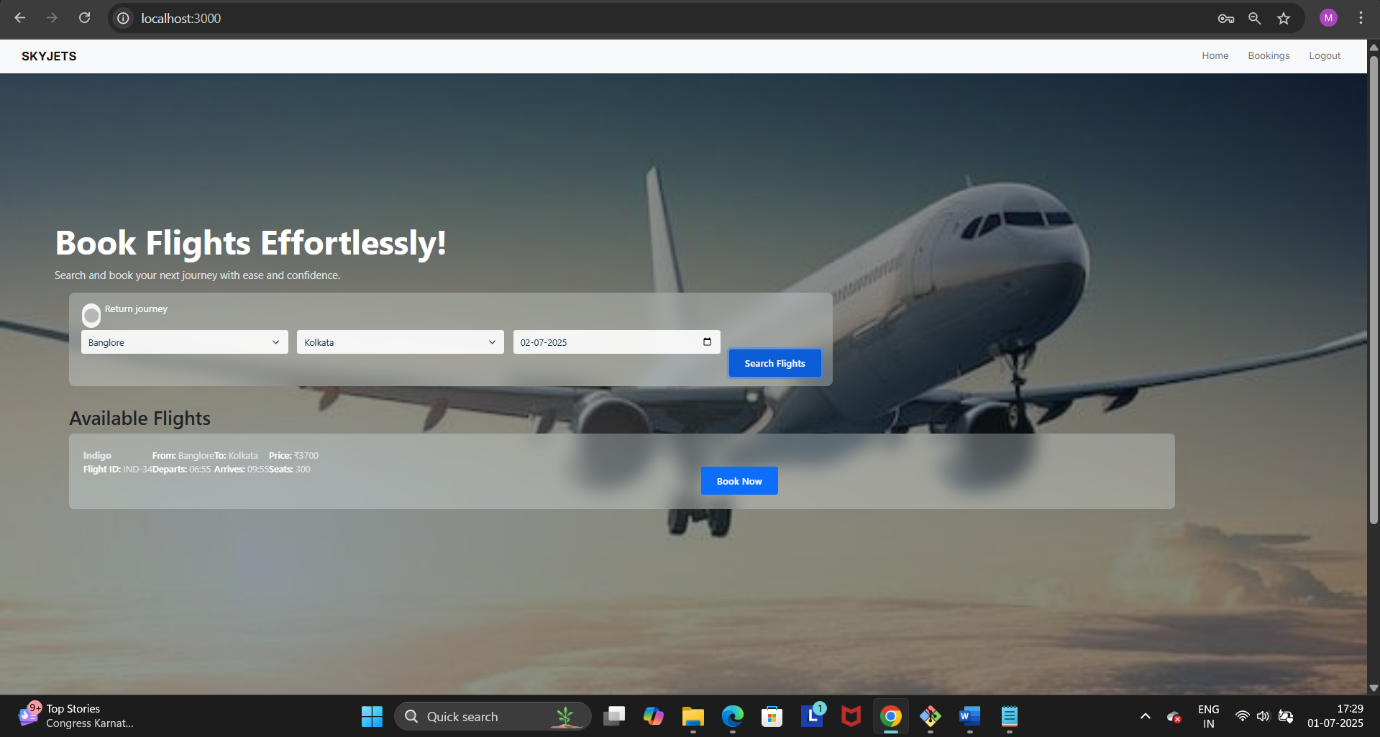
**11. Screenshots or Demo:-**

**Here are the Screenshots of our Project:-**

****

**A screenshot of a computer

