FULL STACK DEVELOPMENT WITH MERN

Project Report

FLIGHT BOOKING APP

Team Members:

Koneti Sasank – 2021115054

Sakthivel K – 2021115088

Saravanan S – 2021115094

Senthil Kumar J - 2021115098

Project overview:

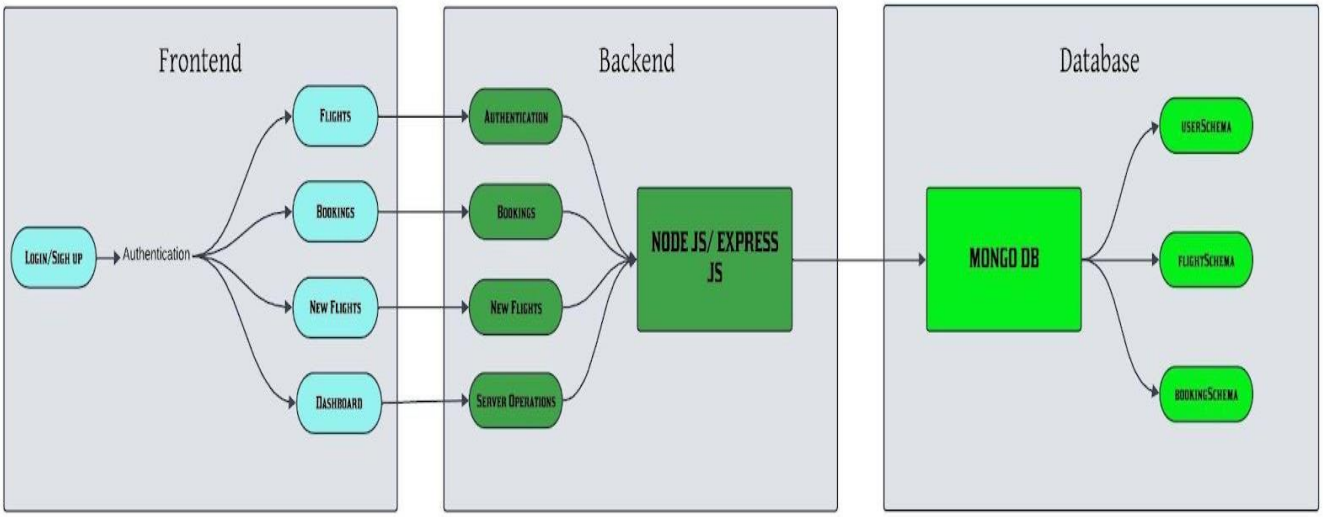
Purpose:

The Flight Booking App is a user-centric platform designed to streamline the flight booking process. It enables users to search for available flights, view real-time availability, and book tickets effortlessly. With an intuitive interface, users can customize their search by selecting their origin, destination, and travel dates, making it easy to find flights that match their preferences. Once a flight is chosen, users can manage bookings, modify their reservations, and receive instant booking confirmations. The app also includes secure user authentication to protect personal data, ensuring only authorized users access their bookings. Additionally, an admin panel provides tools for managing flights and overseeing bookings, giving administrators control over available inventory. The app's ultimate goal is to offer a hassle-free, all-in-one flight booking experience for travelers.

Features:

- **Flight Search**: Users can search for available flights by specifying criteria like origin, destination, and travel dates.
- **Booking Management**: After booking, users can view, update, or cancel their reservations.
- **User Authentication**: Secure login and registration for new users, as well as role-based access for admins.
- **Admin Panel**: Admins can add, update, and delete flights and manage bookings.
- **Payment Integration**: Secure payment processing for flight bookings.

Architecture:



Frontend:

- Developed using React.js with a focus on creating reusable components and utilizing React Hooks for managing state and side effects.
- Components such as FlightSearch, BookingForm, and AdminDashboard make the UI modular and responsive.
- React Router is used to manage page navigation across multiple views like Home, Flights, Bookings, and Admin.

Backend:

- Built with Node.js and Express.js, with API routes structured to handle all key functionalities (e.g., fetching flights, booking flights, user registration).
- Controllers process the business logic, while middleware ensure data validation and authentication.
- Mongoose is used for database interaction with MongoDB.

Database:

- MongoDB hosts collections such as users, flights, bookings, and transactions.
- Each schema is designed for optimized querying, with relationships managed to ensure efficient data retrieval and integrity.