AI-generated content may be incorrect.**



A screenshot of a computer

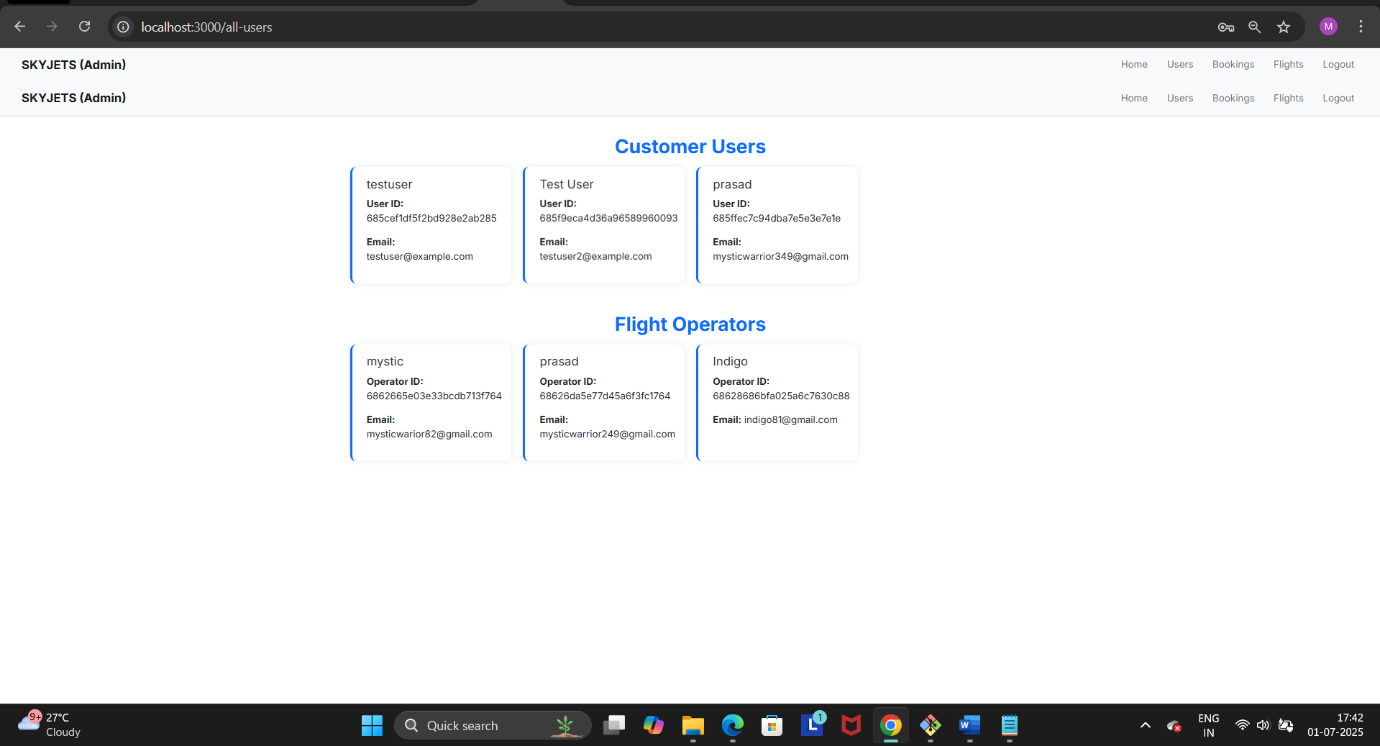
AI-generated content may be incorrect.

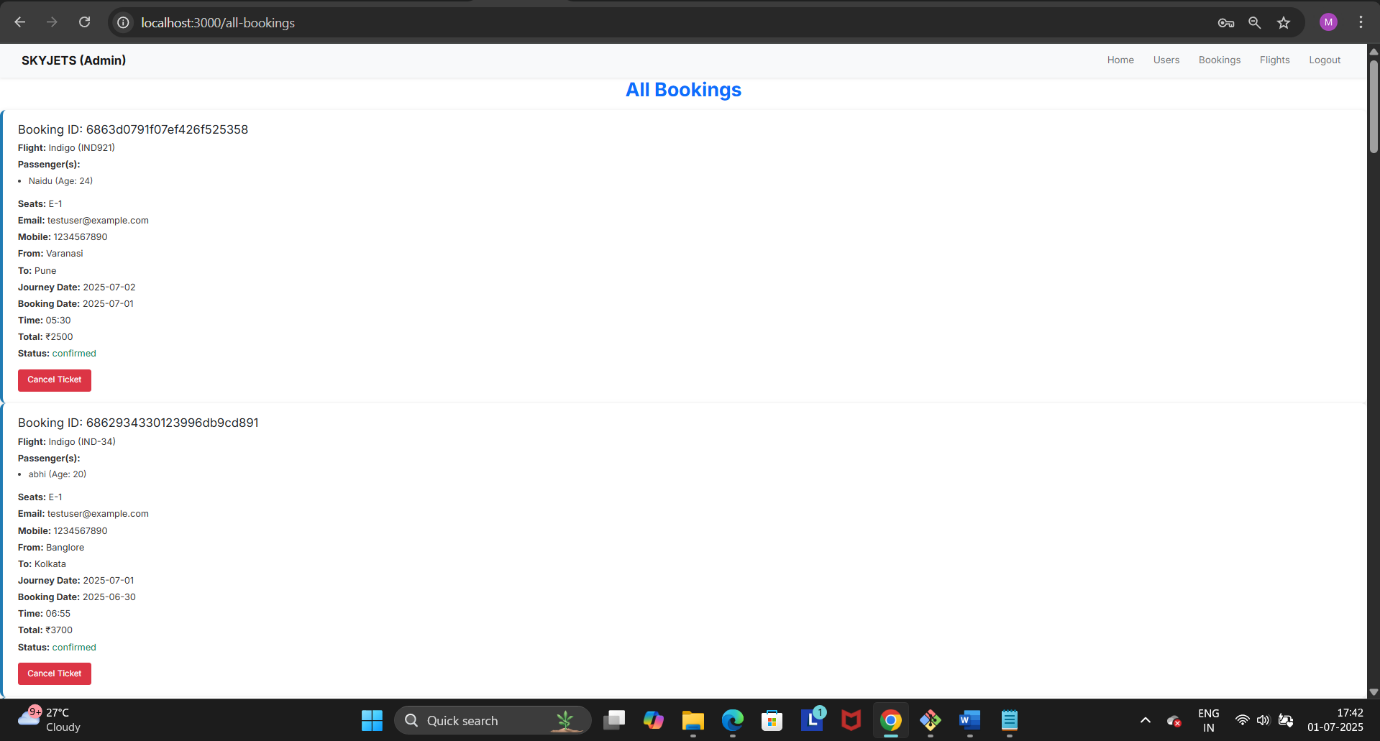
A screenshot of a computer

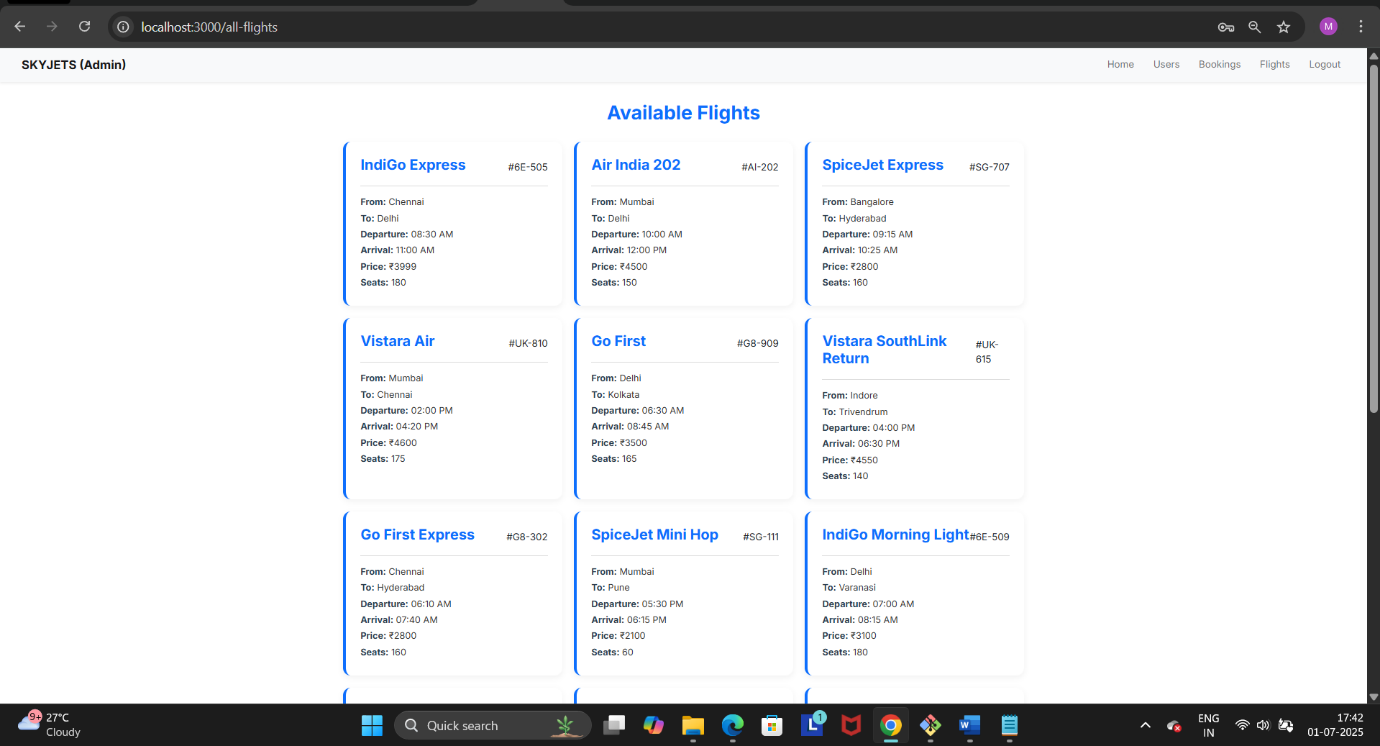
AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.