Setup Instructions:

Prerequisites:

- Node.js: Ensure that Node.js (v14 or higher) is installed. You can download it from nodejs.org.
- MongoDB: Set up a MongoDB instance, either locally or using a cloud-based MongoDB provider like MongoDB Atlas.
- Environment Variables: Create a .env file in the root directory to store sensitive information such as:

  DATABASE_URL: MongoDB connection URL

  JWT_SECRET: Secret key for JSON Web Token generation

  API_KEY: Required for third-party services (e.g., payment gateway)

Installation:

- Clone the Repository: Open a terminal and clone the repository to your local machine using the following command:

  git clone https://github.com/harsha-vardhan-reddy-07/Flight-Booking-App-MERN.git

- Navigate to the project's root directory:

  cd Flight-Booking-App-MERN

  Install root-level dependencies:

  npm install

- Install frontend dependencies:

```
cd client
npm install
```

- Install backend dependencies:

```
cd ../server
npm install
```

- Configure Environment Variables:

  Ensure the .env file is set up in the root directory with the necessary secrets and configurations.

  This file should be in the same directory as your package.json files for the backend. Do not include sensitive information in version control.

Database Setup:

- Start MongoDB locally, or ensure your MongoDB Atlas connection string is correct in the .env file.
- If required, create a new database for the application and configure the DATABASE_URL in .env accordingly.
- Optionally seed the database with initial flight data if there's a script provided (e.g., npm run seed).

Frontend and Backend Configuration:

- Ensure the backend server is configured to listen on a specified port (e.g., PORT=5000) in .env.
- Update any API endpoint URLs in the frontend if necessary (e.g., in the client/src/config.js file) to match the backend server's base URL.

Folder Structure:

Client:

- src/components: Contains reusable components like FlightCard, BookingSummary, and LoginForm.
- src/pages: Includes major pages such as Home, Flights, MyBookings, and AdminDashboard.
- src/services: API service functions to fetch data from the backend.

Server:

- server/routes: Defines API routes, such as /api/flights, /api/bookings, and /api/users.
- server/controllers: Contains business logic for each route (e.g., creating bookings, fetching flight details).
- server/models: Mongoose schemas for collections like Flight, User, and Booking.

Running the Application:

- Start the Backend Server:

  Open a terminal, navigate to the server directory, and start the backend server:

  ```
  cd server
  npm start
  ```

  The backend server should now be running, usually on http://localhost:5000 if using the default configuration.

- Start the Frontend Client:

  In a new terminal window, navigate to the client directory, and start the React app:

  ```
  cd client
  npm start
  ```

  This should start the client on http://localhost:3000.

Accessing the App

- Local Testing: Open your browser and go to http://localhost:3000 to access the frontend. The frontend will make requests to the backend server to fetch flight data and handle bookings.
- API Endpoints: Test backend endpoints (if needed) using tools like Postman to verify that the API is functional.

API Documentation:

- **Flight Search Endpoint**:
  - **URL**: /api/flights
  - **Method**: GET
  - **Description**: Retrieves available flights based on search criteria.
  - **Parameters**:
    - origin (required): String, airport code of departure location.
    - destination (required): String, airport code of arrival location.
    - date (optional): Date string for the travel date.

Example Response:


```
[
  {
    "flightNumber": "AA123",
    "origin": "JFK",
    "destination": "LAX",
    "departureTime": "2024-12-12T08:00:00Z",
    "arrivalTime": "2024-12-12T11:00:00Z",
    "price": 299
  }
]
```


Authentication and Authorization:

JWT Authentication:

The Flight Booking App uses JWT (JSON Web Token) authentication to manage user sessions securely. JWTs are a compact and self-contained way to transmit information between parties as a JSON object. This information is digitally signed, so it can be verified and trusted.

Working:

- User Login: When a user logs in, their credentials are verified. If the credentials are valid, the server generates a JWT token.
- Token Generation: The JWT contains the user's unique identifier (e.g., userID), and optionally, role information (e.g., admin or user). This token is signed with a secret key stored on the server (specified in the JWT_SECRET environment variable).
- Token Structure: The JWT consists of three parts – the header, payload, and signature. The payload contains the user's ID and role information, making it easy for the server to identify the user's permissions.
- Token Storage: The generated JWT token is sent to the client, which stores it (typically in localStorage or cookies) for future requests.
- Token Usage: On subsequent requests to protected routes (e.g., viewing bookings, updating profile information), the client includes the JWT in the HTTP headers (usually in the Authorization header as Bearer <token>).
- Token Verification: The server checks the token's validity on each request. If the token is valid, the request proceeds; otherwise, it's rejected, and the user is prompted to log in again.

- Token Expiration: The JWT has an expiration time to enhance security. Once expired, the user must log in again to receive a new token. This helps prevent unauthorized access if a token is compromised.

Authorization (Role-Based Access Control):

Authorization ensures that only users with specific roles can access certain parts of the app. In the Flight Booking App, two main roles are defined:

- User: Regular users who can search for flights, book tickets, and manage their bookings.
- Admin: Users with special privileges, including managing flights, updating flight details, and viewing all user bookings.