A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

The demo video of our Project:-

<https://drive.google.com/file/d/1AYd-QCxGxE3NwPcoCpevXwm6e_sLYg-x/view?usp=sharing>

**12. Known Issues:**

**Despite thorough testing, a few known issues still exist in the current version of the application:**

**🔸 1. Flight Operator Approval Delay**

* **After a flight operator registers, their approval status may not refresh automatically on the dashboard.**
* **Workaround: Manual page reload is required after admin approval.**

**🔸 2. No Session Persistence**

* **The app currently uses localStorage for storing user info, which may lead to potential security issues or logout on browser refresh.**
* **Improvement Suggestion: Implement JWT-based authentication with token expiration and secure storage.**

**🔸 3. Lack of Input Validation on Backend**

* **While the frontend validates forms, the backend lacks detailed input validation (e.g., checking seat availability before booking).**
* **Risk: Malformed or unexpected data can be posted via direct API calls.**

**🔸 4. Flight ID Collision**

* **Flight IDs must be unique. If a flight operator accidentally uses an existing flight ID, the system throws a server error.**
* **Improvement Suggestion: Add frontend warning or auto-generate unique IDs.**

**🔸 5. Time Format Inconsistency**

* **Departure and arrival times may appear differently across browsers due to local format interpretation.**
* **Fix: Normalize time display using a standard format (e.g., 24-hour HH:mm) in both frontend and backend.**

**13. Future Enhancements:**

**To improve functionality, scalability, and user experience, the following features and enhancements are planned for future versions of the Flight Booking App:**

**1. JWT-based Authentication & Role Authorization**

* **Replace localStorage with secure JWT tokens for authentication.**
* **Implement route-based protection and session timeout.**

**2. Dynamic Seat Selection UI**

* **Allow users to view and select available seats on an interactive seat map before booking.**
* **Display real-time seat availability.**

**3. Booking Invoice Generation**

* **Generate downloadable PDF tickets/invoices after successful bookings.**
* **Include journey details, passengers, seat numbers, and QR code.**

**4. Dashboard Analytics**

* **Admin and operators can view charts of bookings, revenue, and seat occupancy rates.**
* **Use libraries like Chart.js or Recharts.**

**5. Email Notifications**

* **Send booking confirmations and cancellation alerts via email.**
* **Integrate with services like SendGrid or Nodemailer.**

**6. Flight Schedule Management**

* **Allow flight operators to manage daily/weekly schedules instead of creating separate entries manually.**

**7. Deployment to Cloud Platforms**

* **Deploy the app on platforms like Render, Vercel, or Netlify (frontend) and Railway or MongoDB Atlas (backend).**

**8. Multi-language Support**

* **Add localization to support users in multiple languages.**

**9. Responsive PWA Version**

* **Enhance mobile usability with Progressive Web App (PWA) features such as offline access and push notifications.**

**- These enhancements will help transition the project from an academic model to a production-ready flight booking platform.**