Working:

- Role Assignment: Each user is assigned a role upon registration or by an admin. The role information is stored in the user's JWT token payload.
- Access Control Middleware: Middleware functions are created to check user roles. For example, authMiddleware checks if the request contains a valid JWT, while adminMiddleware verifies if the user has an admin role.

Protected Routes:

- User Routes: These routes are accessible to authenticated users. For instance, a user with a valid JWT can view and update their bookings.
- Admin Routes: Certain routes, like those used to create, update, or delete flights, are restricted to admins. Only users with an admin role can access these endpoints. If a non-admin user tries to access these routes, the server returns a "Forbidden" error (HTTP status code 403).

Flow Example:

- User Booking a Flight: When a user tries to book a flight, the server checks for a valid JWT token to ensure the user is authenticated. If valid, the booking process proceeds.
- Admin Managing Flights: When an admin attempts to add or modify flight data, the server checks both the validity of the JWT token and the role encoded within it. If the token is valid and the user's role is admin, the action is permitted.

Security Measures:

- Token Expiration and Refresh: To enhance security, the JWT tokens are given an expiration time. This limits the validity of the token, ensuring that users must re-authenticate after a period.
- HTTPS: For additional security, it is recommended to run the app over HTTPS to prevent token interception during transmission.

- Revoking Tokens: If a user logs out, their token is considered invalid. Implementing a token blacklist or token versioning can further enhance security by invalidating tokens when needed.

User Interface:

Testing:

- **Strategy**
    - Unit Tests:
        - Frontend: Tests individual components (e.g., FlightCard) to verify correct UI rendering and behavior.
        - Backend: Tests isolated functions, like calculatePrice, to ensure accuracy.
    - Integration Tests:
        - API Routes: Tests backend endpoints (e.g., /api/flights) to ensure data flow and accurate responses.
        - Frontend-Backend Interaction: Confirms data received from backend displays correctly on the frontend.
    - End-to-End (E2E) Tests:
        - Tests complete user journeys, like booking a flight, to ensure the app functions end-to-end.
- **Tools**
    - Jest: Used for both frontend and backend testing, allowing unit and integration tests to run efficiently.
    - Supertest: Tests HTTP requests in the backend to validate API responses.
    - React Testing Library: Simulates user actions to verify component rendering and behavior in the frontend.
- **Running Tests**

    Run tests using Jest and Supertest with the command:

```
npm test
```

Known Issues:

- Occasional Delay in Fetching Flights: There may be delays in retrieving flight data due to network latency or slow database responses, which can impact the user experience.
- Minor UI Bugs on Smaller Screen Sizes: Certain UI components may not display correctly or align properly on smaller screens, affecting mobile users.
- Authentication Timeout: Users may experience session timeouts if idle for too long, requiring them to log in again. This is due to token expiration for security but can disrupt user flow.
- Inconsistent Data Refresh: After certain actions (e.g., updating or canceling a booking), the app may not immediately refresh to show the updated data without a manual refresh.
- Limited Error Handling on API Failures: If an API call fails (e.g., due to server downtime), error messages may not display detailed information, leading to a poor user experience.

Future Enhancement:

- Multi-language Support:
    - Implement support for multiple languages to make the app accessible to a global audience. This could include translation for all user interface text, date formatting, and currency conversion based on the user's region.
    - A language selection option can be added to the app settings or detected automatically based on user location or browser preferences.

- Improved Flight Filtering:
    - Enhance the flight search filters to include options for flight duration, number of stops, preferred airlines, cabin class, and departure/arrival time ranges.
    - Adding real-time filtering capabilities can help users find flights more easily by adjusting their criteria without reloading the page.

- Wishlist and Price Alerts:
    - Allow users to add flights to a wishlist and receive notifications or emails when the price changes or new flights matching their criteria are available.
    - This feature can improve user engagement and provide value by helping users book flights at the best prices.

- User Reviews and Ratings:
    - Enable users to rate their flight experiences and leave reviews for airlines, providing feedback to other users and enhancing the decision-making process.
    - This feature could include star ratings, comments, and verified reviews only from users who completed the booking.

- Offline Mode:
    - Enable offline capabilities where users can view previously searched flights or check their booking history even without an internet connection.
    - This can be achieved using local storage and will improve the app's reliability, especially for users in areas with inconsistent connectivity.

- Customer Support Chatbot:
    - Integrate an AI-powered chatbot to assist users with common queries, such as booking assistance, itinerary changes, and flight information.
    - A chatbot can offer 24/7 support, reducing response times and improving the overall user experience.